Systemd by Example

Frank Krick

10/8/2018

Table of Contents

Whether you like systemd or not, systemd is probably here to stay given that is used as PID 1 in most unix distributions. Systemd was the subject of a lot of controversy in the unix world, but that is not something I want to make the subject of my writing here. Instead, I would say that whether you like systemd or not, it is useful to understand systemd and to learn how to do things the systemd way.

In the spirit of full disclosure, let me point out my own biases here. I started out with unix distributions using SysV init and later moved on to upstart based linux distribution. As I was always comfortable with SysV init based distribution, given that the whole approach is simple and easy to understand as most of the boot is managed by scripts, I was not necessarily taken by storm when I finally moved to a linux

distribution based on systemd. On the other hand, the deeper I dug in to systemd, the more I could see the potential applications which were hard to implement with other approaches, especially given the interfaces that systemd provides for use by all kinds of programming languages and scripts in a much more usable manner than calling shell scripts and parsing the output.

As I could not find a lot of introductory as well as practical information regarding systemd, I decided to write my own introductory and practical material based on my needs for several projects of mine as well as to cure my own curiosity. Because I allege that there is no good introductory material available, I will start this text with an overview of the relevant material that I used as reference and basis for this text in the hope that the relevant resources will be helpful not only to me but also to the reader.

The first and biggest source of information for systemd is the systemd wiki on freedesktop.org, the official systemd wiki [1]. The systemd wiki includes links to several articles and blog posts for systemd functionality as well as the github repository for the source code but does not include an in-depth introduction to systemd itself.

The systemd wiki offers links to articles about systemd in several publications which provide high level introductions of systemd including the underlying philosophy in several languages. These articles will be mentioned in the following sections if they are relevant for their contents will be referenced there. Therefore I will not mention the respective articles here. Additionally, the systemd wiki includes links to several blog posts and documentation about specific systemd features from an administrators, users or developer perspective, most prominently featured, the blog post series on Lennart Poettering's blog Pid Eins [2].

While these materials provide a good amount of resources for anyone interested in systemd I found them lacking a single document combining introductory material, architecture description, background information and actionable descriptions of systemd tools in a single source presented in a cohesive and easy to follow manner, which I hope to deliver in this document.

1. Introduction to Systemd

Next to the linux kernel itself, the most important piece of any linux operating system is the first process started, the process with the process id 1. This process is responsible for spawning all the processes required for the operation of the operating system. Depending on the scope of the process running as process id 1, the process can also be responsible for more than just spawning other processes. Systemd in particular has a much bigger scope than that, is one of the main critiques regarding systemd as a bigger scope necessarily means a more complicated code base.

The core responsibilities of systemd are the following [3]:

- Starting the rest of the system
- Creating sockets that can in turn be used by started processes
- On-demand starting of processes based on socket activation
- Keeping track of processes using linux control groups
- Maintaining mount and automount points
- Implementing an elaborate transactional dependency-based service control logic
- Supporting of SysV and LSB init scripts for backwards compatibility

Additionally systemd provides daemons and tools for logging, basic system configuration, network configuration, network time synchronization, log forwarding and name resolution.

The descripion of systemd's responsibilities on the systemd wiki is a mouthful. I found it hard to wrap my head around all of systemd's features based on the existing documentation and high level descriptions in short articles. It is for that reason that I decided to write this example based introduction. In the following

[1] https://www.freedesktop.org/wiki/Software/systemd/
[2] http://0pointer.net/blog/
[3] https://www.freedesktop.org/wiki/Software/systemd/

sections I will introduce systemd's features with increasingly more complicated examples each highlighting another of systemd's features starting with the first example demonstrating how to enable manual starting and stopping of a service using systemd.

2. Manually Starting a Simple Application

Note: The files for this example can be found in the GitLab repository for the book [4] in the directory **examples/example-simple-service-manually**.

First I will define a simple service that systemd should start. The service will be a simple python script that does nothing else but sleep in an endless loop to simulate a service running in the background. The python script is implemented with the following code and stored on the filesystem as **/usr/local/bin/sample_service_1.py**:

```
import time

while True:
    time.sleep(60)
```

When we start that simple script, it will do nothing but sleep for 60 seconds repeatedly, but can be used as a placeholder for a service that would do something useful like listening to a socket. If I would start that script on the console, it would block the console and do nothing until I would cancel the script using Ctrl + C on the keyboard.

To start this service using systemd, we will have to create a systemd unit and start this unit with the systemd command line interface. But before we continue with the example, I will first explain what a systemd unit is and which of the available unit types we will use for out example.

Units are how systemd organizes the tasks it has to execute. Every task is represented by a unit of the respective type described by a configuration for the unit called the unit file. As mentioned, every unit has a type. Systemd identifies the type of a unit based on the suffid of the unit file. The unit types supported by systemd are listed in the systemd unit man page [5].

The following unit types were supported by systemd at the time of writing:

- Service [6]

  A service unit describes a process controlled and supervised by systemd.

- Socket [7]

  A socket unit describes a socket for socket based activation. The socket can be a unix socket, a network socket or a filesystem FIFO.

- Device [8]

  A device unit describes a device as exposed by sysfs/udev. Device units are automatically created by systemd and allow the definition of dependencies of other units to devices.

- Mount [9]

  Mount unit describe mount points managed by systemd. In addition to mount units described in unit files, systemd creates mount units based on entries in fstab.

[4] https://gitlab.com/franks_reich/systemd-by-example
[5] https://www.freedesktop.org/software/systemd/man/systemd.unit.html
[6] https://www.freedesktop.org/software/systemd/man/systemd.service.html
[7] https://www.freedesktop.org/software/systemd/man/systemd.socket.html
[8] https://www.freedesktop.org/software/systemd/man/systemd.device.html
[9] https://www.freedesktop.org/software/systemd/man/systemd.mount.html

- Automount [10]

  Automount units describe mount points that can systemd mounts automatically when accessed. Automount units have to be accompanied by a corresponding mount unit.

- Swap [11]

  Swap units described swap devices or files.

- Target [12]

  Target units group units and describe synchronization points during startup. They can be used to define boot targets and dependencies to groups of units.

- Path [13]

  Path units are used to watch file system paths and activate services based on changes in that path.

- Timer [14]

  Timer units activate units when a timer elapses.

- Slice [15]

  Slice units are concepts for hierarchically managing resources of a group of processes.

- Scope [16]

  Scope units manage an externally created set of system processes.

Based on the short overview of the available unit types, the unit type used for starting and stopping the python service is the service unit. A service unit is described using a unit file with the extension **.service**. To actually create the unit file, I will have to explain where systemd looks for unit files.

Systemd generally looks in two different directories for unit files:

- **/lib/systemd/system**

  The directory **/lib/systemd/system** is used for units delivered by the linux distribution.

- **/etc/systemd/system**

  The directory **/etc/systemd/system** is used for user defined units. A unit file in this directory has precedence over a unit file of the same name in **/lib/system/systemd**. This can be used to customize the unit delivered by the distribution without having the customizations overwritten by updates to the unit files.

As the sample python service is not a service delivered by the distribution, I create the unit file as **/etc/systemd/system/python-sample-1.service**.

```
[Unit]
Description=The sample service

[Service]
Type=simple
ExecStart=/usr/bin/python3 /usr/local/bin/sample_service_1.py
```

[10] https://www.freedesktop.org/software/systemd/man/systemd.automount.html
[11] https://www.freedesktop.org/software/systemd/man/systemd.swap.html
[12] https://www.freedesktop.org/software/systemd/man/systemd.target.html
[13] https://www.freedesktop.org/software/systemd/man/systemd.path.html
[14] https://www.freedesktop.org/software/systemd/man/systemd.timer.html
[15] https://www.freedesktop.org/software/systemd/man/systemd.slice.html
[16] https://www.freedesktop.org/software/systemd/man/systemd.scope.html

The unit files describes the the service unit using a configuration file similar to windows INI files. In the example above, the unit file contains two sections, the *Unit* section and the *Service* section. The *Unit* section contains keywords used by all units. The example only uses the *Description* keyword which systemd uses to display a description of the service in its tools. The *Service* section contains keywords specific to a service unit. The *Type* describes how systemd starts the service and *ExecStart* describes the executable that is started by systemd.

I describe the service types in detail in the section about service units. For now we will use the simple type for which systemd expects the process started by ExecStart to be the main process. Otherwise systemd will not assume anything about the service. Specifically if the service is used as a dependency for another service systemd will not wait before starting any of the depending services. If more control is required the service unit has to be created using a different type.

Once the service file has been created, we can ask systemd to start the service with the systemctl command line tool and systemd will execute the executable we provided with the *ExecStart* keyword. The systemctl tool allows management of systemd units on the command line [17].

```
systemctl start python-sample-1
```

Systemd starts queues the task to start the service asynchronously and the call to systemctl does not return if the service could be started successfully. The status of the service can be checked by calling the status command of the systemctl tool.

```
systemctl status python-sample-1
```

This will output something similar to the following output taken from my machine.

```
 python-sample-1.service - The sample service
   Loaded: loaded (/etc/systemd/system/python-sample-1.service; static; vendor preset: enabled)
   Active: active (running) since Mon 2018-10-01 22:16:57 UTC; 29min ago
Main PID: 12020 (python3)
    Tasks: 1 (limit: 4915)
   CGroup: /system.slice/python-sample-1.service
           12020 /usr/bin/python3 /usr/local/bin/sample_service_1.py

Oct 01 22:16:57 halloween-test systemd[1]: Started The sample service.
```

The output of the status command has a great deal of information about the service. The first line it displays the unit's name and the description from the unit file. The second line displays the status of the unit file, in this particular example, the status is loaded and the location of the unit file is shown. The third line displays the status of the unit itself. I started the unit a few minutes ago so the status of the unit is active and the sub status is running. Additionally, systemd display the time and day when the unit was started. Systemd displays the process id of the main process in the fourth line of the status commands output. The fifth line displays the name of the control group systemd creates for each process it supervises followed by all processes in that particular control group on the following lines.

This information is followed by the last entries in the log systemd keeps for the unit.

Systemd allows stopping the processes using the stop command of systemctl.

```
systemctl stop python-sample-1
```

After executing the command calling the status command will result in an output similar to the following.

```
 python-sample-1.service - The sample service
   Loaded: loaded (/etc/systemd/system/python-sample-1.service; static; vendor preset: enabled)
   Active: inactive (dead)

Oct 01 22:16:57 halloween-test systemd[1]: Started The sample service.
```

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

```
Oct 01 23:08:05 halloween-test systemd[1]: Stopping The sample service...
Oct 01 23:08:05 halloween-test systemd[1]: Stopped The sample service.
```

The output now has the same information in the first and second line compared to the output before stopping the service. The third line is different though. The status is now inactive with the sub status dead indicating that the process is no longer running. Additionally, systemd displays additional log messages relating to the unit.

This example, even though simple, touched in several important topics in relation to systemd. I described that systemd is configured using units and which unit types systemd provides. The directories containing the systemd unit files were described and I demonstrated how a systemd can be configured to be able to start and stop a simple service, which command line tool can be used to accomplish that and how systemd displays the status of a simple service.

3. Automatically Starting a Simple Service as Part of a Target

Note: The files for this example can be found in the GitLab repository for the book [18] in the directory **examples/example-simple-service-target**.

The setting I made for the previous service allows staring and stopping the service with systemd manually but systemd does not start the service automatically on system boot. Systemd provides several methods to implement that behavior, with the easiest, but certainly not the best, probably being to add the service to a systemd target.

I described targets shortly in the previous example but not that I will use a target unit in this example, I will describe target units in more detail here first.

A target unit[19] is described in a unit file with the ending **.target**. It describes is used for grouping units and as synchronization point during system start. If we add a unit to a specific target, systemd will start the unit if booting the system to the specific boot target identified by the target unit is requested. Target units are similar to SysV runlevels in this regard.

