freedesktop.org

www/ Software/ systemd/ dbus

Edit Page History Repo Info

Back to systemd

This page has been obsoleted and replaced: https://www.freedesktop.org/software/systemd/man/org.freedesktop.systemd1.html.

The D-Bus API of systemd/PID 1

systemd and its auxiliary daemons expose a number of APIs on D-Bus. The following describes the various APIs exposed by the system and service manager itself, and does not cover the auxiliary daemons.

The service manager exposes a number of objects on the bus: one manager object as central entry point for clients and individual objects for each unit and for each queued job. The unit objects each implement a generic Unit interface plus a type-specific interface. For example, service units implement `org.freedesktop.systemd1.Unit` as well as `org.freedesktop.system1.Service`. The manager object can be used to list unit and job objects, or to directly convert a unit name or job id into a bus path of the corresponding D-Bus objects.

Note that properties exposing time values are usually encoded in microseconds (usec) on the bus, even if their corresponding settings in the unit files are in seconds.

In contrast to most of the other services of the systemd suite PID 1 does not use PolicyKit for controlling access to privileged operations, but relies exclusively on the low-level D-Bus policy language. (This is done in order to avoid a cyclic dependency between PolicyKit and systemd/PID 1.) This means that sensitive operations exposed by PID 1 on the bus are generally not available to unprivileged processes directly. However some (such as shutdown/reboot/suspend) are made available via logind's interfaces.

The Manager Object

The main entry point object is available on the fixed `/org/freedesktop/systemd1` object path:

```
$ gdbus introspect --system --dest org.freedesktop.systemd1 --object-path /org/fr

node /org/freedesktop/systemd1 {
  interface org.freedesktop.systemd1.Manager {
    methods:
      GetUnit(in  s name,
              out o unit);
      GetUnitByPID(in  u pid,
                   out o unit);
```

```
        LoadUnit(in  s name,
                 out o unit);
        StartUnit(in  s name,
                  in  s mode,
                  out o job);
        StartUnitReplace(in  s old_unit,
                         in  s new_unit,
                         in  s mode,
                         out o job);
        StopUnit(in  s name,
                 in  s mode,
                 out o job);
        ReloadUnit(in  s name,
                   in  s mode,
                   out o job);
        RestartUnit(in  s name,
                    in  s mode,
                    out o job);
        TryRestartUnit(in  s name,
                       in  s mode,
                       out o job);
        ReloadOrRestartUnit(in  s name,
                            in  s mode,
                            out o job);
        ReloadOrTryRestartUnit(in  s name,
                               in  s mode,
                               out o job);
        KillUnit(in  s name,
                 in  s who,
                 in  i signal);
        ResetFailedUnit(in  s name);
        GetJob(in  u id,
               out o job);
        CancelJob(in  u id);
        ClearJobs();
        ResetFailed();
        ListUnits(out a(ssssssouso) units);
        ListJobs(out a(usssoo) jobs);
        Subscribe();
        Unsubscribe();
        CreateSnapshot(in  s name,
                       in  b cleanup,
                       out o unit);
        RemoveSnapshot(in  s name);
        Reload();
        Reexecute();
        Exit();
        Reboot();
        PowerOff();
        Halt();
        KExec();
        SwitchRoot(in  s new_root,
                   in  s init);
        SetEnvironment(in  as names);
        UnsetEnvironment(in  as names);
        UnsetAndSetEnvironment(in  as unset,
                               in  as set);
        ListUnitFiles(out a(ss) files);
        GetUnitFileState(in  s file,
                         out s state);
        EnableUnitFiles(in  as files,
                        in  b runtime,
                        in  b force,
                        out b carries_install_info,
                        out a(sss) changes);
```

```
      DisableUnitFiles(in  as files,
                       in  b runtime,
                       out a(sss) changes);
      ReenableUnitFiles(in  as files,
                        in  b runtime,
                        in  b force,
                        out b carries_install_info,
                        out a(sss) changes);
      LinkUnitFiles(in  as files,
                    in  b runtime,
                    in  b force,
                    out a(sss) changes);
      PresetUnitFiles(in  as files,
                      in  b runtime,
                      in  b force,
                      out b carries_install_info,
                      out a(sss) changes);
      MaskUnitFiles(in  as files,
                    in  b runtime,
                    in  b force,
                    out a(sss) changes);
      UnmaskUnitFiles(in  as files,
                      in  b runtime,
                      out a(sss) changes);
      SetDefaultTarget(in  as files,
                       out a(sss) changes);
      GetDefaultTarget(out s name);
      SetUnitProperties(in  s name,
                        in  b runtime,
                        in  a(sv) properties);
      StartTransientUnit(in  s name,
                         in  s mode,
                         in  a(sv) properties,
                         in  a(sa(sv)) aux,
                         out o job);
    signals:
      UnitNew(s id,
              o unit);
      UnitRemoved(s id,
                  o unit);
      JobNew(u id,
             o job,
             s unit);
      JobRemoved(u id,
                 o job,
                 s unit,
                 s result);
      StartupFinished(t firmware,
                      t loader,
                      t kernel,
                      t initrd,
                      t userspace,
                      t total);
      UnitFilesChanged();
      Reloading(b active);
    properties:
      readonly s Version = 'systemd 205';
      readonly s Features = '+PAM +LIBWRAP +AUDIT +SELINUX +IMA +SYSVINIT +LIBCRY
      readonly s Tainted = '';
      readonly t FirmwareTimestamp = 0;
      readonly t FirmwareTimestampMonotonic = 0;
      readonly t LoaderTimestamp = 0;
      readonly t LoaderTimestampMonotonic = 0;
      readonly t KernelTimestamp = 0;
      readonly t KernelTimestampMonotonic = 0;
      readonly t InitRDTimestamp = 0;
      readonly t InitRDTimestampMonotonic = 0;
      readonly t UserspaceTimestamp = 1373892700771932;
      readonly t UserspaceTimestampMonotonic = 347348267507;
      readonly t FinishTimestamp = 1373892717621078;
      readonly t FinishTimestampMonotonic = 347365116654;
      readonly t GeneratorsStartTimestamp = 1374345509428734;
      readonly t GeneratorsStartTimestampMonotonic = 500549330609;
      readonly t GeneratorsFinishTimestamp = 1374345509562375;
      readonly t GeneratorsFinishTimestampMonotonic = 500549464250;
      readonly t UnitsLoadStartTimestamp = 1374345509562782;
      readonly t UnitsLoadStartTimestampMonotonic = 500549464657;
      readonly t UnitsLoadFinishTimestamp = 1374345509652212;
      readonly t UnitsLoadFinishTimestampMonotonic = 500549554088;
      readonly t SecurityStartTimestamp = 1374345509562782;
      readonly t SecurityStartTimestampMonotonic = 500549464657;
      readonly t SecurityFinishTimestamp = 1374345509652212;
      readonly t SecurityFinishTimestampMonotonic = 500549554088;
      readwrite s LogLevel = 'info';
      readwrite s LogTarget = 'journal';
      readonly u NNames = 100;
      readonly u NJobs = 0;
      readonly u NInstalledJobs = 266;
      readonly u NFailedJobs = 4;
      readonly d Progress = 1.0;
      readonly as Environment = ['PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/
      readonly b ConfirmSpawn = false;
      readonly b ShowStatus = true;
      readonly as UnitPath = ['/etc/systemd/system', '/run/systemd/system', '/run
      readonly s DefaultStandardOutput = 'journal';
      readonly s DefaultStandardError = 'inherit';
      readwrite t RuntimeWatchdogUSec = 0;
      readwrite t ShutdownWatchdogUSec = 600000000;
      readonly s Virtualization = '';
      readonly s Architecture = 'x86-64';
  };
  interface org.freedesktop.DBus.Properties {
    ...
  };
  interface org.freedesktop.DBus.Peer {
    ...
  };
  interface org.freedesktop.DBus.Introspectable {
    ...
  };
};
```

Security

Read access is generally granted to all clients, but changes may only be made by privileged clients. PolicyKit is not used by this service, and access controlled exclusively via the D-Bus policy enforcement.

Methods

Note that many of the calls exist twice: once on the Manager object, and once on the respective unit objects. This is to optimize access times so that methods that belong to unit objects do not have to be called with a resolved unit path, but can be called with only the unit id, too.

GetUnit() may be used to get the unit object path for a unit name. It takes the unit name and returns the object path. If a unit has not been loaded yet by this name this call will fail.

GetUnitByPID() may be used to get the unit object path of the unit a process ID belongs to. Takes a Unix PID and returns the object path. The PID must refer to an existing process of the system.

LoadUnit() is similar to GetUnit() but will load the unit from disk if possible.

StartUnit() enqeues a start job, and possibly depending jobs. Takes the unit to activate, plus a mode string. The mode needs to be one of replace, fail, isolate, ignore-dependencies, ignore-requirements. If "replace" the call will start the unit and its dependencies, possibly replacing already queued jobs that conflict with this. If "fail" the call will start the unit and its dependencies, but will fail if this would change an already queued job. If "isolate" the call will start the unit in question and terminate all units that aren't dependencies of it. If "ignore-dependencies" it will start a unit but ignore all its dependencies. If "ignore-requirements" it will start a unit but only ignore the requirement dependencies. It is not recommended to make use of the latter two options. Returns the newly created job object.

StartUnitReplace() is similar to **StartUnit()** but replaces a job that is queued for one unit by a job for another.

StopUnit() is similar to **StartUnit()** but stops the specified unit rather than starting it. Note that "isolate" mode is invalid for this call.

ReloadUnit(), **RestartUnit()**, **TryRestartUnit()**, **ReloadOrRestartUnit()**, **ReloadOrTryRestartUnit()** may be used to restart and/or reload a unit, and takes similar arguments as **StartUnit()**. Reloading is done only if the unit is already running and fails otherwise. If a service is restarted that isn't running it will be started, unless the "Try" flavor is used in which case a service that isn't running is not affected by the restart. The "ReloadOrRestart" flavors attempt a reload if the unit supports it and use a restart otherwise.

KillUnit() may be used to kill (i.e. send a signal to) all processes of a unit. Takes the unit name, an enum *who* and a UNIX signal number to send. The who enum is one of "main", "control" or "all". If "main", only the main process of a unit is killed. If "control" only the control process of the unit is killed, if "all" all processes are killed. A "control" process is for example a process that is configured via ExecStop= and is spawned in parallel to the main daemon process, in order to shut it down.

GetJob() returns the job object path for a specific job, identified by its id.

CancelJob() cancels a specific job identified by its numer ID. This operation is also available in the **Cancel()** method of Job objects (see below), and exists primarily to reduce the necessary round trips to execute this operation. Note that this will not have any effect on jobs whose execution has already begun.

ClearJobs() flushes the job queue, removing all jobs that are still queued. Note that this does not have any effect on jobs whose execution has already begun, it only flushes jobs that are queued and have not yet begun execution.

ResetFailedUnit() resets the "failed" state of a specific unit.

ResetFailed() resets the "failed" state of all units.

ListUnits() returns an array with all currently loaded units. Note that units may be known by multiple names at the same name, and hence there might be more unit names loaded than actual units behind them. The array consists of structures with the following elements:

- The primary unit name as string
- The human readable description string
- The load state (i.e. whether the unit file has been loaded successfully)
- The active state (i.e. whether the unit is currently started or not)
- The sub state (a more fine-grained version of the active state that is specific to the unit type, which the active state is not)
- A unit that is being followed in its state by this unit, if there is any, otherwise the empty string.
- The unit object path
- If there is a job queued for the job unit the numeric job id, 0 otherwise
- The job type as string
- The job object path

ListJobs() returns an array with all currently queued jobs. Returns an array consisting of structures with the following elements:

- The numeric job id
- The primary unit name for this job
- The job type as string
- The job state as string
- The job object path
- The unit object path

Subscribe() enables most bus signals to be sent out. Clients which are interested in signals need to call this function. Signals are only sent out if at least one client invoked this function. **Unsubscribe()** undoes the signal subscription that Subscribe() implements. It is not necessary to invoke Unsubscribe() as clients are tracked. Signals are no longer sent out as soon as all clients which previously asked for Subscribe() either closed the bus connection or invoked Unsubscribe().

CreateSnapshot() creates a snapshot unit for the current system state, and stores it under the specified name. It will return the unit object path to the new snapshot. If the cleanup boolean is true the snapshot will be removed automatically when it has been activated, otherwise it remains and can be activated multiple times. Snapshots are not persistent.

RemoveSnapshot() removes a snapshot. This call is also available in the **Remove()** method of Snapshot objects (see below), and exists primarily to reduce the number of required roundtrips for this call.

Reload() may be invoked to reload all unit files.

Reexecute() may be invoked to reexecute the main manager process. It will serialize its state, reexecute, and deserizalize the state again. This is useful for upgrades and is a more comprehensive version of Reload().

Exit() may be invoked to ask the manager to exit. This is not available for the system manager and is useful only for user session managers.

Reboot(), **PowerOff()**, **Halt()**, **KExec()** may be used to ask for immediate reboot, powering down, halt or kexec based reboot of the system. Note that this does not shut down any services and immediately transitions into the reboot process. These functions are normally only called as last step of shutdown, and should not be called directly. To shut down the machine it is a much better choice generally to invoke Reboot() and PoweOff() on the logind manager object. See On logind for more information.

SwitchRoot() may be used to transition to a new root directory. This is intended to be used by initial RAM disks. The call takes two arguments: the new root directory (which needs to be specified), plus an init binary path (which may be left empty, in which case it is automatically searched for). The state of the system manager will be serialized before the transition. After the transition the manager binary on the main system is invoked and replaces the old PID 1. All state will then be deserialized.

SetEnvironment() may be used to alter the environment block that is passed to all spawned processes. Takes a string array with environment variable assignments. Settings passed will override previously set variables.

UnsetEnvironment() may be used to unset environment variables. Takes a string array with environment variable names. All variables specified will be unset (if they have been set previously) and no longer be passed to all spawned processes. This call has no effect for variables that were previously not set, but will not fail in that case.

UnsetAndSetEnvironment() is a combination of UnsetEnvironment() and SetEnvironment(). It takes two lists. The first one is a list of variables to unset, the second one of assignments to set. If a variable is listed in both the variable is set after this call, i.e. the set list overrides the unset list.

ListUnitFiles() returns an array of unit names plus their enablement status. Note that ListUnit() returns a list of units currently loaded into memory, while ListUnitFiles() returns a list of unit *files* that could be found on disk. Note that while most units are read directly from a unit file with the same name some units are not backed by files, and some files (templates) cannot directly be loaded as units but need to be instantiated.

GetUnitFileState() returns the current enablement status of specific unit file.

EnableUnitFiles() may be used to enable one or more units in the system (by creating symlinks to them in /etc or /run). It takes a list of unit files to enable (either just file names or full absolute paths if the unit files are residing outside the usual unit search paths), and two booleans: the first controls whether the unit shall be enabled for runtime only (true, /run), or persistently (false, /etc). The second one controls whether symlinks pointing to other units shall be replaced if necessary. This call returns one boolean and an array with the changes made. The boolean signals whether the unit files contained any enablement information (i.e. an [Install]) section. The changes list consists of structures with three strings: the type of the change (one of *symlink* or *unlink*), the file name of the symlink and the destination of the symlink. Note that most of the following calls return a changes list in the same format.

Similar, **DisableUnitFiles()** disables one or more units in the system, i.e. removes all symlinks to them in /etc and /run.

Similar, **ReenableUnitFiles()** applies the changes to one or more units that would result from disabling and enabling the unit quickly one after the other in an atomic fashion. This is useful to apply updated [Install] information contained in unit files.

Similar, **LinkUnitFiles()** links unit files (that are located outside of the usual unit search paths) into the unit search path.

Similar, **PresetUnitFiles()** enables/disables one or more units file according to the preset policy. See Presets for more information.

Similar, **MaskUnitFiles()** masks unit files, and **UnmaskUnitFiles()** unmasks them again.

SetDefaultTarget() changes the `default.target` link. See bootup(7) for more information.

GetDefaultTarget() retrieves the name of the unit to which `default.target` is aliased.

SetUnitProperties() may be used to modify certain unit properties at runtime. Not all properties may be changed at runtime, but many resource management settings (primarily those in systemd.cgroup(5)) may. The changes are applied instantly, and stored on disk for future boots, unless *runtime* is true, in which case the settings only apply until the next reboot. *name* is the name of the unit to modify. *properties* are the settings to set, encoded as an array of property name and value pairs. Note that this is not a dictionary! Note that when setting array properties with this call this usually results in appending to the pre-configured array. To reset the configured arrays set the property to an empty array first, then append to it.

StartTransientUnit() may be used to create and start a transient unit, which will be released as soon as it is not running or referenced anymore or the system is rebooted. **name** is the unit name including suffix, and must be unique. **mode** is the same as in **StartUnit()**, **properties** contains properties of the unit, specified like in **SetUnitProperties()**. **aux** is currently unused and should be passed as empty array. See the New Control Group Interfaces for more information how to make use of this functionality for resource control purposes.

Signals

Note that most signals are sent out only after **Subscribe()** has been invoked by at least one client. Make sure to invoke this call when subscribing to these signals!

UnitNew() and **UnitRemoved()** are sent out each time a new unit is loaded or unloaded. Note that this has little to do with whether a unit is available on disk or not, and simply reflects the units that are currently loaded into memory. The signals take two parameters: the primary unit name and the object path.

JobNew() and **JobRemoved()** are sent out each time a new job is queued or dequeued. Both signals take the numeric job ID, the bus path and the primary unit name for this job as argument. **JobRemoved()** also includes a result string, being one of `done`, `canceled`, `timeout`, `failed`, `dependency`, `skipped`. `done` indicates successful execution of a job. `canceled` indicates that a job has been canceled (via CancelJob() above) before it finished execution (this doesn't necessarily mean though that the job operation is actually cancelled too, see above). `timeout` indicates that the job timeout was reached. `failed` indicates that the job failed. `dependency` indicates that a job this job has been depending on failed and the job hence has been removed too. `skipped` indicates that a job was skipped because it didn't apply to the units current state.

StartupFinished() is sent out when startup finished. It carries six usec timespan values each indicating how much boot time has been spent in the firmware (if known), in the boot loader (if known), in the kernel initialization phase, in the initrd (if known), in userspace and in total. These values may also be calculated from the FirmwareTimestampMonotonic, LoaderTimestampMonotonic, InitRDTimestampMonotonic, UserspaceTimestampMonotonic, FinishTimestampMonotonic properties (see below).