In normal circumstances systemd boots all targets up to the target unit **default.target** which typically is a symlink to the requested target unit. On my ubuntu system, the **default.target** is symlinked to the target unit **graphical.target** which is defined with the following content.

```
[Unit]
Description=Graphical Interface
Documentation=man:systemd.special(7)
Requires=multi-user.target
Wants=display-manager.service
Conflicts=rescue.service rescue.target
After=multi-user.target rescue.service rescue.target display-manager.service
AllowIsolate=yes
```

Target units have no specific options and are fully described using the options common to all units [20]. The unit file for the **graphical.target** contains the following options:

- Description=Graphical Interface

  The description of the unit. Systemd uses the description in the output of its tools

- Documentation=man:systemd.special(7)

[18] https://gitlab.com/franks_reich/systemd-by-example
[19] https://www.freedesktop.org/software/systemd/man/systemd.target.html
[20] https://www.freedesktop.org/software/systemd/man/systemd.unit.html

A space separated list of urls pointing to the documentation relevant for this unit. Allowed url types are:

  - http://
  - https://
  - file:
  - info:
  - man:

- Requires=multi-user.target

  The Requires option specifies that this unit depends on another unit. Systemd will always start the dependent units when this unit is started. The Requires option does not specify in which order the units will be started and systemd will start both units at the same time if the order is not further specified using other directives. If a unit specifies a dependency to another unit using Requires, that unit will be stopped if the unit it depends on is stopped. Additionally, if the unit specifies that it should be started after the required using the After directive (as is the case here), and the required unit fails to start, this unit's start will be aborted. In this specific case, the unit **graphical.target** specifies that it Requires the unit **multi-user.target** which will always be started when the **graphical.target** is started. Because the unit **graphical.target** also specifies that it should be started after **multi-user.target**, the **graphical.target** will not be started if **multi-user.target** fails.

- Wants=display-manager.service

  The Wants option is a weaker version of Requires. It specifies that the units defined in Wants should also be started if this unit is started. But if any of the units fail to start or are stopped this does not prevent the start of this unit. In this particular example, tbe **graphical.target** defines that it also wants the **display-manager.service** to be started.

- Conflicts=rescue.service rescue.target

  The Conflicts option defines units that cannot be started or running at the same time. In this example, the **graphical.target** is conflicting with the **rescue.service** and the **rescue.target** and will be stopped whenever the **rescue.service** or the **rescue.target** is started.

- After=multi-user.target rescue.service rescue.target display-manager.service

  The After option defines which units have to be completely started before this unit is started. In this specific example the unit **graphical.target** defines that it should be started after the units **multi-user.target**, **rescue.service**, **rescue.target** and **display-manager.service**. Note that only the failure of starting the **multi-user.target** will stop the start of the **graphical.service**.

- AllowIsolate=yes

  Allows the unit to be started using the systemctl isolate command.

Based on the unit **default.target** which links to the unit **graphical.target** I can now state that systemd will normally try to start the **graphical.target**. The **graphical.target** in turn will start the **multi-user.target**. I could further look at the unit file for the **multi-user.target** to determine other targets earlier in the chain to find a suitable target. The example service I want to start does not need a graphical user interface, but I would want to change the user the service will be run from later on. Therefore, the **multi-user.target** should be a suitable choice to use as target for our example unit.

The service from the previous example does not need to be changed and will be used unaltered. It is displayed below only for completeness.

```
import time

while True:
    time.sleep(60)
```

The new unit file is shown below.

```
[Unit]
Description=The sample service

[Service]
Type=simple
ExecStart=/usr/bin/python3 /usr/local/bin/sample_service_1.py

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

I added the new section *Install* and the new option *WantedBy* to the unit file. The *Install* section describes what systemd does when the unit is enabled or disabled with the `systemctl enable` or `systemctl disable` commands. The *WantedBy* option in the *Install* section specifies that the unit, in the following called the current unit, should be started when the unit named in the *WantedBy* option, in the following called the listed unit, is started. When the current unit is enabled, systemd creates a symlink to the unit file of the current unit in the wants directory of the listed unit.

Mapped to the example at hand, this means that, when I enable the unit with the enable command, systemd will create a symbolic link to the unit file **python-sample-2.service** in the directory **multi-user.target.wants**, the wants directory of the **multi-user.target** unit. Systemd should now start the unit **python-sample-2.service** whenever it starts the target unit **multi-user.target**.

Now that I changed the unit file so that it defines to be wanted by the **multi-user.target** I will enable the unit.

```
systemctl enable python-sample-2.service
```

After successful execution of the `systemctl enable` command systemd tells us that it created the required symbolic link in the wants directory.

```
Created symlink /etc/systemd/system/multi-user.target.wants/python-sample-2.service → /etc/systemd/syst
```

Verifying using `ls /etc/systemd/system/multi-user.target.wants/` creates the following output which shows that the unit **python-sample-2.service** is now, among others, wanted by the unit **multi-user.target**.

```
atd.service            irqbalance.service          ondemand.service         remote-fs.target          s
console-setup.service  lxcfs.service               open-vm-tools.service    rsync.service             s
cron.service           lxd-containers.service      pollinate.service        rsyslog.service           s
ebtables.service       networkd-dispatcher.service python-sample-2.service snapd.autoimport.service  s
```

To test if systemd starts the service when restarting the system I restart the system with `shutdown -r now`. Checking the status if the unit **python-sample-2.service** using `systemctl status python-sample-2.service` now yields the following output.

```
 python-sample-2.service - The sample service
   Loaded: loaded (/etc/systemd/system/python-sample-2.service; enabled; vendor preset: enabled)
   Active: active (running) since Tue 2018-10-02 03:55:02 UTC; 23s ago
 Main PID: 189 (python3)
    Tasks: 1 (limit: 4915)
   CGroup: /system.slice/python-sample-2.service
           189 /usr/bin/python3 /usr/local/bin/sample_service_1.py

Oct 02 03:55:02 systemd-by-example systemd[1]: Started The sample service.
```

The output shows that systemd started the service successfully and automatically.

4. Automatically Starting a Simple Service Using Socket Activation

Note: The files for this example can be found in the GitLab repository for the book [21] in the directory **examples/example-simple-service-socket**.

Socket activation is one of the most important features implemented in systemd. Strictly speaking, socked activation is nothing new, it existed on Linux systems for a long time in the form of INETD or XINETD which listened on internet protocol sockets and started the respective services when they received a network request for that socket.

Systemd extends that concept unix sockets in addition to network sockets. As most linux services that are meant to provide services to other services on the local machine use unix sockets, that greatly expands the usability of socket activation to other services.

In addition to starting services on demand which can potentially save resources when the services are not in use, it also allows starting services in parallel regardless of the dependencies between each other. Systemd achieves that by creating the sockets before the service starts and passes the sockets to the services on startup. The clients for the services can therefore start using the sockets even though the servers are not yet started. When the clients write something to the socket, the socket buffers the data. Once the server is ready it can read the buffered data and act on it. The client receives the responses as if the server would have been running all along. If the buffer of the socket fill up, the client blocks. When the server is fully started, the it starts processing the data in the socket and empties the buffer so that client can continue running. The only thing the client notices is that it blocks for a period of time until the server can emtpy the buffer enough for the client to continue. This setup is nice because systemd can start clients and servers in parallel greatly decreasing the time the systems needs to start up. Additionally, socket activation also allows starting of services in a controlled manner without explicitly controlling the start order of the services because the socket are created outside of the servers and can immediately be used by the clients.

Decoupling the socket creation and the server start has additional benefits outside of the system start. When the socket already exists before the process starts, the lifecycle of the socket and the process is decoupled. That means that the process can restart without loosing data as the socket is kept alive and buffers the data until the process is ready again. This could for example be used to update processes or restart failed processes without the clients even noticing.

More information on socket activation and the benefits can be found in Pid Eins - Systemd for Developers 1 [22]. Lennart Poettering also describes a real life example which I do not want to regurgitate here. Interested readers are encouraged to check out Lennart's blog as he describes the scenario much better than I could do anyway. Rather I want to demonstrate socket activation with two simple programs, a client and a server communicating via unix sockets.

I will focus on the server first. In contrast to a server creating its own socket, the server I want to use for the example has to be able to get the socket from an outside source. In practice that requires code changes to the server. However, because systemd provides inetd compatibility, services supporting inetd can be immediately used with systemd. The service I want to use for the example will be newly implemented so I will use the systemd APIs to get the socket from systemd on startup.

Systemd provides a C api for daemons which Linux distributions typically deliver with the systemd library development package. In my Ubuntu system I can install the library and header files using `apt install libsystemd-dev`. Among others, this creates the header file */usr/include/systemd/sd-daemon.h* which defines the daemon related libraries for systemd. The library is documented in the man page for *sd-daemon* [23].

The library defines prefixes for standard error based logging which I will describe in a later section. Further it defines five functions defining the systemd api for the implementation of new style systemd daemons which each are described in their own man page.

[21] https://gitlab.com/franks_reich/systemd-by-example
[22] http://0pointer.de/blog/projects/socket-activation.html
[23] https://www.freedesktop.org/software/systemd/man/sd-daemon.html

- sd_listen_fds [24]

  The functions **fd_listen_fds()** and **fd_listen_fds_with_names()** check for file descriptors passed by the service manager as part of the socket based activation.

- sd_notify [25]

  Function **sd_notify()** and its cousins notify systemd about service status changes like completion of startup, error conditions or shutdown.

- sd_booted [26]

  Function **sd_booted()** checks if the system was booted using systemd.

- sd_is_fifo [27]

  The functions in this group check the file of a file descriptor.

- sd_watchdog_enabled [28]

  Checks if the watchdog capabilities of the service manager are enabled.

Based on the list above, it is apparent that the function to implement socket activation in the server is **sd_listen_fds()**. Because I used python scripts in the previous examples, I would like to continue using python scripts for this example, too. Fortunately, systemd already provides python bindings for the systemd apis including *sd-daemon* [29]. The following python script is used as server in my example.

```
import systemd.daemon as daemon
import socket
import threading


def _create_socket(file_descriptor):
    if daemon.is_socket_unix(file_descriptor) > 0:
        return socket.fromfd(file_descriptor, socket.AF_UNIX, socket.SOCK_STREAM)
    elif daemon.is_socket_inet(file_descriptor, socket.AF_INET) > 0:
        return socket.fromfd(file_descriptor, socket.AF_INET, socket.SOCK_STREAM)
    elif daemon.is_socket_inet(file_descriptor, socket.AF_INET6) > 0:
        return socket.fromfd(file_descriptor, socket.AF_INET6, socket.SOCK_STREAM)
    else:
        return None


def _client_handler(connection):
    while True:
        data = connection[0].recv(4096)
        if not data:
            break
        print('Received from {}: {}'.format(connection[1], data))
        connection[0].send(data)


if __name__ == '__main__':
    socket_file_descriptors = daemon.listen_fds()
```

[24] https://www.freedesktop.org/software/systemd/man/sd_listen_fds.html
[25] https://www.freedesktop.org/software/systemd/man/sd_notify.html
[26] https://www.freedesktop.org/software/systemd/man/sd_booted.html
[27] https://www.freedesktop.org/software/systemd/man/sd_is_fifo.html
[28] https://www.freedesktop.org/software/systemd/man/sd_watchdog_enabled.html
[29] https://www.freedesktop.org/software/systemd/python-systemd/index.html

```
    socket_file_descriptor = socket_file_descriptors[0]
    server_socket = _create_socket(socket_file_descriptor)

    if server_socket is not None:
        while True:
            connection = server_socket.accept()
            handler_tread = threading.Thread(target=_client_handler, args=(connection,))
            handler_tread.start()
```

The test script first retrieves a list of the file descriptors from systemd by calling **listen_fds()**. It then takes the first file descriptor in the array and creates a socket based on the file descriptor in function **_client_handler()**. The function checks first checks which type of socket the file descriptor represents by calling the following functions.

- **is_socket_unix(file_descriptor, socket.AF_UNIX)**

  If called with the parameters displayed above, the function returns an integer greater than 0 if the socket is a unix socket.

- **is_socket_inet(file_descriptor, socket.AF_INET)**

  If called with the parameters displayed above, the function returns an integer greater than 0 if the socket is a IPv4 socket.

- **is_socket_inet(file_descriptor, socket.AF_INET6)**

  If called with the parameters displayed above, the function returns an integer greater than 0 if the socket is a IPv6 socket.

After determining which type of socket the file descriptor represents the server script creates the socket based on the file descriptor using **socket.fromfd()**. It starts listening on the socket automatically because systemd provides the socket already in a listening state. After creating the socket, the server starts an endless while loop waiting for connection requests and accepting them with **accept()**. The returned connection is the passed on the a new thread that handles the connection using the target function **_client_handler()** which prints the data received on the console and also returns it to the client.

Systemd provides tools to test socket activated servers using internet protocol sockets. Calling the socket activated service with `systemd-socket-activate -l 2000 python3 sample_service_3.py` will start listening on localhost with port 2000. I can now connect to the server using telnet by calling `telnet localhost 2000` and the systemd will spawn the server passing along the socket it created previously. The server will not return everything sent to it to telnet which telnet will print on the console. It is unfortunately not possible to test unix sockets using `systemd-socket-activate`.

Because I already tested the server script using internet protocol sockets I will now change the service unit file **python-sample-3.service** so that it does not start automatically with the target anymore. Additionally, I create the socket unit file **python-sample-3.socket** that describes the socket unit used to start the service socket activated.

The service unit file **python-sample-3.service** does not include the *WantedBy* option in the *Install* section anymore so that the service is not started automatically when systemd starts the **multi-user.target**. Instead it mentions the socket unit **python-sample-3.socket** in the *Also* option in the *Install* section as well as in the *After* option in the *Unit* section. This tells systemd that the socket unit has to be created before the service unit as well as that the socket unit should be installed when the service unit is installed.