UnitFilesChanged() is sent out each time the list of enabled or masked unit files on disk have changed.

Reloading() is sent out immediately before a daemon reload is done (with the boolean parameter set to True) and after a daemon reload is completed (with the boolean parameter set to False). This may be used by UIs to optimize UI updates.

Properties

Most properties simply reflect the respective options in `/etc/systemd/system.conf` and the kernel command line. The others:

Version encodes the version string of the running systemd instance. Note that the version string is purely informational, it should not be parsed, one may not assume the version to be formatted in any particular way. We take the liberty to change the versioning scheme at any time and it is not part of the API.

Features encodes the features that have been enabled resp. disabled for this build. Enabled options are prefixed with +, disabled options with -.

Tainted encodes a couple of taint flags, as colon separated list. When systemd detects it is run on a system with certain problems it will set an appropriate taint flag. Taints may be used to lower the chance of bogus bug reports. The following taints are currently known: `split-usr`, `mtab-not-symlink`, `cgroups-missing`, `local-hwclock`. `split-usr` is set if /usr is not pre-mounted when systemd is first invoked. See Booting Without /usr is Broken for details why this is bad. `mtab-not-symlink` indicates that `/etc/mtab` is not a symlink to `/proc/self/mounts` as required. `cgroups-missing` indicates that control groups have not been enabled in the kernel. `local-hwclock` indicates that the local RTC is configured to be in local time rather than UTC.

FirmwareTimestamp, **FirmwareTimestampMonotonic**, **LoaderTimestamp**, **LoaderTimestampMonotonic**, **KernelTimestamp**, **KernelTimestampMonotonic**, **InitRDTimestamp**, **InitRDTimestampMonotonic**, **UserspaceTimestamp**, **UserspaceTimestampMonotonic**, **FinishTimestamp**, **FinishTimestampMonotonic** encode CLOCK_REALTIME resp. CLOCK_MONOTONIC usec timestamps taken when the firmware first began execution, when the boot loader first began execution, when the kernel first began execution, when the initrd first began execution, when the main systemd instance began execution and finally, when all queued startup jobs finished execution. These values are useful for determining boot-time performance. Note that as monotonic time begins with the kernel startup the KernelTimestampMonotonic timestamp will always be 0, and FirmwareTimestampMonotonic and LoaderTimestampMonotonic are to be read as negative values. Also, not all fields are available, depending on the used firmware, boot loader or initrd implementation. In these cases the resp. pairs of timestamps are both 0, indicating that no data is available.

Similar, the **SecurityStartTimestamp**, **GeneratorsStartTimestamp** and **LoadUnitTimestamp** (plus their monotonic and stop counterparts) expose performance data for uploading the security policies to the kernel (such as the SELinux, IMA, or SMACK policies), for running the generator tools and for loading the unit files.

NNames encodes how many unit names are currently known. This only includes names of units that are currently loaded and can be more than actually loaded units since units may have more than one name.

NJobs encodes how many jobs are currently queued.

NInstalledJobs encodes how many jobs have ever been queued in total.

NFailedJobs encodes how many jobs have ever failed in total.

Progress encodes boot progress as floating point value between 0.0 and 1.0. This value begins at 0.0 at early-boot and ends at 1.0 when boot is finished and is based on the number of executed and queued jobs. After startup this field is always 1.0 indicating a finished boot.

Environment encodes the environment block passed to all executed services. It may be altered with bus calls such as SetEnvironment() (see above).

UnitPath encodes the currently active unit file search path. It is an array of strings, each being one file system path.

Virtualization contains a short ID string describing the virtualization technology the system runs in. On bare-metal hardware this is the empty string, otherwise an identifier such as "kvm", "vmware" and so on. For a full list of IDs see systemd-detect-virt(1). Note that only the "innermost" virtualization technology is exported here. This detects both full-machine virtualizations (VMs) and shared-kernel virtualization (containers).

Architecture contains a short ID string describing the architecture the systemd instance is running on. This follows the same vocabulary as ConditionArchitectures=.

ControlGroup contains the root control group path of this system manager. Note that the root path is encoded as empty string here (not as "/"!), so that it can be appended to `/sys/fs/cgroup/systemd` easily. This value will be set to the empty string for the host instance, and some other string for container instances.

Unit Objects

All Unit objects implement the generic `org.freedesktop.systemd1.Unit` interface. Depending on the unit type they also implement one unit-type-specific interface, as described below.

```
$ gdbus introspect --system --dest org.freedesktop.systemd1 --object-path /org/fr
node /org/freedesktop/systemd1/unit/avahi_2ddaemon_2eservice {
  interface org.freedesktop.systemd1.Unit {
    methods:
      Start(in  s mode,
            out o job);
      Stop(in  s mode,
           out o job);
      Reload(in  s mode,
             out o job);
      Restart(in  s mode,
              out o job);
      TryRestart(in  s mode,
                 out o job);
      ReloadOrRestart(in  s mode,
                      out o job);
      ReloadOrTryRestart(in  s mode,
                         out o job);
      Kill(in  s who,
           in  i signal);
      ResetFailed();
      SetProperties(in  b runtime,
                    in  a(sv) properties);
    signals:
    properties:
      readonly s Id = 'avahi-daemon.service';
      readonly as Names = ['avahi-daemon.service'];
      readonly s Following = '';
      readonly as Requires = ['avahi-daemon.socket', 'dbus.socket', 'basic.target
      readonly as RequiresOverridable = [];
      readonly as Requisite = [];
      readonly as RequisiteOverridable = [];
      readonly as Wants = [];
      readonly as BindsTo = [];
      readonly as PartOf = [];
      readonly as RequiredBy = [];
      readonly as RequiredByOverridable = [];
      readonly as WantedBy = ['multi-user.target'];
      readonly as BoundBy = [];
      readonly as ConsistsOf = [];
      readonly as Conflicts = ['shutdown.target'];
      readonly as ConflictedBy = [];
      readonly as Before = ['shutdown.target', 'multi-user.target'];
      readonly as After = ['avahi-daemon.socket', 'systemd-journald.socket', 'dbu
      readonly as OnFailure = [];
      readonly as Triggers = [];
      readonly as TriggeredBy = ['avahi-daemon.socket'];
      readonly as PropagatesReloadTo = [];
      readonly as ReloadPropagatedFrom = [];
      readonly as RequiresMountsFor = [];
      readonly s Description = 'Avahi mDNS/DNS-SD Stack';
      readonly s SourcePath = '';
      readonly as DropInPaths = [];
      readonly as Documentation = [];
      readonly s LoadState = 'loaded';
      readonly s ActiveState = 'active';
      readonly s SubState = 'running';
      readonly s FragmentPath = '/usr/lib/systemd/system/avahi-daemon.service';
      readonly s UnitFileState = 'enabled';
      readonly t InactiveExitTimestamp = 1368576738559539;
      readonly t InactiveExitTimestampMonotonic = 22561234;
      readonly t ActiveEnterTimestamp = 1368576738648314;
      readonly t ActiveEnterTimestampMonotonic = 22650009;
      readonly t ActiveExitTimestamp = 0;
      readonly t ActiveExitTimestampMonotonic = 0;
      readonly t InactiveEnterTimestamp = 0;
      readonly t InactiveEnterTimestampMonotonic = 0;
      readonly b CanStart = true;
      readonly b CanStop = true;
      readonly b CanReload = true;
      readonly b CanIsolate = false;
      readonly (uo) Job = (0, '/org/freedesktop/systemd1/unit/avahi_2ddaemon_2ese
      readonly b StopWhenUnneeded = false;
      readonly b RefuseManualStart = false;
      readonly b RefuseManualStop = false;
      readonly b AllowIsolate = false;
      readonly b DefaultDependencies = true;
      readonly b OnFailureIsolate = false;
      readonly b IgnoreOnIsolate = false;
      readonly b IgnoreOnSnapshot = false;
      readonly b NeedDaemonReload = false;
      readonly t JobTimeoutUSec = 0;
      readonly t ConditionTimestamp = 1368576738557978;
      readonly t ConditionTimestampMonotonic = 22559674;
      readonly b ConditionResult = true;
      readonly a(sbbsi) Conditions = [('ConditionVirtualization, false, false, 'n
      readonly (ss) LoadError = ('', '');
      readonly b Transient = false;
  };
  interface org.freedesktop.systemd1.Service {
    ...
  };
  interface org.freedesktop.DBus.Properties {
    ...
  };
  interface org.freedesktop.DBus.Peer {
    ...
  };
  interface org.freedesktop.DBus.Introspectable {
    ...
  };
};
```

Methods

Start(), **Stop()**, **Reload()**, **Restart()**, **TryRestart()**, **ReloadOrRestart()**, **ReloadOrTryRestart()**, **Kill()**, **ResetFailed()** and **SetProperties()** implement the same operation as the respective method calls on the Manager object (see above), however operate on the unit object and hence do not take a unit name parameter. Invoking the methods directly on the Manager object has the advantage of not requiring a **GetUnit()** call to get the unit object for a specific unit name. Calling the methods on the Manager object is hence a round trip optimization.

Properties

Id contains the primary name of the unit.