```
[Unit]
Description=The sample service
After=python-sample-3.socket

[Service]
Type=simple
```

```
ExecStart=/usr/bin/python3 /usr/local/bin/sample_service_3.py

[Install]
Also=python-sample-3.socket
```

The socket unit file **python-sample-3.socket** describes the socket created by systemd. It includes the *Description* option in the the *Unit* section which has the same meaning as in the other units I created so far. It has two options in the *Socket* section which describe the socket created by systemd. The *ListenStream* option instructs systemd to create a stream socket. The socket family is determined by the way the socket address is supplied which is described in the man page in more detail [30]. It typically does what one would expect, e.g. if an ipv4 address including port is specified, systemd creates a socket of the internet protocol v4 family. In this case I specified a file name and systemd creates a unix socket. The *Service* option names the service unit which the socket belongs to.

In the *Install* section I added the socket to the target unit **sockets.target**. This target unit is one of the special units described in the man pages [ˆman-systemd-special] and is used to tell systemd which sockets to activate automatically after boot. The option tells systemd to create a symbolic link in the **sockets.target.wants** directory when enabled.

```
[Unit]
Description=The unix socket for the sample service

[Socket]
ListenStream=/var/lib/python-sample/unix.socket
Service=python-sample-3.service

[Install]
WantedBy=sockets.target
```

After the unit files are created and copied to **/etc/systemd/system** both units can be enabled with `systemctl enable python-sample-3`. Enabling the socket unit does not create the socket though. The output of `systemctl status python-sample-3.socket` is as follows, and indicates that the socket is *inactive*.

```
python-sample-3.socket - The unix socket for the sample service
  Loaded: loaded (/etc/systemd/system/python-sample-3.socket; enabled; vendor preset: enabled)
  Active: inactive (dead)
  Listen: /var/lib/python-sample/unix.socket (Stream)
```

After I start the socket unit using `systemctl start python-sample-3.socket`, systemd creates the unit file which can be verified by looking at the files in the parent directory using `ls /var/lib/python-sample` which should display something similar to the following if systemd could successfully create the socket.

```
unix.socket
```

Additionally systemd should show the socket as activated after starting the socket manually. Verifying with `systemctl status python-sample-3.socket` should yield the following, indicating that the socket is not *active* and *listening*.

```
python-sample-3.socket - The unix socket for the sample service
  Loaded: loaded (/etc/systemd/system/python-sample-3.socket; enabled; vendor preset: enabled)
  Active: active (listening) since Tue 2018-10-02 20:26:48 UTC; 1min 58s ago
  Listen: /var/lib/python-sample/unix.socket (Stream)
  CGroup: /system.slice/python-sample-3.socket

Oct 02 20:26:48 systemd-by-example systemd[1]: Listening on The unix socket for the sample service.
```

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

I can now connect to the unix socket with netcat using `nc -U /var/lib/python-sample/unix.socket` and systemd starts the server process which then returns everything we enter on the console back to netcat. Checking on the status of the service unit using `systemctl status python-sample-3` shows that the service unit is now *active*.

```
 python-sample-3.service - The sample service
   Loaded: loaded (/etc/systemd/system/python-sample-3.service; indirect; vendor preset: enabled)
   Active: active (running) since Tue 2018-10-02 20:30:55 UTC; 2min 40s ago
 Main PID: 3387 (python3)
    Tasks: 1 (limit: 4915)
   CGroup: /system.slice/python-sample-3.service
           3387 /usr/bin/python3 /usr/local/bin/sample_service_3.py

Oct 02 20:30:55 systemd-by-example systemd[1]: Started The sample service.
```

Additionally, checking on the socket unit with reveals that the socket unit's status changed from *active (listening)* to *active (running)* to indicate that systemd is not listening for connections on the socket anymore as the socket was handed over to the service unit.

```
 python-sample-3.socket - The unix socket for the sample service
   Loaded: loaded (/etc/systemd/system/python-sample-3.socket; enabled; vendor preset: enabled)
   Active: active (running) since Tue 2018-10-02 20:39:04 UTC; 1min 13s ago
   Listen: /var/lib/python-sample/unix.socket (Stream)
   CGroup: /system.slice/python-sample-3.socket

Oct 02 20:39:04 systemd-by-example systemd[1]: Listening on The unix socket for the sample service.
```

This concludes my current example which demonstrated using the systemd apis to retrieve file descriptors from the systemd service manager. I tested the service using the systemd tool `systemd-socket-activate` using an internet protocol socket. I also showed how to create the unit files for the service and socket which control the socket activation. After the creation of the socket and service units, I used systemd to start the socket unit and show how systemd starts the service unit and passes the socket on to the service once I connect to the unix socket.

5. Using the Systemd Event Loop and the Systemd Dbus Library

Note: The files for this example can be found in the GitLab repository for the book [31] in the directory **examples/example-sd-event-and-sd-bus**.

Systemd provides two libraries that build the basis for the implementation of systemd tools itself, namely **sd-event** [32] an event loop based on **epoll** and **sd-bus** [33] a DBus library. Lennart Poettering has an introduction to **sd-event** [34] and **sd-bus** [35] on his blog. In this section I will build on his examples and implement a systemd daemon using the systemd libraries **sd-daemon**, **sd-event** and **sd-bus** all in combination with each other.

In the following two sections I will first describe how to implement a very simple program usind **sd-event** and in the following section I will extend the program to handle messages from DBus.

[31] https://gitlab.com/franks_reich/systemd-by-example
[32] https://www.freedesktop.org/software/systemd/man/sd-event.html
[33] https://www.freedesktop.org/software/systemd/man/sd-bus.html
[34] http://0pointer.net/blog/introducing-sd-event.html
[35] http://0pointer.net/blog/the-new-sd-bus-api-of-systemd.html

5.1. Systemd Event Loop sd-event

A typical application based on **sd-event** first created the event loop using one of **sd_event_net()** or **sd_event_default()** functions described in the man pages of **sd_event_default** [36].

Once the event loop is created, the application registers event handler for the event sources it wants to handle. The systemd event loop handles events from the following sources for which the event handler will be registered with different registration functions.

- I/O events

  I/O events are handled using the **epoll** mechanism of the linux kernel and monitor a file described by a file descriptor. The file descriptor can represent a socket, a FIFO, a message queue, a serial connection, a character device or any other file descriptor compatible with **epoll**

  I/O events are registered using **sd_event_add_io()** [37].

- Timer events

  The event loop can handle timer events with different resolution and accuracy. Timer events are registered using function **sd_event_add_time** [38].

- Unix process signal events

  The event loop can handle unix process signals. Unix process signals are registered using the function **sd_event_add_signal** [39].

- Child process state events

  The event loop can listen to state changes of child processes. Handler for events of that type are scheduled using **sd_event_add_child** [40].

- Inotify file system inode events

  The event loop can listen to inotify inode events [41] which can be registered using **sd_event_add_inotify** [42].

- Static event sources

  The event loop can handle the following static event types.

  - Defer

    Defers an event for immediate execution in the next loop run, registered using **sd_event_add_defer** [43].

  - Post

    Defers an event for execution in the next event loop run but only if no other event non-post event is available. Registered using **sd_event_add_post** [44].

  - Exit

    Will be executed when the event loop is terminated using **sd_event_exit**. The functions are registered using **sd_event_add_exit** [45].

[36] https://www.freedesktop.org/software/systemd/man/sd_event_default.html
[37] https://www.freedesktop.org/software/systemd/man/sd_event_add_io.html
[38] https://www.freedesktop.org/software/systemd/man/sd_event_add_time.html
[39] https://www.freedesktop.org/software/systemd/man/sd_event_add_signal.html
[40] https://www.freedesktop.org/software/systemd/man/sd_event_add_child.html
[41] http://man7.org/linux/man-pages/man7/inotify.7.html
[42] https://www.freedesktop.org/software/systemd/man/sd_event_add_inotify.html
[43] https://www.freedesktop.org/software/systemd/man/sd_event_add_defer.html
[44] https://www.freedesktop.org/software/systemd/man/sd_event_add_post.html
[45] https://www.freedesktop.org/software/systemd/man/sd_event_add_exit.html

After the event handler are registered, the application starts the event loop using **sd_event_loop** [46]. The example I will present in the following part does not really do anything exiting. It only sets up the event loop and registers handling for the following unix signals to build a very simple basic event loop for an **sd-event** based application.

- SIGTERM [47]

  The termination signal, signals that the process should be terminated.

- SIGINT [48]

  The keyboard interrupt signal, signals that the process should be interrupted.

The event loop provides default handling for both of these signals and I will use the default handling in the sample application. The source code for the event loop is shown below.

```
#include <stdexcept>
#include <iostream>

#include <signal.h>

#include <systemd/sd-event.h>


using namespace std;

sd_event *createEventLoop() {
    int errorCode = 0;

    cout << "Creating event loop" << "\n";

    sd_event *event = nullptr;
    errorCode = sd_event_default(&event);
    if (errorCode < 0) {
        throw runtime_error("Could not create default event loop");
    }

    return event;
}

void blockSignals() {
    cout << "Clocking signals" << "\n";
    sigset_t signalSet;
    if (sigemptyset(&signalSet) < 0 ||
        sigaddset(&signalSet, SIGTERM) < 0 ||
        sigaddset(&signalSet, SIGINT))
    {
        throw runtime_error("Could not setup signal set");
    }

    if (sigprocmask(SIG_BLOCK, &signalSet, nullptr) < 0) {
        throw runtime_error("Could not block signals");
    }
}
```

[46] https://www.freedesktop.org/software/systemd/man/sd_event_loop.html
[47] http://man7.org/linux/man-pages/man7/signal.7.html
[48] http://man7.org/linux/man-pages/man7/signal.7.html

```
void addSignalHandler(sd_event *event) {
    int errorCode = 0;

    cout << "Adding signal handler" << "\n";
    errorCode = sd_event_add_signal(event, nullptr, SIGTERM, nullptr, nullptr);
    if (errorCode < 0) {
        throw runtime_error("Could not add signal handler for SIGTERM to event loop");
    }

    errorCode = sd_event_add_signal(event, nullptr, SIGINT, nullptr, nullptr);
    if (errorCode < 0) {
        throw runtime_error("Could not add signal handler for SIGINT to event loop");
    }
}

void runEventLoop(sd_event *event) {
    int errorCode = 0;

    cout << "Starting event loop" << "\n";
    errorCode = sd_event_loop(event);
    if (errorCode < 0) {
        throw runtime_error("Error while running event loop");
    }
}

int main(int argc, char *argv[]) {
    sd_event *event = nullptr;

    try {
        event = createEventLoop();
        blockSignals();
        addSignalHandler(event);
        runEventLoop(event);
    } catch (runtime_error &error) {
        cerr << error.what();
    }

    event = sd_event_unref(event);
}
```

Running the sample application produces the following output, indicating that the systemd event loop is running and processing events.