Names contains all names of the unit, including the primary name that is also exposed in **Id**.

Following either contains the empty string or contains the name of another unit that this unit follows in state. This is used for some device units which reflect the unit state machine of another unit, and which other unit this is might possibly change.

Requires, **RequiresOverridable**, **Requisite**, **RequisiteOverridable**, **Wants**, **BindsTo**, **RequiredBy**, **RequiredByOverridable**, **WantedBy**, **BoundBy**, **Conflicts**, **ConflictedBy**, **Before**, **After**, **OnFailure**, **Triggers**, **TriggeredBy**, **PropagatesReloadTo**, **RequiresMountsFor** contain arrays which encode the dependencies and their inverse dependencies (where this applies), as configured in the unit file or determined automatically.

Description contains the human readable description string for the unit.

SourcePath contains the path to a configuration file this unit is automatically generated from in case it is not a native unit (in which case it contains the empty string). For example, all mount units generated from `/etc/fstab` have this field set to this value.

Documentation contains a string array with URLs of documentation for this unit.

LoadState contains a state value that reflects whether the configuration file of this unit has been loaded. The following states are currently defined: *loaded*, *error*, *masked*. *loaded* indicates that the configuration was successfully loaded. *error* indicates that the configuration failed to load, the **LoadError** field (see below) contains information about the cause of this failure. *masked* indicates that the unit is currently masked out (i.e. symlinked to /dev/null or suchlike). Note that the **LoadState** is fully orthogonal to the **ActiveState** (see below) as units without valid loaded configuration might be active (because configuration might have been reloaded at a time where a unit was already active).

ActiveState contains a state value that reflects whether the unit is currently active or not. The following states are currently defined: *active*, *reloading*, *inactive*, *failed*, *activating*, *deactivating*. *active* indicates that unit is active (obviously...). *reloading* indicates that the unit is active and currently reloading its configuration. *inactive* indicates that it is inactive and the previous run was successful or no previous run has taken place yet. *failed* indicates that it is inactive and the previous run was not successful (more information about the reason for this is available on the unit type specific interfaces, for example for services in the **Result** property, see below). *activating* indicates that the unit has previously been inactive but is currently in the process of entering an active state. Conversely *deactivating* indicates that the unit is currently in the process of deactivation.

SubState encodes states of the same state machine that **ActiveState** covers, but knows more fine-grained states that are unit-type-specific. Where **ActiveState** only covers six high-level states, **SubState** covers possibly many more low-level unit-type-specific states that are mapped to the six high-level states. Note that multiple low-level states might map to the same high-level state, but not vice versa. Not all high-level states have low-level counterparts on all unit types. At this point the low-level states are not documented here, and are more likely to be extended later on than the common high-level states explained above.

FragmentPath contains the unit file path this unit was read from, if there is any (if not this contains the empty string).

UnitFileState encodes the install state of the unit file of **FragmentPath**. It currently knows the following states: *enabled*, *enabled-runtime*, *linked*, *linked-runtime*, *masked*, *masked-runtime*, *static*, *disabled*, *invalid*. *enabled* indicates that a unit file is permanently enabled. *enable-runtime* indicates the unit file is only temporarily enabled, and will no longer be enabled after a reboot (that means, it is enabled via /run symlinks, rather than /etc). *linked* indicates that a unit is linked into /etc permanently, *linked* indicates that a unit is linked into /run temporarily (until the next reboot). *masked* indicates that the unit file is masked permanently, *masked-runtime* indicates that it is only temporarily masked in /run, until the next reboot. *static* indicates that the unit is statically enabled, i.e. always enabled and doesn't need to be enabled explicitly. *invalid* indicates that it could not be determined whether the unit file is enabled.

InactiveExitTimestamp, **InactiveExitTimestampMonotonic**, **ActiveEnterTimestamp**, **ActiveEnterTimestampMonotonic**, **ActiveExitTimestamp**, **ActiveExitTimestampMonotonic**, **InactiveEnterTimestamp**, **InactiveEnterTimestampMonotonic** contain CLOCK_REALTIME and CLOCK_MONOTONIC 64bit usec timestamps of the last time a unit left the inactive state, entered the active state, exited the active state, or entered an inactive state. These are the points in time where the unit transitioned *inactive/failed → activating*, *activating → active*, *active → deactivating*, and finally *deactivating → inactive/failed*. The fields are 0 in case such a transition has not been recording on this boot yet.

CanStart, **CanStop**, **CanReload** encodes as booleans whether the unit supports the start, stop or reload operations. Even if a unit supports such an operation the client might not necessary have the right privileges to execute them.

CanIsolate encodes as boolean whether the unit may be started in isolation mode.

Job encodes the job ID and job object path of the job currently scheduled or executed for this unit, if there is any. If no job is scheduled or executed the job id field will be 0.

StopWhenUnneeded, **RefuseManualStart**, **RefuseManualStop**, **AllowIsolate**, **DefaultDependencies**, **OnFailureIsolate**, **IgnoreOnIsolate**, **IgnoreOnSnapshot** map directly to the corresponding configuration booleans in the unit file.

DefaultControlGroup contains the main control group of this unit as a string. This refers to a group in systemd's own *name=systemd* hierarchy, which systemd uses to watch and manipulate the unit and all its processes.

NeedDaemonReload is a boolean that indicates whether the configuration file this unit is loaded from (i.e. **FragmentPath** or **SourcePath**) has changed since the configuration was read and hence whether a configuration reload is recommended.

JobTimeoutUSec maps directly to the corresponding configuration setting in the unit file.

ConditionTimestamp and **ConditionTimestampMonotonic** contain the CLOCK_REALTIME/CLOCK_MONOTONIC usec timestamps of the last time the configured conditions of the unit have been checked, or 0 if they have never been checked. Conditions are checked when a unit is requested to start.

ConditionResult contains the condition result of the last time the configured conditions of this unit were checked.

Conditions contains all configured conditions of the unit. For each condition five fields are given: condition type (e.g. **ConditionPathExists**), whether the condition is a trigger condition, whether the condition is reversed, the right hand side of the condtion (e.g. the path in case of **ConditionPathExists**), and the status. The status can be 0, in which case the condition hasn't been checked yet, a positive value, in which case the condition passed, or a negative value, in which case the condition failed. Currently only 0, +1, and -1 are used, but additional values may be used in the future, retaining the meaning of zero/positive/negative values.

LoadError contains a pair of strings. If the unit failed to load (as encoded in **LoadState**, see above), then this will include a D-Bus error pair consisting of the error ID and an explanatory human readable string of what happened. If it succeeded to load this will be a pair of empty strings.

Transient contains a boolean that indicates whether the unit was created as transient unit (i.e. via `CreateTransientUnit()` on the manager object).