```
Creating event loop
Clocking signals
Adding signal handler
Starting event loop
```

Because there are no events that the event loop processes except for the unix signals I added handler for, the output does not show anything interesting. Adding a timer event to the event loop using **sd_event_add_time()** to output something on standard output every few seconds will change that.

```
#include <stdexcept>
#include <iostream>
```

```
#include <signal.h>

#include <systemd/sd-event.h>


using namespace std;

sd_event *createEventLoop() {
    int errorCode = 0;

    cout << "Creating event loop" << "\n";

    sd_event *event = nullptr;
    errorCode = sd_event_default(&event);
    if (errorCode < 0) {
        throw runtime_error("Could not create default event loop");
    }

    return event;
}

void blockSignals() {
    cout << "Clocking signals" << "\n";
    sigset_t signalSet;
    if (sigemptyset(&signalSet) < 0 ||
        sigaddset(&signalSet, SIGTERM) < 0 ||
        sigaddset(&signalSet, SIGINT))
    {
        throw runtime_error("Could not setup signal set");
    }

    if (sigprocmask(SIG_BLOCK, &signalSet, nullptr) < 0) {
        throw runtime_error("Could not block signals");
    }
}

void addSignalHandler(sd_event *event) {
    int errorCode = 0;

    cout << "Adding signal handler" << "\n";
    errorCode = sd_event_add_signal(event, nullptr, SIGTERM, nullptr, nullptr);
    if (errorCode < 0) {
        throw runtime_error("Could not add signal handler for SIGTERM to event loop");
    }

    errorCode = sd_event_add_signal(event, nullptr, SIGINT, nullptr, nullptr);
    if (errorCode < 0) {
        throw runtime_error("Could not add signal handler for SIGINT to event loop");
    }
}

void runEventLoop(sd_event *event) {
    int errorCode = 0;
```

```
    cout << "Starting event loop" << "\n";
    errorCode = sd_event_loop(event);
    if (errorCode < 0) {
        throw runtime_error("Error while running event loop");
    }
}

int timerCallback(sd_event_source *timerEvent, uint64_t time, void * userData) {
    cout << "Timer was triggered" << "\n";
    uint64_t wakeupTime = 0;
    sd_event *event = sd_event_source_get_event(timerEvent);
    sd_event_now(event, CLOCK_MONOTONIC, &wakeupTime);
    wakeupTime += 1000000;
    sd_event_source_set_time(timerEvent, wakeupTime);
    return 0;
}

int main(int argc, char *argv[]) {
    sd_event *event = nullptr;
    sd_event_source *timerEvent = nullptr;

    try {
        event = createEventLoop();
        blockSignals();
        addSignalHandler(event);

        uint64_t wakeupTime = 0;
        sd_event_now(event, CLOCK_MONOTONIC, &wakeupTime);
        wakeupTime += 1000000;
        sd_event_add_time(
                event, &timerEvent, CLOCK_MONOTONIC, wakeupTime, 0, &timerCallback, nullptr);
        sd_event_source_set_enabled(timerEvent, SD_EVENT_ON);

        runEventLoop(event);
    } catch (runtime_error &error) {
        cerr << error.what();
    }

    timerEvent = sd_event_source_unref(timerEvent);
    event = sd_event_unref(event);
}
```

The function to add a timer event **sd_event_add_time()** takes an absolute time value in microseconds as parameter. To calculate the wakeup time I retrieve the time using **sd_event_now()** and add 1 second (1000000 microseconds) to the current time. Then I schedule the timer to call the callback **timerCallback** at the calculated wakeup time. Additionally, I set the timer event to be a repeated event, because the timer events are one shot events by default. The timer handler calculates a new wake up time and updates the time of the timer event so that the timer is triggered again after one second. Running the application now yields the following output adding "Timer was triggered" every second.

```
Creating event loop
Clocking signals
Adding signal handler
Starting event loop
Timer was triggered
```

```
Timer was triggered
Timer was triggered
Timer was triggered
Timer was triggered
Timer was triggered
Timer was triggered
Timer was triggered
```

5.2. Systemd DBus library sd-bus

An application implementing a D-Bus api exposes interfaces implemented as objects epxosed by a service on a service bus. D-Bus is specified in the D-Bus Specification [49] and guidelines for designing a D-Bus api can be found in the D-Bus API Design Guidelines [50]. Systemd uses D-Bus heavily to allow the control of systemd itself as well as the daemons provided by systemd and also provides its own api to implement D-Bus apis **sd-bus** integrated with **sd-event** and provides a nice c api to implement D-Bus apis.

Lennart Poettering provides a short introduction to **sd-bus** on his blog [51] which I use as a basis to demonstrate how **sd-bus** can be integrated with **sd-event**.

An implementation of a D-Bus api using **sd-bus** typically connects to the bus and then describes the implementation of the provided objects using an object v-table. The bus connection can either be creted using **sd_bus_open_system()**, which connects to the system bus, or **sd_bus_open_user()**, which connects to the user bus. I decided to use **sd_bus_open_user()** for this example to prevent any permission problems when executing the code.

V-tables assign the object's methods and properties to calllback function. Additionally, they store the name and the signature of methods, properties and signals. V-tables are created as array using the special macros defined in **sd-bus-vtable.h** [52] which in turn is imported by **sd-bus.h**. Unfortunately there currently is no man page describing the construction of v-tables so I'll do my best to describe constructing v-tables in the following paragraphs. For the sake of completeness I have to mention that v-tables are not the only way to create D-Bus objects, it is also possible to assign an object directly to a callback using **sd_bus_add_object**. I find it preferable to create objects using v-tables because **sd-bus** handles the multiplexing of the object's methods and properties and, even better, creates all the necessary data for the D-Bus introspection.

As mentioned above, v-tables are constructed as array using the following macros.

- **SD_BUS_VTABLE_START**

  Signals the start of the v-table.

- **SD_BUS_METHOD_WITH_OFFSET**

  Creates a method for the object. The parameters are the name of the method, the signature of the input parameters, the signature of the output parameters, the callback function, flags and an offset parameter. The offset parameter is used to move the pointer to the user data which is registered with **sd_bus_add_object_vtable** to allow more generic implementations.

- **SD_BUS_METHOD**

  Creates a method for the object. The parameters are the name of the method, the signature of the input parameters, the signature of the output parameters, the callback function and flags.

- **SD_BUS_SIGNAL**

[49] https://dbus.freedesktop.org/doc/dbus-specification.html
[50] https://dbus.freedesktop.org/doc/dbus-api-design.html
[51] http://0pointer.net/blog/the-new-sd-bus-api-of-systemd.html
[52] https://github.com/systemd/systemd/blob/385b2eb262a99373f09d01b7f5571dd71a14dc98/src/systemd/sd-bus-vtable.h

Describes a signal for the objects. The parameters are the name of the signal, the signature of the signal and flags.

- **SD_BUS_PROPERTY**

  Creates a read-only property for the object. The parameters are the name of the property, the signature of the property, the get handler, an offset for generic implementations and flags.

- **SD_BUS_WRITABLE_PROPERTY**

  Create a property for the object. The parameters are the name of the property, the signature of the property, the get handler, the set handler, an offset parameters and flags.

- **SD_BUS_VTABLE_END**

  Signals the end of the v-table.

The handler for methods have the following signature.

```
typedef int (*sd_bus_message_handler_t)(sd_bus_message *m, void *userdata, sd_bus_error *ret_error);
```

The handler for getter have the following signature.

```
typedef int (*sd_bus_property_get_t) (sd_bus *bus, const char *path, const char *interface, const char *
```

And the handler for setter have the following signature.

```
typedef int (*sd_bus_property_set_t) (sd_bus *bus, const char *path, const char *interface, const char *
```

Using these macros **sd-bus** allows the definition of any object legal D-Bus object. After the v-table is created, the actual object is registered using **sd_bus_add_object_vtable()**. Once the object is described, the object name has to be requested using **sd_bus_request_name()** [53]. When all this is done, I instruct the systemd event bus **sd-event** to handle **sd-bus** using the function **sd_bus_attach_event()**.

For the example I implement the object *Counter* with the method *IncrementBy* and the writable property *Count*. The method *IncrementBy* increments the *Count* by the integer supplied and setting the property *Count* sets the *Count* to the supplied value.

Enhancing the service implementation from earlier with the D-Bus object yields the following.

```
#include <stdexcept>
#include <iostream>
#include <sstream>

#include <signal.h>

#include <systemd/sd-event.h>
#include <systemd/sd-bus.h>


using namespace std;

sd_event *createEventLoop() {
    int errorCode = 0;

    cout << "Creating event loop" << "\n";

    sd_event *event = nullptr;
    errorCode = sd_event_default(&event);
    if (errorCode < 0) {
```

[53] http://0pointer.de/public/systemd-man/sd_bus_request_name.html

```
        throw runtime_error("Could not create default event loop");
    }

    return event;
}

void blockSignals() {
    cout << "Clocking signals" << "\n";
    sigset_t signalSet;
    if (sigemptyset(&signalSet) < 0 ||
        sigaddset(&signalSet, SIGTERM) < 0 ||
        sigaddset(&signalSet, SIGINT))
    {
        throw runtime_error("Could not setup signal set");
    }

    if (sigprocmask(SIG_BLOCK, &signalSet, nullptr) < 0) {
        throw runtime_error("Could not block signals");
    }
}

void addSignalHandler(sd_event *event) {
    int errorCode = 0;

    cout << "Adding signal handler" << "\n";
    errorCode = sd_event_add_signal(event, nullptr, SIGTERM, nullptr, nullptr);
    if (errorCode < 0) {
        throw runtime_error("Could not add signal handler for SIGTERM to event loop");
    }

    errorCode = sd_event_add_signal(event, nullptr, SIGINT, nullptr, nullptr);
    if (errorCode < 0) {
        throw runtime_error("Could not add signal handler for SIGINT to event loop");
    }
}

struct CounterData {
    uint32_t count = 0;
};

int methodIncrement(sd_bus_message *message, void *userData, sd_bus_error *error) {
    int errorCode = 0;
    uint32_t parameter = 0;
    errorCode = sd_bus_message_read(message, "u", &parameter);
    auto counterData = static_cast<CounterData *>(userData);
    counterData->count += parameter;
    return sd_bus_reply_method_return(message, "u", counterData->count);
}

int propertyCounterGet(
        sd_bus *bus, const char *path, const char *interface, const char *property,
        sd_bus_message *reply, void *userData, sd_bus_error *error)
{
    auto counterData = static_cast<CounterData *>(userData);
```

```
    return sd_bus_message_append(reply, "u", counterData->count);
}

int propertyCounterSet(
        sd_bus *bus, const char *path, const char *interface, const char *property,
        sd_bus_message *message, void *userData, sd_bus_error *error)
{
    int errorCode = 0;
    uint32_t parameter = 0;
    errorCode = sd_bus_message_read(message, "u", &parameter);
    auto counterData = static_cast<CounterData *>(userData);
    counterData->count = parameter;
}

static const sd_bus_vtable counter_vtable[] = {
   SD_BUS_VTABLE_START(0),
   SD_BUS_WRITABLE_PROPERTY("Count", "u", propertyCounterGet, propertyCounterSet, 0, 0),
   SD_BUS_METHOD("IncrementBy", "u", "u", methodIncrement, SD_BUS_VTABLE_UNPRIVILEGED),
   SD_BUS_VTABLE_END
};

void runEventLoop(sd_event *event) {
    int errorCode = 0;

    cout << "Starting event loop" << "\n";
    errorCode = sd_event_loop(event);
    if (errorCode < 0) {
        throw runtime_error("Error while running event loop");
    }
}

int main(int argc, char *argv[]) {
    sd_event *event = nullptr;
    sd_bus *sdBus = nullptr;

    try {
        event = createEventLoop();
        blockSignals();
        addSignalHandler(event);

        int errorCode = 0;
        errorCode = sd_bus_default_user(&sdBus);
        if (errorCode < 0) {
            throw runtime_error("Could not connect to system bus");
        }

        CounterData counterData;
        errorCode = sd_bus_add_object_vtable(
            sdBus,
            nullptr,
            "/net/franksreich/Counter",
            "net.franksreich.Counter",
            counter_vtable,
            &counterData);
```

```
        if (errorCode < 0 ) {
            ostringstream oss;
            oss << "Could not add object v-table, " << strerror(-errorCode);
            throw runtime_error(oss.str());
        }

        errorCode = sd_bus_request_name(sdBus, "net.franksreich.Counter", 0);
        if (errorCode < 0) {
            ostringstream oss;
            oss << "Could not request name, " << strerror(-errorCode);
            throw runtime_error(oss.str());
        }

        errorCode = sd_bus_attach_event(sdBus, event, 0);
        if (errorCode < 0) {
            throw runtime_error("Could not add sd-bus handling to event bus");
        }

        runEventLoop(event);
    } catch (runtime_error &error) {
        cerr << error.what();
    }

    event = sd_event_unref(event);
    sdBus = sd_bus_unref(sdBus);
}
```

Starting the application registers the D-Bus object with the service bus. Systemd provides the tool `busctl` to monitor and test applications connected to the service bus. Calling `busctl --user` shows all D-Bus services currently connected to the service bus and includes the following line for the service registered by the sample application.

```
net.franksreich.Counter    27338 sd_event_loop_w frank    :1.225        session-2.scope
```

D-Bus opens up the services for introspection, the following command will display all methods and properties for the service along with the implemented interfaces.

```
busctl --user introspect net.franksreich.Counter /net/franksreich/Counter
```

The output for the sample service looks similar to the following.

```
NAME                                TYPE      SIGNATURE RESULT/VALUE FLAGS
net.franksreich.Counter             interface -         -            -
.IncrementBy                        method    u         u            -
.Count                              property  u         0            writable
org.freedesktop.DBus.Introspectable interface -         -            -
.Introspect                         method    -         s            -
org.freedesktop.DBus.Peer           interface -         -            -
.GetMachineId                       method    -         s            -
.Ping                               method    -         -            -
org.freedesktop.DBus.Properties     interface -         -            -
.Get                                method    ss        v            -
.GetAll                             method    s         a{sv}        -
.Set                                method    ssv       -            -
.PropertiesChanged                  signal    sa{sv}as  -            -
```

The output shows that the object has the property and the method I implemented. Additionally, **sd-bus**

did a lot of work. It provided implementations for the interfaces *org.freedesktop.DBus.Introspectable*, allowing us to introspect the service using D-Bus tools like `busctl`, *org.freedesktopt.DBus.Peer* and *org.freedesktop.DBus.Properties*.

I can now call the D-Bus method with the following `busctl` call.

```
busctl --user call net.franksreich.Counter /net/franksreich/Counter net.franksreich.Counter IncrementBy
```

Checking the counter with ´busctl –user introspect net.franksreich.Counter /net/franksreich/Counter‘ yields the following.

```
NAME                                TYPE      SIGNATURE RESULT/VALUE FLAGS
net.franksreich.Counter             interface -         -            -
.IncrementBy                        method    u         u            -
.Count                              property  u         5            writable
org.freedesktop.DBus.Introspectable interface -         -            -
.Introspect                         method    -         s            -
org.freedesktop.DBus.Peer           interface -         -            -
.GetMachineId                       method    -         s            -
.Ping                               method    -         -            -
org.freedesktop.DBus.Properties     interface -         -            -
.Get                                method    ss        v            -
.GetAll                             method    s         a{sv}        -
.Set                                method    ssv       -            -
.PropertiesChanged                  signal    sa{sv}as  -            -
```

And I can see that the property *Count* was incremented by 5. The implementation also allows setting the property using `busctl` with the following.

```
busctl --user set-property net.franksreich.Counter /net/franksreich/Counter net.franksreich.Counter Cou
```

Reading the property with the introspect call shows that the property was changed to 13.

```
NAME                                TYPE      SIGNATURE RESULT/VALUE FLAGS
net.franksreich.Counter             interface -         -            -
.IncrementBy                        method    u         u            -
.Count                              property  u         13           writable
org.freedesktop.DBus.Introspectable interface -         -            -
.Introspect                         method    -         s            -
org.freedesktop.DBus.Peer           interface -         -            -
.GetMachineId                       method    -         s            -
.Ping                               method    -         -            -
org.freedesktop.DBus.Properties     interface -         -            -
.Get                                method    ss        v            -
.GetAll                             method    s         a{sv}        -
.Set                                method    ssv       -            -
.PropertiesChanged                  signal    sa{sv}as  -            -
```

The application allows to increment and set the counter. The D-Bus specification defines several standard Interfaces which **sd-bus** either implements or provides functions to ease the implementation of. The standard interfaces are:

- **org.freedesktop.DBus.Introspectable**

  The introspection interface allows the introspection of D-Bus apis connected to the bus. I already used the introspection api when using `busctl` to connect to the service bus, **sd-bus** created the required introspection data based on the v-table registered for the exposed objects.

- **org.freedesktop.DBus.Peer**

The peer api allows to check on which machine the service runs and allows to ping the service. The interface is automatically created by **sd-bus**. Because the functionality is very simple I will not explore the feature further.

- **org.freedesktop.DBus.Properties**

  The properties interface allows accessing the properties of the object. **Sd-bus** implements the method of the interface automatically. It does not implement the signal *PropertiesChanged* because **sd-bus** cannot know how the property is represented and if it changed. I will show how to signal a change in the property value in the next section.

- **org.freedesktop.DBus.ObjectManager**

  The object manager interface allow to retrieve all objects of a service as well as defines signals that indicate if an interface is added or removed from an object. The object manager does not emit signals for property changes in existing objects, this is what the properties interface should be used for, but it emits signals for interfaces that are added or removed if asked to do so using **sd_bus_emit_object_added()** or **sd_bus_emit_object_removed()**. Using this interface in combination with the properties interface allows the client to monitor the service regarding all changes. I will discuss the interface in the next section.

5.3. The Properties and ObjectManager Interfaces

The object manager is created using **sd_bus_add_object_manager()**. The function takes the path as argument and **sd-bus** creates the object manager for that path. The created object manager can handle the method **GetManagedObjects** immediately. It does not automatically emit the signals **InterfacesAdded** or **InterfacesRemoved**. To emit the signals, the application has to call **sd_bus_emit_object_added** or **sd_bus_emit_object_removed** respectively. I enhanced the service in such a way that it exposes the interface **net.franksreich.Manager** on path **/net/franksreich/counter1**. The interface implements the method **AddCounter** which adds a new counter object to the service. When the counter object is added, the application calls **sd_bus_emit_object_added** so that the object manager emits the signal.

In addition to these changes I changed the v-table for the *Counter* object and added the flag **SD_BUS_VTABLE_PROPERTY_EMITS_CHANGE** so that the introspection of the *Counter* object indicated that it sends a **PropertiesChanged** signal when the property changes. To actually send the signal, I added a call to **sd_bus_emit_properties_changed()** in the setter for the property and the handler for the **IncrementBy** method.

The source code for the complete script follows below.

```
#include <stdexcept>
#include <iostream>
#include <sstream>
#include <vector>
#include <memory>

#include <signal.h>

#include <systemd/sd-event.h>
#include <systemd/sd-bus.h>


using namespace std;

sd_event *createEventLoop() {
    int errorCode = 0;
```

```
    cout << "Creating event loop" << "\n";

    sd_event *event = nullptr;
    errorCode = sd_event_default(&event);
    if (errorCode < 0) {
        throw runtime_error("Could not create default event loop");
    }

    return event;
}

void blockSignals() {
    cout << "Clocking signals" << "\n";
    sigset_t signalSet;
    if (sigemptyset(&signalSet) < 0 ||
        sigaddset(&signalSet, SIGTERM) < 0 ||
        sigaddset(&signalSet, SIGINT))
    {
        throw runtime_error("Could not setup signal set");
    }

    if (sigprocmask(SIG_BLOCK, &signalSet, nullptr) < 0) {
        throw runtime_error("Could not block signals");
    }
}

void addSignalHandler(sd_event *event) {
    int errorCode = 0;

    cout << "Adding signal handler" << "\n";
    errorCode = sd_event_add_signal(event, nullptr, SIGTERM, nullptr, nullptr);
    if (errorCode < 0) {
        throw runtime_error("Could not add signal handler for SIGTERM to event loop");
    }

    errorCode = sd_event_add_signal(event, nullptr, SIGINT, nullptr, nullptr);
    if (errorCode < 0) {
        throw runtime_error("Could not add signal handler for SIGINT to event loop");
    }
}

struct CounterData {
    uint32_t count = 0;
};

int methodIncrement(sd_bus_message *message, void *userData, sd_bus_error *error) {
    uint32_t parameter = 0;
    sd_bus_message_read(message, "u", &parameter);
    auto counterData = static_cast<CounterData *>(userData);
    counterData->count += parameter;
    sd_bus *sdBus = sd_bus_message_get_bus(message);
    const char *path = sd_bus_message_get_path(message);
    const char*interface = sd_bus_message_get_interface(message);
    sd_bus_emit_properties_changed(sdBus, path, interface, "Count", nullptr);
```

```
    return sd_bus_reply_method_return(message, "u", counterData->count);
}

int propertyCounterGet(
        sd_bus *bus, const char *path, const char *interface, const char *property,
        sd_bus_message *reply, void *userData, sd_bus_error *error)
{
    auto counterData = static_cast<CounterData *>(userData);
    return sd_bus_message_append(reply, "u", counterData->count);
}

int propertyCounterSet(
        sd_bus *bus, const char *path, const char *interface, const char *property,
        sd_bus_message *message, void *userData, sd_bus_error *error)
{
    uint32_t parameter = 0;
    sd_bus_message_read(message, "u", &parameter);
    auto counterData = static_cast<CounterData *>(userData);
    counterData->count = parameter;
    sd_bus_emit_properties_changed(bus, path, interface, "Count", nullptr);
    return 1;
}

static const sd_bus_vtable counter_vtable[] = {
    SD_BUS_VTABLE_START(0),
    SD_BUS_WRITABLE_PROPERTY(
        "Count", "u", propertyCounterGet,
        propertyCounterSet, 0, SD_BUS_VTABLE_PROPERTY_EMITS_CHANGE),
    SD_BUS_METHOD("IncrementBy", "u", "u", methodIncrement, SD_BUS_VTABLE_UNPRIVILEGED),
    SD_BUS_VTABLE_END
};

void addCounterObjectToService(sd_bus *sdBus, const uint32_t id, CounterData *counterData) {
    ostringstream pathOss;
    pathOss << "/net/franksreich/counter1/" << id;
    int errorCode = 0;
    errorCode = sd_bus_add_object_vtable(
            sdBus,
            nullptr,
            pathOss.str().c_str(),
            "net.franksreich.Counter",
            counter_vtable,
            counterData);
    if (errorCode < 0 ) {
        ostringstream oss;
        oss << "Could not add object v-table, " << strerror(-errorCode);
        throw runtime_error(oss.str());
    }
}

struct ApplicationData {
    sd_bus *sdBus = nullptr;
    vector<unique_ptr<CounterData>> counters;
};
```