Service Unit Objects

All service unit objects implement the `org.freedesktop.systemd1.Service` interface (described here) in addition to the generic `org.freedesktop.systemd1.Unit` interface (see above).

```
$ gdbus introspect --system --dest org.freedesktop.systemd1 --object-path /org/fr
node /org/freedesktop/systemd1/unit/avahi_2ddaemon_2eservice {
  interface org.freedesktop.systemd1.Unit {
    ...
  };
  interface org.freedesktop.systemd1.Service {
    methods:
    signals:
    properties:
      readonly s Type = 'dbus';
      readonly s Restart = 'no';
      readonly s PIDFile = '';
      readonly s NotifyAccess = 'main';
      readonly t RestartUSec = 100000;
      readonly t TimeoutUSec = 90000000;
      readonly t WatchdogUSec = 0;
      readonly t WatchdogTimestamp = 0;
      readonly t WatchdogTimestampMonotonic = 0;
      readonly t StartLimitInterval = 10000000;
      readonly u StartLimitBurst = 5;
      readwrite s StartLimitAction = 'none
      readonly s Slice = 'system.slice';
      readonly s ControlGroup = '/system.slice/avahi-daemon.service';
      readonly a(sasbttuii) ExecStartPre = [];
      readonly a(sasbttuii) ExecStart = [('/usr/sbin/avahi-daemon', ['/usr/sbin/a
      readonly a(sasbttuii) ExecStartPost = [];
      readonly a(sasbttuii) ExecReload = [('/usr/sbin/avahi-daemon', ['/usr/sbin/
      readonly a(sasbttuii) ExecStop = [];
      readonly a(sasbttuii) ExecStopPost = [];
      readonly as Environment = [];
      readonly a(sb) EnvironmentFiles = [];
      readonly u UMask = 18;
      readonly t LimitCPU = 18446744073709551615;
      readonly t LimitFSIZE = 18446744073709551615;
      readonly t LimitDATA = 18446744073709551615;
      readonly t LimitSTACK = 18446744073709551615;
      readonly t LimitCORE = 18446744073709551615;
      readonly t LimitRSS = 18446744073709551615;
      readonly t LimitNOFILE = 4096;
      readonly t LimitAS = 18446744073709551615;
      readonly t LimitNPROC = 256963;
      readonly t LimitMEMLOCK = 65536;
      readonly t LimitLOCKS = 18446744073709551615;
      readonly t LimitSIGPENDING = 256963;
      readonly t LimitMSGQUEUE = 819200;
      readonly t LimitNICE = 0;
      readonly t LimitRTPRIO = 0;
      readonly t LimitRTTIME = 18446744073709551615;
      readonly s WorkingDirectory = '';
      readonly s RootDirectory = '';
      readonly i OOMScoreAdjust = 0;
      readonly i Nice = 0;
      readonly i IOScheduling = 0;
      readonly i CPUSchedulingPolicy = 0;
      readonly i CPUSchedulingPriority = 0;
      readonly ay CPUAffinity = [];
      readonly t TimerSlackNSec = 50000;
      readonly b CPUSchedulingResetOnFork = false;
      readonly b NonBlocking = false;
      readonly s StandardInput = 'null';
      readonly s StandardOutput = 'journal';
      readonly s StandardError = 'inherit';
      readonly s TTYPath = '';
      readonly b TTYReset = false;
      readonly b TTYVHangup = false;
      readonly b TTYVTDisallocate = false;
      readonly i SyslogPriority = 30;
      readonly s SyslogIdentifier = '';
      readonly b SyslogLevelPrefix = true;
      readonly s Capabilities = '';
      readonly i SecureBits = 0;
      readonly t CapabilityBoundingSet = 18446744073709551615;
      readonly s User = '';
      readonly s Group = '';
      readonly as SupplementaryGroups = [];
      readonly s TCPWrapName = '';
      readonly s PAMName = '';
      readonly as ReadWriteDirectories = [];
      readonly as ReadOnlyDirectories = [];
      readonly as InaccessibleDirectories = [];
      readonly t MountFlags = 0;
      readonly b PrivateTmp = false;
      readonly b PrivateNetwork = false;
      readonly b SameProcessGroup = false;
      readonly s UtmpIdentifier = '';
      readonly b IgnoreSIGPIPE = true;
      readonly b NoNewPrivileges = false;
      readonly au SystemCallFilter = [];
      readonly s KillMode = 'control-group';
      readonly i KillSignal = 15;
      readonly b SendSIGKILL = true;
      readonly b SendSIGHUP = false;
      readonly b CPUAccounting = false;
      readonly t CPUShares = 1024;
      readonly b BlockIOAccounting = false;
      readonly t BlockIOWeight = 1000;
      readonly a(st) BlockIODeviceWeight = [];
      readonly a(st) BlockIOReadBandwidth=;
      readonly a(st) BlockIOWriteBandwidth=;
      readonly b MemoryAccounting = false;
      readonly t MemoryLimit = 18446744073709551615;
      readonly s DevicePolicy = 'auto';
      readonly a(ss) DeviceAllow = [];
      readonly b PermissionsStartOnly = false;
      readonly b RootDirectoryStartOnly = false;
      readonly b RemainAfterExit = false;
      readonly t ExecMainStartTimestamp = 1368576738559537;
      readonly t ExecMainStartTimestampMonotonic = 22561233;
      readonly t ExecMainExitTimestamp = 1368576738559537;
      readonly t ExecMainExitTimestampMonotonic = 22561233;
      readonly u ExecMainPID = 666;
      readonly i ExecMainCode = 0;
      readonly i ExecMainStatus = 0;
      readonly u MainPID = 666;
      readonly u ControlPID = 0;
      readonly s BusName = 'org.freedesktop.Avahi';
      readonly s StatusText = 'Server startup complete. Host name is fedora.local
      readonly s Result = 'success';
  };
  interface org.freedesktop.DBus.Properties {
    ...
  };
  interface org.freedesktop.DBus.Peer {
    ...
  };
  interface org.freedesktop.DBus.Introspectable {
    ...
  };
};
```

Properties

Most properties of the Service interface map directly to the corresponding settings in service unit files. For the sake of brevity, here's a list of all exceptions only:

WatchdogTimestamp and **WatchdogTimestampMonotonic** contain CLOCK_REALTIME/CLOCK_MONOTONIC usec timestamps of the last watchdog ping received from the service, or 0 if none was ever received.

ExecStartPre, **ExecStart**, **ExecStartPost**, **ExecReload**, **ExecStop**, **ExecStop** each are arrays of structures each containing: the binary path to execute; an array with all arguments to pass to the executed command, starting with argument 0; a boolean whether it should be considered a failure if the process exits uncleanly; two pairs of CLOCK_REALTIME/CLOCK_MONOTONIC usec timestamps when the process began and finished running the last time, or 0 if it never ran or never finished running; the PID of the process, or 0 if it has not run yet; the exit code and status of the last run. This field hence maps more or less to the corresponding setting in the service unit file but is augmented with runtime data.

LimitCPU (and related properties) map more or less directly to the corresponding settings in the service unit files, however are set to 18446744073709551615 (i.e. -1) if they aren't set.

Capabilities contains the configured capabilities, as formatted with cap_to_text().

SecureBits, **CapabilityBoundingSet**, **MountFlags** also correspond to the configured settings of the unit files, but are encoded as the actual binary flag fields they are, rather than formatted as string.

ExecMainStartTimestamp, **ExecMainStartTimestampMonotonic**, **ExecMainExitTimestamp**, **ExecMainExitTimestampMonotonic**, **ExecMainPID**, **ExecMainCode**, **ExecMainStatus** contain information about the main process of the service as far as it is known. This is often the same runtime information that is stored in **ExecStart**. However, it deviates for Type=forking services where the main process of the service is not forked off systemd directly. These fields either contain information of the last run of the process or of the current running process.

MainPID and **ControlPID** contain the main and control PID of the service. The main PID is the current main PID of the service and is 0 when the service currently has no main PID. The control PID is the PID of the current start/stop/reload process running and is 0 if no such process is currently running. That means that **ExecMainPID** and **MainPID** differ in the way that the latter immediately reflects whether a main process is currently running while the latter possible contains information collected from the last run even if the process is no longer around.

StatusText contains the status text passed to the service manager via a call to sd_notify(). This may be used by services to inform the service manager about its internal state with a nice explanatory string.

Result encodes the execution result of the last run of the service. It is useful to determine the reason a service failed if it is in *failed* state (see **ActiveState** above). The following values are currently known: *success* is set if the unit didn't fail. *resources* indicates that not enough resources have been available to fork off and execute the service processes. *timeout* indicates that a time-out occurred while executing a service operation. *exit-code* indicates that a service process exited with an unclean exit code. *signal* indicates that a service process exited with an uncaught signal. *core-dump* indicates that a service process exited uncleanly and dumped core. *watchdog* indicates that a service did not send out watchdog ping messages often enough. *start-limit* indicates that a service has been started too frequently in a time frame (as configured in **StartLimitInterval**, **StartLimitBurst**).

ControlGroup indicates the control group path the processes of this service unit are placed in.