```
int methodAddCounter(sd_bus_message *message, void *userData, sd_bus_error *error) {
    uint32_t parameter = 0;
    sd_bus_message_read(message, "u", &parameter);
    auto data = static_cast<ApplicationData *>(userData);
    data->counters.push_back(make_unique<CounterData>());
    data->counters[data->counters.size() - 1]->count = parameter;
    addCounterObjectToService(
        data->sdBus,
        (uint32_t) data->counters.size() - 1,
        data->counters[data->counters.size() - 1].get());
    ostringstream pathOss;
    pathOss << "/net/franksreich/counter1/" << data->counters.size() - 1;
    sd_bus_emit_object_added(data->sdBus, pathOss.str().c_str());
    return sd_bus_reply_method_return(message, "u", data->counters.size() - 1);
}

static const sd_bus_vtable manager_vtable[] = {
    SD_BUS_VTABLE_START(0),
    SD_BUS_METHOD("AddCounter", "u", "u", methodAddCounter, SD_BUS_VTABLE_UNPRIVILEGED),
    SD_BUS_VTABLE_END
};

void runEventLoop(sd_event *event) {
    int errorCode = 0;

    cout << "Starting event loop" << "\n";
    errorCode = sd_event_loop(event);
    if (errorCode < 0) {
        throw runtime_error("Error while running event loop");
    }
}

sd_bus *connectToSystemBus() {
    sd_bus *sdBus = nullptr;
    int errorCode = 0;
    errorCode = sd_bus_default_user(&sdBus);
    if (errorCode < 0) {
        throw runtime_error("Could not connect to system bus");
    }
    return sdBus;
}

void addManagerObjectToService(sd_bus *sdBus, ApplicationData *applicationData) {
    int errorCode = 0;
    errorCode = sd_bus_add_object_vtable(
            sdBus,
            nullptr,
            "/net/franksreich/counter1",
            "net.franksreich.Manager",
            manager_vtable,
            applicationData);
    if (errorCode < 0 ) {
        ostringstream oss;
```

```
        oss << "Could not add object v-table, " << strerror(-errorCode);
        throw runtime_error(oss.str());
    }
}

void requestServiceName(sd_bus *sdBus, string name) {
    int errorCode = 0;
    errorCode = sd_bus_request_name(sdBus, name.c_str(), 0);
    if (errorCode < 0) {
        ostringstream oss;
        oss << "Could not request name, " << strerror(-errorCode);
        throw runtime_error(oss.str());
    }
}

void attachBusHandlingToEventLoop(sd_bus *sdBus, sd_event *event) {
    int errorCode = 0;
    errorCode = sd_bus_attach_event(sdBus, event, 0);
    if (errorCode < 0) {
        throw runtime_error("Could not add sd-bus handling to event bus");
    }
}

int main(int argc, char *argv[]) {
    sd_event *event = nullptr;
    ApplicationData applicationData;

    try {
        event = createEventLoop();
        blockSignals();
        addSignalHandler(event);
        applicationData.sdBus = connectToSystemBus();
        CounterData counterData;
        sd_bus_add_object_manager(applicationData.sdBus, nullptr, "/net/franksreich/counter1");
        addManagerObjectToService(applicationData.sdBus, &applicationData);
        requestServiceName(applicationData.sdBus, "net.franksreich.counter1");
        attachBusHandlingToEventLoop(applicationData.sdBus, event);
        runEventLoop(event);
    } catch (runtime_error &error) {
        cerr << error.what();
    }

    event = sd_event_unref(event);
    applicationData.sdBus = sd_bus_unref(applicationData.sdBus);
}
```

After I start the application, it registers itself under the path **/net/franksreich/counter1**. The `busctl` tool allows monitoring messages on the service bus. Running the tool with `busctl --user monitor net.franksreich.counter1` will output all messages relating to the service of the sample application on standard output.

I use `busctl` with the following call to add a new counter.

```
busctl --user call net.franksreich.counter1 /net/franksreich/counter1 net.franksreich.Manager AddCounte
```

This yields the following output on the monitoring `busctl` on standard output, indicating that the object

manager send the **InterfacesAdded** signal.

```
Type=signal  Endian=l  Flags=1  Version=1  Priority=0 Cookie=3
Sender=:1.502  Path=/net/franksreich/counter1  Interface=org.freedesktop.DBus.ObjectManager  Member=I
UniqueName=:1.502
MESSAGE "oa{sa{sv}}" {
        OBJECT_PATH "/net/franksreich/counter1/0";
        ARRAY "{sa{sv}}" {
                DICT_ENTRY "sa{sv}" {
                        STRING "org.freedesktop.DBus.Peer";
                        ARRAY "{sv}" {
                        };
                };
                DICT_ENTRY "sa{sv}" {
                        STRING "org.freedesktop.DBus.Introspectable";
                        ARRAY "{sv}" {
                        };
                };
                DICT_ENTRY "sa{sv}" {
                        STRING "org.freedesktop.DBus.Properties";
                        ARRAY "{sv}" {
                        };
                };
                DICT_ENTRY "sa{sv}" {
                        STRING "org.freedesktop.DBus.ObjectManager";
                        ARRAY "{sv}" {
                        };
                };
                DICT_ENTRY "sa{sv}" {
                        STRING "net.franksreich.Counter";
                        ARRAY "{sv}" {
                                DICT_ENTRY "sv" {
                                        STRING "Count";
                                        VARIANT "u" {
                                                UINT32 5;
                                        };
                                };
                        };
                };
        };
};
```

Calling the method **IncrementBy** of the counter object I just added using `busctl --user call net.franksreich.counter1 /net/franksreich/counter1/0 net.franksreich.Counter IncrementBy u 5` yields the following output indicating that the property changed by signalling a **PropertiesChanged** signal.

```
Type=signal  Endian=l  Flags=1  Version=1  Priority=0 Cookie=9
Sender=:1.529  Path=/net/franksreich/counter1/0  Interface=org.freedesktop.DBus.Properties  Member=Pr
UniqueName=:1.529
MESSAGE "sa{sv}as" {
        STRING "net.franksreich.Counter";
        ARRAY "{sv}" {
                DICT_ENTRY "sv" {
                        STRING "Count";
```

```
                        VARIANT "u" {
                                UINT32 35;
                        };
                };
        };
        ARRAY "s" {
        };
};
```

6. Automatically Starting a Service using D-Bus Activation

Note: The files for this example can be found in the GitLab repository for the book [54] in the directory **examples/example-service-dbus-activation**.

Similar to socket activation systemd allows activating applications on demand using the D-Bus bus. Activating a service using D-Bus is strictly speaking a feature of D-Bus. To know which service is provides by which executable D-Bus uses service files in the directory **/usr/share/dbus-1/services** or **/usr/share/dbus-1/system-services**. The former is used for system services, the latter for session services. To connect the D-Bus service file with a systemd service unit, the configuration file provides the *SystemdService* key which can be used to provide the name of a systemd unit that will be started instead of the normally supplied executable. Because systemd unit have to provide the user name that is used to run the executable and services on the session bus run under the user name to whom the session bus belongs, using systemd services in D-Bus service files is only meaningful for system services.

Therefore I will change the application we used from the prior example so that it connects to the system bus instead of the session bus by changing the call to `sd_bus_default_user()` to `sd_bus_default_system()` to connect to the system bus instead.

The name of the D-Bus service file can be chosen arbitrarily, the D-Bus daemon will scan all the files in the respective directories for a matching *BusName* option when it encounters a call to a service that was not started beforehand. I use following D-Bus service file to configure starting the test service.

```
[D-BUS Service]
Name=net.franksreich.counter1
Exec=/usr/bin/d-bus-activation-example
User=root
SystemdService=d-bus-activation-example.service
```

The D-Bus service file describes the service to be started. It specifies that it provides a service named **net.franksreich.counter1** specified in the *Name* option. The *Exec* option specifies the name of the executable and the *User* option specifies that the service should be run by the user root. The only systemd related option *SystemdService* specifies the name of the unit that should be started instead of starting the executable directly.

I use the following systemd unit file to start the systemd service.

```
[Unit]
Description=Frank's Reich Counter Service

[Service]
Type=dbus
BusName=net.franksreich.counter1
ExecStart=/usr/local/bin/d-bus-activation-example
```

[54] https://gitlab.com/franks_reich/systemd-by-example

The systemd unit file states that the type of the service is D-Bus in the *Type* option. It mentions the name of the service as **net.franksreich.counter1**, the same name as in the D-Bus service file. The executable is the same executable mentioned in the D-Bus service file, too.

To test the example I copy the D-Bus service file **d-bus-activation-example.d-bus.service** to **/usr/share/dbus-1/system-services** and **d-bus-activation-example.service** to **/etc/systemd/system**. Additionally I copy the executable **d-bus-activation-example** to the directory **/usr/bin/**.

The service unit does not include an *Install* section and therefore does not have to be enabled. The D-Bus daemon automatically scans the services files in the configuration directory. Calling `busctl` lists the service I registered as activatable.

```
...
fi.epitest.hostap.WPASupplicant           883 wpa_supplicant  root             :1.10         wpa_suppli
fi.w1.wpa_supplicant1                     883 wpa_supplicant  root             :1.10         wpa_suppli
net.franksreich.counter1                    - -               -                (activatable) -
net.hadess.SensorProxy                    850 iio-sensor-prox root             :1.3          iio-senso
org.bluez                                 848 bluetoothd      root             :1.5          bluetooth
...
```

Trying to start the service using `busctl introspect net.franksreich.counter1 /net/franksreich/counter1` or `sudo busctl introspect net.franksreich.counter1 /net/franksreich/counter1` yields the following error.

```
Failed to introspect object /net/franksreich/counter1 of service net.franksreich.counter1: Access denie
```

This happens because the service does not have a policy definition so D-Bus prevents everybody to call the service as default setting. Policy definitions for system bus services are located in **/etc/dbus-1/system.d** and policy definitions for session bus services are located in **/etc/dbus-1/session.d**. I create the config file **net.franksreich.counter1.conf** in directory **/etc/dbus-1/system.d** with the following content.

```
<!DOCTYPE
    busconfig PUBLIC
    "-//freedesktop//DTD D-BUS Bus Configuration 1.0//EN"
    "http://www.freedesktop.org/standards/dbus/1.0/busconfig.dtd">

<busconfig>
  <policy user="root">
    <allow own="net.franksreich.counter1" />
  </policy>

  <policy context="default">
    <allow send_destination="net.franksreich.counter1" />
    <allow receive_sender="net.franksreich.counter1" />
  </policy>
</busconfig>
```

The configuration file defines a node with two nodes. The first node specifies that user *root* is allowed to tkae the name **net.franksreich.counter1** name. The second node specifies that by default, as indicated with the attribute *context="default"*, everybody can send and receive messages to and from the service. The configuration file uses an xml language defined in the D-Bus daemon man page [55].

Calling `busctl introspect net.franksreich.counter1 /net/franksreich/counter1` now yields the following.

```
NAME                                TYPE      SIGNATURE  RESULT/VALUE  FLAGS
net.franksreich.Manager             interface -          -             -
```

[55] https://dbus.freedesktop.org/doc/dbus-daemon.1.html

```
.AddCounter                            method    u          u            -
org.freedesktop.DBus.Introspectable interface -          -            -
.Introspect                            method    -          s            -
org.freedesktop.DBus.ObjectManager  interface -          -            -
.GetManagedObjects                     method    -          a{oa{sa{sv}}} -
.InterfacesAdded                       signal    oa{sa{sv}} -            -
.InterfacesRemoved                     signal    oas        -            -
org.freedesktop.DBus.Peer              interface -          -            -
.GetMachineId                          method    -          s            -
.Ping                                  method    -          -            -
org.freedesktop.DBus.Properties     interface -          -            -
.Get                                   method    ss         v            -
.GetAll                                method    s          a{sv}        -
.Set                                   method    ssv        -            -
.PropertiesChanged                     signal    sa{sv}as   -            -
```

Calling `busctl` shows that the service has been activated.

```
...
com.ubuntu.WhoopsiePreferences               - -               -               (activatable) -
fi.epitest.hostap.WPASupplicant            883 wpa_supplicant  root            :1.10         wpa_suppli
fi.w1.wpa_supplicant1                      883 wpa_supplicant  root            :1.10         wpa_suppli
net.franksreich.counter1                  1435 d-bus-activatio root            :1.30257      d-bus-acti
net.hadess.SensorProxy                     850 iio-sensor-prox root            :1.3          iio-senso
org.bluez                                  848 bluetoothd      root            :1.5          bluetooth
...
```

Calling `systemctl status d-bus-activation-example.service` also shows that the service is active.