Socket Unit Objects

All socket unit objects implement the `org.freedesktop.systemd1.Socket` interface (described here) in addition to the generic `org.freedesktop.systemd1.Unit` interface (see above).

```
$ gdbus introspect --system --dest org.freedesktop.systemd1 --object-path /org/fr
node /org/freedesktop/systemd1/unit/avahi_2ddaemon_2esocket {
  interface org.freedesktop.systemd1.Unit {
    ...
  };
  interface org.freedesktop.systemd1.Socket {
    methods:
    signals:
    properties:
      readonly b BindIPv6Only = 'default';
      readonly u Backlog = 128;
      readonly t TimeoutUSec = 90000000;
      readonly s Slice = 'system.slice';
      readonly s ControlGroup = '/system.slice/avahi-daemon.socket';
      readonly a(sasbttuii) ExecStartPre = [];
      readonly a(sasbttuii) ExecStartPost = [];
      readonly a(sasbttuii) ExecStopPre = [];
      readonly a(sasbttuii) ExecStopPost = [];
      readonly as Environment = [];
      readonly a(sb) EnvironmentFiles = [];
      readonly u UMask = 18;
      readonly t LimitCPU = 18446744073709551615;
      readonly t LimitFSIZE = 18446744073709551615;
      readonly t LimitDATA = 18446744073709551615;
      readonly t LimitSTACK = 18446744073709551615;
      readonly t LimitCORE = 18446744073709551615;
      readonly t LimitRSS = 18446744073709551615;
      readonly t LimitNOFILE = 4096;
```

```
        readonly t LimitAS = 18446744073709551615;
        readonly t LimitNPROC = 61434;
        readonly t LimitMEMLOCK = 65536;
        readonly t LimitLOCKS = 18446744073709551615;
        readonly t LimitSIGPENDING = 61434;
        readonly t LimitMSGQUEUE = 819200;
        readonly t LimitNICE = 0;
        readonly t LimitRTPRIO = 0;
        readonly t LimitRTTIME = 18446744073709551615;
        readonly s WorkingDirectory = '';
        readonly s RootDirectory = '';
        readonly i OOMScoreAdjust = 0;
        readonly i Nice = 0;
        readonly i IOScheduling = 0;
        readonly i CPUSchedulingPolicy = 0;
        readonly i CPUSchedulingPriority = 0;
        readonly ay CPUAffinity = [];
        readonly t TimerSlackNSec = 50000;
        readonly b CPUSchedulingResetOnFork = false;
        readonly b NonBlocking = false;
        readonly s StandardInput = 'null';
        readonly s StandardOutput = 'journal';
        readonly s StandardError = 'inherit';
        readonly s TTYPath = '';
        readonly b TTYReset = false;
        readonly b TTYVHangup = false;
        readonly b TTYVTDisallocate = false;
        readonly i SyslogPriority = 30;
        readonly s SyslogIdentifier = '';
        readonly b SyslogLevelPrefix = true;
        readonly s Capabilities = '';
        readonly i SecureBits = 0;
        readonly t CapabilityBoundingSet = 18446744073709551615;
        readonly s User = '';
        readonly s Group = '';
        readonly as SupplementaryGroups = [];
        readonly s TCPWrapName = '';
        readonly s PAMName = '';
        readonly as ReadWriteDirectories = [];
        readonly as ReadOnlyDirectories = [];
        readonly as InaccessibleDirectories = [];
        readonly t MountFlags = 0;
        readonly b PrivateTmp = false;
        readonly b PrivateNetwork = false;
        readonly b SameProcessGroup = false;
        readonly s UtmpIdentifier = '';
        readonly b IgnoreSIGPIPE = true;
        readonly b NoNewPrivileges = false;
        readonly au SystemCallFilter = [];
        readonly s KillMode = 'control-group';
        readonly i KillSignal = 15;
        readonly b SendSIGKILL = true;
        readonly b SendSIGHUP = false;
        readonly b CPUAccounting = false;
        readonly t CPUShares = 1024;
        readonly b BlockIOAccounting = false;
        readonly t BlockIOWeight = 1000;
        readonly a(st) BlockIODeviceWeight = [];
        readonly a(st) BlockIOReadBandwidth=;
        readonly a(st) BlockIOWriteBandwidth=;
        readonly b MemoryAccounting = false;
        readonly t MemoryLimit = 18446744073709551615;
        readonly s DevicePolicy = 'auto';
        readonly a(ss) DeviceAllow = [];
        readonly u ControlPID = 0;
```

```
      readonly s BindToDevice = '';
      readonly u DirectoryMode = 493;
      readonly u SocketMode = 438;
      readonly b Accept = false;
      readonly b KeepAlive = false;
      readonly i Priority = -1;
      readonly t ReceiveBuffer = 0;
      readonly t SendBuffer = 0;
      readonly i IPTOS = -1;
      readonly i IPTTL = -1;
      readonly t PipeSize = 0;
      readonly b FreeBind = false;
      readonly b Transparent = false;
      readonly b Broadcast = false;
      readonly b PassCredentials = false;
      readonly b PassSecurity = false;
      readonly i Mark = -1;
      readonly u MaxConnections = 64;
      readonly u NAccepted = 0;
      readonly u NConnections = 0;
      readonly x MessageQueueMaxMessages = 0;
      readonly x MessageQueueMessageSize = 0;
      readonly a(ss) Listen = [('Stream', '/var/run/avahi-daemon/socket')];
      readonly s Result = 'success';
      readonly b ReusePort = false;
      readonly s SmackLabel = '';
      readonly s SmackLabelIPIn = '';
      readonly s SmackLabelIPOut = '';
  };
  interface org.freedesktop.DBus.Properties {
    ...
  };
  interface org.freedesktop.DBus.Peer {
    ...
  };
  interface org.freedesktop.DBus.Introspectable {
    ...
  };
};
```

Properties

Most of the properties map directly to the corresponding settings in socket unit files. As socket units can include **ExecStartPre** (and similar) fields which contain information about processes to execute. They also share most of the fields related to the execution context that Service objects expose (see above). In addition to these properties there are the following:

NAccepted contains the accumulated number of connections ever accepted on this socket. This only applies to sockets with **Accept** set to *true*, i.e. those where systemd is responsible for accepted connections.

Similarly **NConnections** contains the number of currently open connections on this socket, and also applies only to socket with **Accept** set to *true*.

Result encodes the reason why a socket unit failed if it is in *failed* state (see **ActiveState** above). The values *success*, *resources*, *timeout*, *exit-code*, *signal* and *core-dump* have the same meaning as they have for the corresponding field of service units (see above). In addition to that the value **service-failed-permanent** indicates that the service of this socket failed continuously.

Target Unit Objects

All target unit objects implement the `org.freedesktop.systemd1.Target` interface (described here) in addition to the generic `org.freedesktop.systemd1.Unit` interface (see above).

```
$ gdbus introspect --system --dest org.freedesktop.systemd1 --object-path /org/fr
node /org/freedesktop/systemd1/unit/basic_2etarget {
  interface org.freedesktop.systemd1.Unit {
    ...
  };
  interface org.freedesktop.systemd1.Target {
    methods:
    signals:
    properties:
  };
  interface org.freedesktop.DBus.Properties {
    ...
  };
  interface org.freedesktop.DBus.Peer {
    ...
  };
  interface org.freedesktop.DBus.Introspectable {
    ...
  };
};
```

Target units have neither type-specific methods nor properties.

Device Unit Objects

All device unit objects implement the `org.freedesktop.systemd1.Device` interface (described here) in addition to the generic `org.freedesktop.systemd1.Unit` interface (see above).

```
$ gdbus introspect --system --dest org.freedesktop.systemd1 --object-path /org/fr
node /org/freedesktop/systemd1/unit/dev_2ddisk_2dby_5cx2did_2data_5cx2dSAMSUNG_5f
  interface org.freedesktop.systemd1.Unit {
    ...
  };
  interface org.freedesktop.systemd1.Device {
    methods:
    signals:
    properties:
      readonly s SysFSPath = '/sys/devices/pci0000:00/0000:00:1f.2/ata2/host1/tar
  };
  interface org.freedesktop.DBus.Properties {
    ...
  };
  interface org.freedesktop.DBus.Peer {
    ...
  };
  interface org.freedesktop.DBus.Introspectable {
    ...
  };
};
```

Properties

Device units only expose a single type-specific property:

SysFSPath contains the sysfs path of the kernel device this object corresponds to.

Mount Unit Objects

All mount unit objects implement the `org.freedesktop.systemd1.Mount` interface (described here) in addition to the generic `org.freedesktop.systemd1.Unit` interface (see above).

```
$ gdbus introspect --system --dest org.freedesktop.systemd1 --object-path /org/fr
node /org/freedesktop/systemd1/unit/home_2emount {
  interface org.freedesktop.systemd1.Unit {
    ...
  };
  interface org.freedesktop.systemd1.Mount {
    methods:
    signals:
    properties:
      readonly s Where = '/home';
      readonly s What = '/dev/mapper/home';
      readonly s Options = 'rw,relatime,rw,seclabel,data=ordered';
      readonly s Type = 'ext4';
      readonly t TimeoutUSec = 90000000;
      readonly s Slice = 'system.slice';
      readonly s ControlGroup = '/system.slice/home.mount';
      readonly a(sasbttuii) ExecMount = [('/bin/mount', ['/bin/mount', '/dev/disk
      readonly a(sasbttuii) ExecUnmount = [];
      readonly a(sasbttuii) ExecRemount = [];
      readonly as Environment = [];
      readonly a(sb) EnvironmentFiles = [];
      readonly u UMask = 18;
      readonly t LimitCPU = 18446744073709551615;
      readonly t LimitFSIZE = 18446744073709551615;
      readonly t LimitDATA = 18446744073709551615;
      readonly t LimitSTACK = 18446744073709551615;
      readonly t LimitCORE = 18446744073709551615;
      readonly t LimitRSS = 18446744073709551615;
      readonly t LimitNOFILE = 4096;
      readonly t LimitAS = 18446744073709551615;
      readonly t LimitNPROC = 61434;
      readonly t LimitMEMLOCK = 65536;
      readonly t LimitLOCKS = 18446744073709551615;
      readonly t LimitSIGPENDING = 61434;
      readonly t LimitMSGQUEUE = 819200;
      readonly t LimitNICE = 0;
      readonly t LimitRTPRIO = 0;
      readonly t LimitRTTIME = 18446744073709551615;
      readonly s WorkingDirectory = '';
      readonly s RootDirectory = '';
      readonly i OOMScoreAdjust = 0;
      readonly i Nice = 0;
      readonly i IOScheduling = 0;
      readonly i CPUSchedulingPolicy = 0;
      readonly i CPUSchedulingPriority = 0;
      readonly ay CPUAffinity = [];
      readonly t TimerSlackNSec = 50000;
      readonly b CPUSchedulingResetOnFork = false;
      readonly b NonBlocking = false;
      readonly s StandardInput = 'null';
      readonly s StandardOutput = 'journal';
      readonly s StandardError = 'inherit';
      readonly s TTYPath = '';
      readonly b TTYReset = false;
      readonly b TTYVHangup = false;
      readonly b TTYVTDisallocate = false;
      readonly i SyslogPriority = 30;
      readonly s SyslogIdentifier = '';
      readonly b SyslogLevelPrefix = true;
      readonly s Capabilities = '';
      readonly i SecureBits = 0;
      readonly t CapabilityBoundingSet = 18446744073709551615;
      readonly s User = '';
      readonly s Group = '';
      readonly as SupplementaryGroups = [];
      readonly s TCPWrapName = '';
      readonly s PAMName = '';
      readonly as ReadWriteDirectories = [];
      readonly as ReadOnlyDirectories = [];
      readonly as InaccessibleDirectories = [];
      readonly t MountFlags = 0;
      readonly b PrivateTmp = false;
      readonly b PrivateNetwork = false;
      readonly b SameProcessGroup = true;
      readonly s UtmpIdentifier = '';
      readonly b IgnoreSIGPIPE = true;
      readonly b NoNewPrivileges = false;
      readonly au SystemCallFilter = [];
      readonly s KillMode = 'control-group';
      readonly i KillSignal = 15;
      readonly b SendSIGKILL = true;
      readonly b SendSIGHUP = false;
      readonly b CPUAccounting = false;
      readonly t CPUShares = 1024;
      readonly b BlockIOAccounting = false;
      readonly t BlockIOWeight = 1000;
      readonly a(st) BlockIODeviceWeight = [];
      readonly a(st) BlockIOReadBandwidth=;
      readonly a(st) BlockIOWriteBandwidth=;
      readonly b MemoryAccounting = false;
      readonly t MemoryLimit = 18446744073709551615;
      readonly s DevicePolicy = 'auto';
      readonly a(ss) DeviceAllow = [];
      readonly u ControlPID = 0;
      readonly u DirectoryMode = 493;
      readonly s Result = 'success';
  };
  interface org.freedesktop.DBus.Properties {
    ...
  };
  interface org.freedesktop.DBus.Peer {
    ...
  };
  interface org.freedesktop.DBus.Introspectable {
    ...
  };
};
```

Properties

Most of the properties map directly to the corresponding settings in mount unit files. As mount units invoke the `/usr/bin/mount` command their bus objects include implicit **ExecMount** (and similar) fields which contain information about processes to execute. They also share most of the fields related to the execution context that Service objects expose (see above). In addition to these properties there are the following:

ControlPID contains the PID of the currently running `/usr/bin/mount` or `/usr/bin/umount` command if there is one running, otherwise 0.

Result contains a value explaining why a mount unit failed if it failed. It can take the values *success*, *resources*, *timeout*, *exit-code*, *signal*, *core-dump* which have the identical meaning as the corresponding values of the corresponding field of service unit objects (see above).

Automount Unit Objects

All automount unit objects implement the `org.freedesktop.systemd1.Automount` interface (described here) in addition to the generic `org.freedesktop.systemd1.Unit` interface (see above).

```
$ gdbus introspect --system --dest org.freedesktop.systemd1 --object-path /org/fr
node /org/freedesktop/systemd1/unit/proc_2dsys_2dfs_2dbinfmt_5fmisc_2eautomount {
  interface org.freedesktop.systemd1.Unit {
    ...
  };
  interface org.freedesktop.systemd1.Automount {
    methods:
    signals:
    properties:
      readonly s Where = '/proc/sys/fs/binfmt_misc';
      readonly u DirectoryMode = 493;
      readonly s Result = 'success';
  };
  interface org.freedesktop.DBus.Properties {
    ...
  };
  interface org.freedesktop.DBus.Peer {
    ...
  };
  interface org.freedesktop.DBus.Introspectable {
    ...
  };
};
```

Properties

Most of the properties map directly to the corresponding settings in the automount unit files.

Result knows the values *success* and *resources*, at this time. They have the same meanings as the corresponding values of the corresponding field of the Service object.

Snapshot Unit Objects

All snapshot unit objects implement the `org.freedesktop.systemd1.Snapshot` interface (described here) in addition to the generic `org.freedesktop.systemd1.Unit` interface (see above).

```
$ gdbus introspect --system --dest org.freedesktop.systemd1 --object-path /org/fr
node /org/freedesktop/systemd1/unit/foo_2esnapshot {
  interface org.freedesktop.systemd1.Unit {
    ...
  };
  interface org.freedesktop.systemd1.Snapshot {
    methods:
      Remove();
    signals:
    properties:
      readonly b Cleanup = false;
  };
  interface org.freedesktop.DBus.Properties {
    ...
  };
  interface org.freedesktop.DBus.Peer {
    ...
  };
  interface org.freedesktop.DBus.Introspectable {
    ...
  };
};
```

Methods

Remove() deletes the snapshot unit. This operation is also available in the ?**RemoveSnapshot()** operation of the Manager object (see above), which is sometimes nicer to call, in order to reduce roundtrips.

Properties

Cleanup is a boolean that indicates that the snapshot unit should be removed automatically after the first time it is activated.

Timer Unit Objects

All timer unit objects implement the `org.freedesktop.systemd1.Timer` interface (described here) in addition to the generic `org.freedesktop.systemd1.Unit` interface (see above).

```
$ gdbus introspect --system --dest org.freedesktop.systemd1 --object-path /org/fr
node /org/freedesktop/systemd1/unit/systemd_2dtmpfiles_2dclean_2etimer {
  interface org.freedesktop.systemd1.Unit {
    ...
  };
  interface org.freedesktop.systemd1.Timer {
      readonly s Unit = 'systemd-tmpfiles-clean.service';
      readonly a(stt) TimersMonotonic = [('OnUnitActiveUSec', 86400000000, 173700
      readonly a(sst) TimersCalendar = [];
      readonly t NextElapseUSecRealtime = 0;
      readonly t NextElapseUSecMonotonic = 173700033104;
      readonly s Result = 'success';
  };
  interface org.freedesktop.DBus.Properties {
    ...
  };
  interface org.freedesktop.DBus.Peer {
    ...
  };
  interface org.freedesktop.DBus.Introspectable {
    ...
  };
};
```

Properties

Unit contains the name of the unit to activate when the timer elapses.

TimersMonotonic contains an array of structs that contain information about all monotonic timers of this timer unit. The structs contain a string identifying the timer base, which is one of *OnActiveUSec*, *OnBootUSec*, *OnStartupUSec*, *OnUnitActiveUSec*, *OnUnitInactiveUSec*, which correspond to the settings of the same names in the timer unit files; the usec offset from this timer base in monotonic time; the next elapsation point on the CLOCK_MONOTONIC clock, relative its epoch.

TimersCalendar contains an array of structs that contain information about all realtime/calendar timers of this timer unit. The structs contain a string identifying the timer base, which may only be *OnCalendar* for now; the calendar specification string; the next elapsation point on the CLOCK_REALTIME clock, relative to its epoch.

NextElapseUSecRealtime contains the next elapsation point on the CLOCK_REALTIME clock in usec since the epoch, or 0 if this timer event does not include at least one calendar event.

Similar, **NextElapseUSecMonotonic** contains the next elapsation point on the CLOCK_MONOTONIC clock in usec since the epoch, or 0 if this timer event does not include at least one monotonic event.

Result knows the values *success* and *resources* with the same meanings as the matching values of the corresponding property of the service interface.

Swap Unit Objects

All swap unit objects implement the `org.freedesktop.systemd1.Swap` interface (described here) in addition to the generic `org.freedesktop.systemd1.Unit` interface (see above).

```
$ gdbus introspect --system --dest org.freedesktop.systemd1 --object-path /org/fr
node /org/freedesktop/systemd1/unit/dev_2dsda3_2eswap {
  interface org.freedesktop.systemd1.Unit {
    ...
  };
  interface org.freedesktop.systemd1.Swap {
    methods:
    signals:
    properties:
      readonly s What = '/dev/sda3';
      readonly i Priority = -1;
      readonly t TimeoutUSec = 90000000;
      readonly s Slice = 'system.slice';
      readonly s ControlGroup = '';
      readonly a(sasbttuii) ExecActivate = [];
      readonly a(sasbttuii) ExecDeactivate = [];
      readonly as Environment = [];
      readonly a(sb) EnvironmentFiles = [];
      readonly u UMask = 18;
      readonly t LimitCPU = 18446744073709551615;
      readonly t LimitFSIZE = 18446744073709551615;
      readonly t LimitDATA = 18446744073709551615;
      readonly t LimitSTACK = 18446744073709551615;
      readonly t LimitCORE = 18446744073709551615;
      readonly t LimitRSS = 18446744073709551615;
      readonly t LimitNOFILE = 4096;
      readonly t LimitAS = 18446744073709551615;
      readonly t LimitNPROC = 61434;
      readonly t LimitMEMLOCK = 65536;
      readonly t LimitLOCKS = 18446744073709551615;
      readonly t LimitSIGPENDING = 61434;
      readonly t LimitMSGQUEUE = 819200;
      readonly t LimitNICE = 0;
      readonly t LimitRTPRIO = 0;
      readonly t LimitRTTIME = 18446744073709551615;
      readonly s WorkingDirectory = '';
      readonly s RootDirectory = '';
      readonly i OOMScoreAdjust = 0;
      readonly i Nice = 0;
      readonly i IOScheduling = 0;
      readonly i CPUSchedulingPolicy = 0;
      readonly i CPUSchedulingPriority = 0;
      readonly ay CPUAffinity = [];
      readonly t TimerSlackNS = 50000;
      readonly b CPUSchedulingResetOnFork = false;
      readonly b NonBlocking = false;
      readonly s StandardInput = 'null';
      readonly s StandardOutput = 'journal';
      readonly s StandardError = 'inherit';
      readonly s TTYPath = '';
      readonly b TTYReset = false;
      readonly b TTYVHangup = false;
      readonly b TTYVTDisallocate = false;
      readonly i SyslogPriority = 30;
      readonly s SyslogIdentifier = '';
      readonly b SyslogLevelPrefix = true;
      readonly s Capabilities = '';
      readonly i SecureBits = 0;
      readonly t CapabilityBoundingSet = 18446744073709551615;
      readonly s User = '';
      readonly s Group = '';
      readonly as SupplementaryGroups = [];
      readonly s TCPWrapName = '';
      readonly s PAMName = '';
      readonly as ReadWriteDirectories = [];
      readonly as ReadOnlyDirectories = [];
      readonly as InaccessibleDirectories = [];
      readonly t MountFlags = 0;
      readonly b PrivateTmp = false;
      readonly b PrivateNetwork = false;
      readonly b SameProcessGroup = false;
      readonly s KillMode = 'control-group';
      readonly i KillSignal = 15;
      readonly s UtmpIdentifier = '';
      readonly b IgnoreSIGPIPE = true;
      readonly b NoNewPrivileges = false;
      readonly au SystemCallFilter = [];
      readonly s KillMode = 'control-group';
      readonly i KillSignal = 15;
      readonly b SendSIGKILL = true;
      readonly b SendSIGHUP = false;
      readonly b CPUAccounting = false;
      readonly t CPUShares = 1024;
      readonly b BlockIOAccounting = false;
      readonly t BlockIOWeight = 1000;
      readonly a(st) BlockIODeviceWeight = [];
      readonly a(st) BlockIOReadBandwidth=;
      readonly a(st) BlockIOWriteBandwidth=;
      readonly b MemoryAccounting = false;
      readonly t MemoryLimit = 18446744073709551615;
      readonly s DevicePolicy = 'auto';
      readonly a(ss) DeviceAllow = [];
      readonly u ControlPID = 0;
      readonly s Result = 'success';
  };
  interface org.freedesktop.DBus.Properties {
    ...
  };
  interface org.freedesktop.DBus.Peer {
    ...
  };
  interface org.freedesktop.DBus.Introspectable {
    ...
  };
};
```