```
 d-bus-activation-example.service - Frank's Reich Counter Service
   Loaded: loaded (/etc/systemd/system/d-bus-activation-example.service; static; vendor preset: enabled)
   Active: active (running) since Mon 2018-10-08 19:17:47 EDT; 4min 59s ago
 Main PID: 1435 (d-bus-activatio)
    Tasks: 1 (limit: 4915)
   CGroup: /system.slice/d-bus-activation-example.service
           1435 /usr/bin/d-bus-activation-example

Oct 08 19:17:47 wotan systemd[1]: Starting Frank's Reich Counter Service...
Oct 08 19:17:47 wotan systemd[1]: Started Frank's Reich Counter Service.
```

D-Bus now starts the service using the systemd unit file which allows the service to provide its interface to its clients and allows systemd to manage the service.

7. Interacting with Systemd using the D-Bus API

Systemd provides a D-Bus api that can be used to interface and communicate with systemd. As usual for D-Bus apis the api can be introspected by calling `busctl introspect org.freedesktop.systemd1 /org/freedesktop/systemd1`. The service implements the interface **org.freedesktop.systemd1.Manager** implementing methods that can be used to retrieve information about systemd and control systemd. It implements to many methods to list here, so I will focus on a few exemplary methods demonstrating what is possible with systemd. Interested readers are encouraged to explore the provided methods based on the examples provided here.

- **ListUnits**

The method **ListUnits** of the interface **org.freedesktop.systemd1.Manager** list all available units. Not all units are necessarily represented by a unit file because systemd supports transient units and units created from other configuration files like **/etc/fstab**. The list for my system is to long to duplicate here so the following shows an example output line.

```
"avahi-daemon.service" "Avahi mDNS/DNS-SD Stack" "loaded" "active" "running"
```

- **StartUnit**

  The method **StartUnit** of the interface **org.freedesktop.systemd1.Manager** starts the given unit using the start mode. The start mode can be one of the following.

  - replace
  - fail
  - isolate
  - ignore-dependencies
  - ignore-requirements

  Calling `busctl call org.freedesktop.systemd1 /org/freedesktop/systemd1 org.freedesktop.systemd1.Mana` `StartUnit "ss" python-sample-1.service replace` can be used to start the unit of the first example using the D-Bus api. The output will be as follows.

  ```
  o "/org/freedesktop/systemd1/job/43068"
  ```

The **org.freedesktop.systemd1.Manager** has many more methods all of which are described in "The D-Bus API of systemd/PID 1" [56].

In addition to the **Manager** interface, systemd also provides an interface based on the unit type. The interface and the unit type is shown in the table below.

Unit Type	Interface
Service	org.freedesktop.systemd1.Service
Socket	org.freedesktop.systemd1.Socket
Target	org.freedesktop.systemd1.Target
Device	org.freedesktop.systemd1.Device
Mount	org.freedesktop.systemd1.Mount
Automount	org.freedesktop.systemd1.Automount
Timer	org.freedesktop.systemd1.Timer
Swap	org.freedesktop.systemd1.Swap
Path	org.freedesktop.systemd1.Path
Slice	org.freedesktop.systemd1.Slice
Scope	org.freedesktop.systemd1.Scope
Job	org.freedesktop.systemd1.Job

Each of the interfaces provide specific functionality for the respective unit type. Using the introspection by calling `busctl introspect org.freedesktop.systemd1 /org/freedesktop/systemd1/unit/d_2dbus_2dactivation_2de` This yields a very long output so I only display it shortened, removing most of the methods but preserving the implemented interfaces.

```
NAME                                TYPE
org.freedesktop.DBus.Introspectable interface
...
org.freedesktop.DBus.Peer           interface
...
org.freedesktop.DBus.Properties     interface
...
```

[56] https://www.freedesktop.org/wiki/Software/systemd/dbus/

```
org.freedesktop.systemd1.Service    interface
...
org.freedesktop.systemd1.Unit       interface
...
```

In addition to the standard D-Bus interfaces **Introspectable**, **Peer** and **Properties** systemd implements the interfaces **org.freedesktop.systemd1.Unit** and because it is a service unit it implements **org.freedesktop.systemd1.Service**. Keep in mind that every unit implements the **org.freedesktop.system1.Unit** interfaces. I can for example get the status of a specific unit by calling `busctl get-property org.freedesktop.systemd1 /org/freedesktop/systemd1/unit/d_2dbus_2dactivation_2dexamp org.freedesktop.systemd1.Unit ActiveState`.

Because I started the unit earlier using the systemd api the output will show that the unit is in an active state.

```
s "active"
```

Because systemd implements the standard interface **org.freedesktop.Properties** we can observe the state of units using signals. Calling `sudo busctl monitor org.freedesktop.systemd1` will show all messages on the console. If I shut down the service using `systemctl stop python-sample-1` the following output shows that systemd send the signal to inform us of the properties change. This can be used to monitor the status of services.

```
 Type=signal  Endian=l  Flags=1  Version=1  Priority=0 Cookie=150164
  Sender=:1.0  Path=/org/freedesktop/systemd1/unit/python_2dsample_2d1_2eservice  Interface=org.freedesk
...
```

Using the D-Bus api on the console is cumbersome and the command line tools provide a better interface for that purpose. The D-Bus interface is very useful for for use in other applications. Nearly every programming language provides D-Bus bindings, and in the next chapter I will provide an example of how to use the D-Bus api programmatically with **sd-bus**.

8. Using the Systemd D-Bus API with sd-bus

Note: The files for this example can be found in the GitLab repository for the book [57] in the directory **examples/example-sd-bus-client**.

The real power of the systemd D-Bus api can only be realized by using the api programmatically. There are D-Bus bindings available for many different programming languages, a list of D-Bus libraries can be found on freedesktop.org [58].

8.1. Calling D-Bus Methods

Because I want to focus the tutorial on systemd and systemd provides its own D-Bus C library **sd-bus**, I will use this library here to develop a sample application. Lennart Poettering described the library on his blog [59] and I use his example code a basis for my example here.

The first example just starts the service unit **python-sample-1.service** which I created in a previous example. The code for the sample client follows below.

```
#include <iostream>
#include <sstream>
#include <stdexcept>
```

[57] https://gitlab.com/franks_reich/systemd-by-example
[58] https://www.freedesktop.org/wiki/Software/DBusBindings/
[59] http://0pointer.net/blog/the-new-sd-bus-api-of-systemd.html

```cpp
#include <systemd/sd-bus.h>


using namespace std;

int main(int argc, char *argv[]) {
    sd_bus_error sdBusError = SD_BUS_ERROR_NULL;
    sd_bus_message *message = nullptr;
    sd_bus *sdBus = nullptr;
    char *path;
    int errorCode = 0;

    try {
        errorCode = sd_bus_default_system(&sdBus);
        if (errorCode < 0) {
            ostringstream errorMessage;
            errorMessage << "Failed to connect to system bus, error: "
                         << strerror(-errorCode);
            throw runtime_error(errorMessage.str());
        }

        errorCode = sd_bus_call_method(
                sdBus,
                "org.freedesktop.systemd1",
                "/org/freedesktop/systemd1",
                "org.freedesktop.systemd1.Manager",
                "StartUnit",
                &sdBusError,
                &message,
                "ss",
                "python-sample-1.service",
                "replace");
        if (errorCode < 0) {
            ostringstream errorMessage;
            errorMessage << "Failed to start python-sample-1.service, error: "
                         << strerror(-errorCode);
            throw runtime_error(errorMessage.str());
        }

        errorCode = sd_bus_message_read(message, "o", &path);
        if (errorCode < 0) {
            ostringstream errorMessage;
            errorMessage << "Failed to parse response message, error: "
                         << strerror(-errorCode);
            throw runtime_error(errorMessage.str());
        }

        cout << "Queued service job as: " << path;
    } catch (runtime_error &error) {
        cout << error.what() << "\n";
    }

    sd_bus_error_free(&sdBusError);
```

```
    sd_bus_message_unref(message);
    sd_bus_unref(sdBus);
}
```

As the server we implemented in a previous example, the client first connects to the corresponding bus, in this case the system bus, using **sd_bus_default_system()**. After connecting successfully, the client calls a method on the bus using **sd_bus_call_method()**, passing the following arguments.

- **bus**

  The pointer to the D-Bus

- **destination**

  The service name the service registered on the bus.

- **path**

  The path of the object to be called.

- **interface**

  The interface to be called.

- **member**

  The method to be called.

- **ret_error**

  The D-Bus error to be called.

- **reply**

  A pointer that will contain the reply message.

- **types**

  The signature of the method in the D-Bus signature description.

- **...**

  The variables to be used as parameters for the method call.

When the client calls the method using D-Bus, systemd creates a systemd job that in turn starts the requested application. Running the client yields the following output on the console.

```
Queued service job as: /org/freedesktop/systemd1/job/55902frank@wotan:~/Documents/Development/systemd-by
```

Checking on the status of the call using `systemctl status python-sample-1.service` yields the following output and indicates that starting the service was successful.

```
 python-sample-1.service - The sample service
   Loaded: loaded (/etc/systemd/system/python-sample-1.service; static; vendor preset: enabled)
   Active: active (running) since Mon 2018-10-08 23:45:02 EDT; 12h ago
 Main PID: 7707 (python3)
    Tasks: 1 (limit: 4915)
   CGroup: /system.slice/python-sample-1.service
           7707 /usr/bin/python3 /usr/local/bin/sample_service_1.py

Oct 08 23:45:02 wotan systemd[1]: Started The sample service.
```

8.2. Listening to D-Bus Signals

The sample client allows us to call the systemd D-Bus api and start a systemd service unit. In the next example I will demonstrate how to listen to signals from the systemd D-Bus service to determine when the service unit changes status.

The client listening to the signal has the following source code.

```
#include <iostream>
#include <sstream>
#include <stdexcept>

#include <systemd/sd-bus.h>
#include <systemd/sd-event.h>


using namespace std;

sd_event *createEventLoop() {
    int errorCode = 0;

    cout << "Creating event loop" << "\n";

    sd_event *event = nullptr;
    errorCode = sd_event_default(&event);
    if (errorCode < 0) {
        throw runtime_error("Could not create default event loop");
    }

    return event;
}

void blockSignals() {
    cout << "Clocking signals" << "\n";
    sigset_t signalSet;
    if (sigemptyset(&signalSet) < 0 ||
        sigaddset(&signalSet, SIGTERM) < 0 ||
        sigaddset(&signalSet, SIGINT))
    {
        throw runtime_error("Could not setup signal set");
    }

    if (sigprocmask(SIG_BLOCK, &signalSet, nullptr) < 0) {
        throw runtime_error("Could not block signals");
    }
}

void addSignalHandler(sd_event *event) {
    int errorCode = 0;

    cout << "Adding signal handler" << "\n";
    errorCode = sd_event_add_signal(event, nullptr, SIGTERM, nullptr, nullptr);
    if (errorCode < 0) {
        throw runtime_error("Could not add signal handler for SIGTERM to event loop");
    }
```

```
    errorCode = sd_event_add_signal(event, nullptr, SIGINT, nullptr, nullptr);
    if (errorCode < 0) {
        throw runtime_error("Could not add signal handler for SIGINT to event loop");
    }
}

void runEventLoop(sd_event *event) {
    int errorCode = 0;

    cout << "Starting event loop" << "\n";
    errorCode = sd_event_loop(event);
    if (errorCode < 0) {
        throw runtime_error("Error while running event loop");
    }
}

int signalCallback(sd_bus_message *message, void *userData, sd_bus_error *returnError) {
    int errorCode = 0;
    char *interfaceName = nullptr;
    errorCode = sd_bus_message_read_basic(message, 's', &interfaceName);
    if (errorCode < 0) {
        ostringstream errorMessage;
        errorMessage << "could not decode message, error: "
                     << strerror(-errorCode);
        throw runtime_error(errorMessage.str());
    }

    errorCode = sd_bus_message_enter_container(message, SD_BUS_TYPE_ARRAY, "{sv}");
    if (errorCode < 0) {
        ostringstream errorMessage;
        errorMessage << "could not decode message, error: "
                     << strerror(-errorCode);
        throw runtime_error(errorMessage.str());
    }

    while ((errorCode = sd_bus_message_enter_container(message, SD_BUS_TYPE_DICT_ENTRY, "sv")) > 0) {
        char *attributeName = nullptr;
        errorCode = sd_bus_message_read_basic(message, 's', &attributeName);
        if (errorCode < 0) {
            ostringstream errorMessage;
            errorMessage << "could not read attribute name, error: "
                         << strerror(-errorCode);
            throw runtime_error(errorMessage.str());
        }

        const char *contents = nullptr;
        errorCode = sd_bus_message_peek_type(message, nullptr, &contents);
        if (errorCode < 0) {
            ostringstream errorMessage;
            errorMessage << "could not read peek type, error: "
                         << strerror(-errorCode);
            throw runtime_error(errorMessage.str());
        }
```