Properties

Most of the properties map directly to the corresponding settings in swap unit files. As mount units invoke the `/usr/bin/swapon` command their bus objects include implicit **ExecActivate** (and similar) fields which contain information about processes to execute. They also share most of the fields related to the execution context that Service objects expose (see above). In addition to these properties there are the following:

ControlPID contains the PID of the currently running `/usr/bin/swapon` or `/usr/bin/swapoff` command if there is one running, otherwise 0.

Result contains a value explaining why a mount unit failed if it failed. It can take the values *success*, *resources*, *timeout*, *exit-code*, *signal*, *core-dump* which have the identical meanings as the corresponding values of the corresponding field of service unit objects (see above).

Path Unit Objects

All path unit objects implement the `org.freedesktop.systemd1.Path` interface (described here) in addition to the generic `org.freedesktop.systemd1.Unit` interface (see above).

```
$ gdbus introspect --system --dest org.freedesktop.systemd1 --object-path /org/fr
node /org/freedesktop/systemd1/unit/cups_2epath {
  interface org.freedesktop.systemd1.Unit {
    ...
  };
  interface org.freedesktop.systemd1.Path {
    methods:
    signals:
    properties:
      readonly s Unit = 'cups.service';
      readonly a(ss) Paths = [('PathExistsGlob', '/var/spool/cups/d*')];
      readonly b MakeDirectory = false;
      readonly u DirectoryMode = 493;
      readonly s Result = 'success';
  };
  interface org.freedesktop.DBus.Properties {
    ...
  };
  interface org.freedesktop.DBus.Peer {
    ...
  };
  interface org.freedesktop.DBus.Introspectable {
    ...
  };
};
```

Properties

Most properties correspond directly with the matching settings in path unit files. The others:

Paths contains an array of structs. Each struct contains the condition to watch, which can be one of *PathExists*, *PathExistsGlob*, *PathChanged*, *PathModified*, *DirectoryNotEmpty* which correspond directly to the matching settings in the path unit files; and the path to watch, possibly including glob expressions.

Result contains a result value which can be *success* or *resources*, which have the same meaning as the corresponding field of the service interface.

Slice Unit Objects

All slice unit objects implement the `org.freedesktop.systemd1.Slice` interface (described here) in addition to the generic `org.freedesktop.systemd1.Unit` interface (see above).

```
$ gdbus introspect --system --dest org.freedesktop.systemd1 --object-path /org/fr
node /org/freedesktop/systemd1/unit/system_2eslice {
  interface org.freedesktop.systemd1.Unit {
    ...
  };
  interface org.freedesktop.systemd1.Slice {
    methods:
    signals:
    properties:
      readonly s Slice = '-.slice';
      readonly s ControlGroup = '/system.slice';
      readonly b CPUAccounting = false;
      readonly t CPUShares = 1024;
      readonly b BlockIOAccounting = false;
      readonly t BlockIOWeight = 1000;
      readonly a(st) BlockIODeviceWeight = [];
      readonly a(st) BlockIOReadBandwidth=;
      readonly a(st) BlockIOWriteBandwidth=;
      readonly b MemoryAccounting = false;
      readonly t MemoryLimit = 18446744073709551615;
      readonly s DevicePolicy = 'auto';
      readonly a(ss) DeviceAllow = [];
  };
  interface org.freedesktop.DBus.Properties {
    ...
  };
  interface org.freedesktop.DBus.Peer {
    ...
  };
  interface org.freedesktop.DBus.Introspectable {
    ...
  };
};
```

Properties

Most properties correspond directly with the matching settings in slice unit files.

Scope Unit Objects

All slice unit objects implement the `org.freedesktop.systemd1.Sscope` interface (described here) in addition to the generic `org.freedesktop.systemd1.Unit` interface (see above).

```
$ gdbus introspect --system --dest org.freedesktop.systemd1 --object-path /org/fr
node /org/freedesktop/systemd1/unit/session_2d1_2escope {
  interface org.freedesktop.systemd1.Unit {
    ...
  };
  interface org.freedesktop.systemd1.Scope {
    methods:
      Abandon();
    signals:
      RequestStop();
    properties:
      readonly s Slice = 'user-1000.slice';
      readonly s ControlGroup = '/user.slice/user-1000.slice/session-1.scope';
      readonly t TimeoutStopUSec = 500000;
      readonly s KillMode = 'none';
      readonly i KillSignal = 15;
      readonly b SendSIGKILL = true;
      readonly b SendSIGHUP = true;
      readonly b CPUAccounting = false;
      readonly t CPUShares = 1024;
      readonly b BlockIOAccounting = false;
      readonly t BlockIOWeight = 1000;
      readonly a(st) BlockIODeviceWeight = [];
      readonly a(st) BlockIOReadBandwidth=;
      readonly a(st) BlockIOWriteBandwidth=;
      readonly b MemoryAccounting = false;
      readonly t MemoryLimit = 18446744073709551615;
      readonly s DevicePolicy = 'auto';
      readonly a(ss) DeviceAllow = [];
      readonly s Result = 'success';
      readonly s Controller = '';


  };
  interface org.freedesktop.DBus.Properties {
    ...
  };
  interface org.freedesktop.DBus.Peer {
    ...
  };
  interface org.freedesktop.DBus.Introspectable {
    ...
  };
};
```

Properties

All properties correspond directly with the matching properties of service units.

Controller contains the bus name (unique or well-known) that is notified when the scope unit is to be shut down via a **RequestStop** signal (see below). This is set when the scope is created. If not set the scope's processes will terminated with SIGTERM directly.

Methods

Abandon() may be used to place a scope unit in the "abandoned" state. This may be used to inform the system manager that the manager that created the scope lost interest in the scope (for example, because it is terminating), without wanting to shut down the scope entirely.

Signals

RequestStop is sent to the peer that is configured in the **Controller** property when systemd is requested to terminate the scope unit. A program registering a scope can use this to cleanly shut down the processes it added to the scope, instead of letting systemd do it with the usual SIGTERM logic.

Job Objects

Job objects encapsulate scheduled or running jobs. Each unit can have none or one jobs in the execution queue. Each job is attached to exactly one unit.

```
$ gdbus introspect --system --dest org.freedesktop.systemd1 --object-path /org/fr
node /org/freedesktop/systemd1/job/1292 {
  interface org.freedesktop.systemd1.Job {
    methods:
      Cancel();
    signals:
    properties:
      readonly u Id = 1292;
      readonly (so) Unit = ('slow.service', '/org/freedesktop/systemd1/unit/slow_
      readonly s JobType = 'start';
      readonly s State = 'running';
  };
  interface org.freedesktop.DBus.Properties {
    ...
  };
  interface org.freedesktop.DBus.Peer {
    ...
  };
  interface org.freedesktop.DBus.Introspectable {
    ...
  };
};
```

Methods

Cancel() cancels the job. Note that this will remove a job from the queue if it is not yet executed but generally will not cause a job that is already in the process of being executed to be aborted. This operation may also be requested via the ?**CancelJob()** method of the Manager object (see above), which is sometimes useful to reduce roundtrips.

Properties

Id is the numeric Id of the job. During the runtime of a systemd instance each numeric ID is only assigned once.

Unit refers to the unit this job belongs two. It is a structure consisting of the name of the unit and a bus path to the unit's object.

JobType refers to the job's type and is one of *start*, *verify-active*, *stop*, *reload*, *restart*, *try-restart*, *reload-or-start*. Note that later versions might define additional values.

State refers to the job's state and is one of *waiting* and *running*. The former indicates that a job is currently queued but has not begun to execute yet, the latter indicates that a job is currently being executed.

These D-Bus interfaces follow the usual interface versioning guidelines.

Last edited Fri May 7 01:22:37 2021