```
            cout << "Read attribute with name " << attributeName
                 << " with type " << contents << "\n";

            errorCode = sd_bus_message_enter_container(message, SD_BUS_TYPE_VARIANT, contents);
            if (errorCode < 0) {
                ostringstream errorMessage;
                errorMessage << "could not enter variant, error: "
                             << strerror(-errorCode);
                throw runtime_error(errorMessage.str());
            }

            if (string(contents) == "u") {
                uint32_t unsignedIntValue;

                errorCode = sd_bus_message_read_basic(message, contents[0], &unsignedIntValue);
                if (errorCode < 0) {
                    ostringstream errorMessage;
                    errorMessage << "could read uint32 parameter, error: "
                                 << strerror(-errorCode);
                    throw runtime_error(errorMessage.str());
                }

                cout << "Attribute value " << unsignedIntValue << "\n";
            } else if (string(contents) == "s") {
                const char *stringAttribute = nullptr;

                errorCode = sd_bus_message_read_basic(message, contents[0], &stringAttribute);
                if (errorCode < 0) {
                    ostringstream errorMessage;
                    errorMessage << "could read string parameter, error: "
                                 << strerror(-errorCode);
                    throw runtime_error(errorMessage.str());
                }

                cout << "Attribute value " << stringAttribute << "\n";
            }

            errorCode = sd_bus_message_exit_container(message);
            if (errorCode < 0) {
                ostringstream errorMessage;
                errorMessage << "could not exit container, error: "
                             << strerror(-errorCode);
                throw runtime_error(errorMessage.str());
            }
        }

        cout << "Received signal for interface " << interfaceName << "\n";
        return errorCode;
    }

    int main(int argc, char *argv[]) {
        sd_bus_error sdBusError = SD_BUS_ERROR_NULL;
        sd_bus_message *message = nullptr;
```

```
    sd_bus *sdBus = nullptr;
    sd_event *event = nullptr;
    char *path;
    int errorCode = 0;

    try {
        event = createEventLoop();
        blockSignals();
        addSignalHandler(event);

        errorCode = sd_bus_default_system(&sdBus);
        if (errorCode < 0) {
            ostringstream errorMessage;
            errorMessage << "Failed to connect to system bus, error: "
                         << strerror(-errorCode);
            throw runtime_error(errorMessage.str());
        }

        errorCode = sd_bus_match_signal(
                sdBus,
                nullptr,
                "org.freedesktop.systemd1",
                "/org/freedesktop/systemd1/unit/python_2dsample_2d1_2eservice",
                "org.freedesktop.DBus.Properties",
                "PropertiesChanged",
                signalCallback,
                nullptr);
        if (errorCode < 0) {
            ostringstream errorMessage;
            errorMessage << "Failed to match signal, error: "
                         << strerror(-errorCode);
            throw runtime_error(errorMessage.str());
        }

        errorCode = sd_bus_attach_event(sdBus, event, 0);
        if (errorCode < 0) {
            throw runtime_error("Could not add sd-bus handling to event bus");
        }

        runEventLoop(event);
    } catch (runtime_error &error) {
        cout << error.what() << "\n";
    }

    event = sd_event_unref(event);
    sd_bus_error_free(&sdBusError);
    sd_bus_message_unref(message);
    sd_bus_unref(sdBus);
}
```

Because the client has to wait for the signal to be emitted, it has to run in a loop. I used the same code as in the event loop example to create and set up the event loop. The code is exactly the same as before so I decided to not describe it here again.

After the client created the event loop, it registers the signal handler with systemd using the function

sd_bus_match_signal(). The function takes the following parameters.

- **sd_bus**

  The service bus the client is connected to.

- **return**

  The **sd_bus_slot**. The slot is used to manage the lifetime of the signal handler. If the lifetime should be the same as the bus, it is possible to pass a null pointer and **sd-bus** manages the lifetime.

- **sender**

  The sender interface the client is listening for. The sender interface is "org.freedesktop.systemd1".

- **path**

  The path the client is listening for. I am listening to the path of the specific serivce unit **python-sample-1.service**. Note the the path has to be escaped. The '-' is escaped with '_2d' and '.' is escaped with '_2e'.

- **interface**

  The interface the client is listening for. In the example the interface is the standard D-Bus interface **org.freedesktop.DBus.Properties**.

- **member**

  The signal the client is listening for. In the example the signal is *PropertiesChanged*.

- **callback**

  The callback function to be called when the signal is received. The callback in the example is the function **signalCallback()** which handles the message received. The callback will be described in detail in the following section.

- **userdata**

  A pointer to user data **sd-bus** will pass in the callback when it calls the callback.

8.2.1. The Callback Function

When **sd-bus** retrieves the signal on the bus, it calls the callback function registered using the function **sd_bus_match_signal()**. The callback function takes the following parameters.

- **message**

  The message the signal sends.

- **userData**

  A pointer of type *void* which can be registered when registering the callback function. **Sd-bus** will pass the pointer when it receives the signal and calls the callback function.

- **returnError**

  An error that can be returned.

The callback in the example receives the signal *PropertiesChanged*. The signal emits a message with the D-Bus signature `sa{sv}as`, showing that the messages has a value of type *string*, followed by an array with a dictionary with a key of type *string* and a value of type *variant*, followed by an array of type *string*. Further information about the D-Bus type system can be found in teh D-Bus Specification [60].

[60] https://dbus.freedesktop.org/doc/dbus-specification.html

8.2.1.1. Parsing the Message

The callback function parses the message and prints the message content to the console. The signature only describes the types contained in the message, not the meaning. The standard D-Bus interface **org.freedesktop.DBus.Properties** is described in the [61] and the content of the signal in detail are as follows.

- **interface_name** of type *string*
- **changed_properties** a dictionary with key of type *string* and a value of type *variant*
- **invalidated_properties** an array of *string*

I only implemented reading of the **interface_name** and the **changed_properties** in the example because the implementation of reading the **invalidated_properties** can be done along the lines of the **changed_properties** and would not add additional information.

First the service reads the **interface_name** which is either **org.freedesktop.systemd1.Unit** or **org.freedesktop.systemd1.Service**, because the unit I am listening for changes in is a service unit and implements these two interfaces. Because the **interface_name** is a simple *string* it can be read with just on call to the function **sd_bus_message_read_basic** which can be used to read a single value of a given type. The type can be any of the basic types described in the systemd specification [62]. The **interface_name** is a *string* and I pass *'s'* as type parameter. The function takes the message and a pointer to store the value in addition to the type parameter. It also returns an error code in case something goes wrong. The full call looks like this `sd_bus_message_read_basic(message, 's', &interfaceName)`.

Reading an attribute with **sd_bus_message_read()** also moves the current position in the message along so that we are now ready to read the next part of the message.

Reading the **changed_properties** is more complicated than reading a single attribute because the **changed_properties** is an array of dictionaries. To access the array to access container in the message. Container are arrays, structs, variants or dict_entries as described in the D-Bus Specification [63]. To access a container, we have to enter it using the function **sd_bus_message_enter_container()** and exit it using message **sd_bus_message_exit_container()**. Depending on the type of the container the function **sd_bus_enter_container()** behaves differently based on the type of the container.

- Entering a container of type *array*

  Calling **sd_bus_message_enter_container()** if the current position in the message is at the beginning of an array enters the array and move the current position to the beginning of that array element. Simple elements in the array can be loaded by calling **sd_bus_message_read_basic()** and container in the array can be entered by calling **sd_bus_message_container_enter()**. Calling **sd_bus_message_container_enter()** for the elements in the array will open one element in the array after the other, returning an error code < 0 if there are no more elements in the array left.

- Entering a container of type *dictionary*

  Entering a container of type dictionary by calling **sd_bus_enter_container()** allows reading the contents of the dictionary. After the contents have been read, the container can be exited using **sd_bus_exit_container()**.

- Entering a container of type *variant*

  Entering a container of type variant allows accessing the contained value. Because the value can be of any single complete type, the type has to be determined before it can be entered. The type in the variant can be determined by calling **sd_bus_message_peek_type()** which returns a string describing the type of the variant. Once the type has been determined, the variant can be entered by

[61] https://dbus.freedesktop.org/doc/dbus-specification.html
[62] https://dbus.freedesktop.org/doc/dbus-specification.html
[63] https://dbus.freedesktop.org/doc/dbus-specification.html

calling **sd_bus_message_container_enter**. Once all desired values are read, the variant can be exited by calling **sd_bus_message_exit_container**.

- Entering a container of type *struct*

  TODO!!! How to access a struct correctly?

The function to enter a container **sd_bus_message_enter_container()** takes three parameters, the message, the type of the container and the type of its contents. The type of the contents is a type string. The type of the container can be selected by one of the following constants.

- SD_BUS_TYPE_ARRAY
- SD_BUS_TYPE_DICT_ENTRY
- SD_BUS_TYPE_VARIANT
- SD_BUS_TYPE_STRUCT

Therefore to read the **changed_properties** with type *a{sv}*, the client first enters the array by calling `sd_bus_message_enter_container(message, SD_BUS_TYPE_ARRAY, {sv})`. It then calls `sd_bus_message_enter_container(message, SD_BUS_TYPE_DICT_ENTRY, "sv")` in a while loop to enter each dictionary element in the array one after the other. Once the dictionary container was entered, the client can first read the dictionary key of type string by simply calling `sd_bus_message_read_basic(message, 's', &attributeName)` which retrieves an attribute of type *string* from the message, lets attribute name point on to the string and move the current position in the message to the next attribute. Now that the attribute key was read, the client can read the attribute value stored in the variant.

The variant can store attributes of any single complete type but I will only read the attribute types of *'u'* for *uni32_t* and the attribute type of *'s'* for *string* which are used by the attributes *MainPID* and *ActiveState* respectively.

To read the correct data type the client has to first decide which type the variant holds. It does that by calling `sd_bus_message_peek_type(message, nullptr, &contents)` which stores the type description string of the attribute at the current position of the message in the pointer *contents*. With this information the client calls `sd_bus_message_enter_container(message, SD_BUS_TYPE_VARIANT, contents)` to enter the variant with the type description retrieved previously.

It then uses the type description string to determine if it has to read a *string* or a *uint32_t* based on the type string an reads the value of the appropriate type by calling `sd_bus_message_read_basic(message, contents[0], &unsignedIntValue)` or `sd_bus_message_read_basic(message, contents[0], &stringAttribute)`. It then exits the variant by calling `sd_bus_message_exit_container(message)` and the continues with the next element in the array.

Starting the client and then stopping the unit **python-sample-1.service** by calling `sudo systemctl stop python-sample-1` yields the following output.

```
Read attribute with name MainPID with type u
Attribute value 15412
Received signal for interface org.freedesktop.systemd1.Service
Read attribute with name ActiveState with type s
Attribute value deactivating
Received signal for interface org.freedesktop.systemd1.Unit
Read attribute with name MainPID with type u
Attribute value 0
Received signal for interface org.freedesktop.systemd1.Service
Read attribute with name ActiveState with type s
Attribute value inactive
Received signal for interface org.freedesktop.systemd1.Unit
```

9. Logging in Systemd

Systemd provides logging services for all applications managed by systemd. The logging implementation in the applications itself can be very simple, systemd just redirects the output on **stdout** and **stderr** into the systemd journal. This allows the use of nearly every logging framework as long as it allows logging to the standard output. Systemd also defines log level which can be used by adding prefixes to the output on the console. I will describe information about logging for the applications perspective in the following sections.

The systemd journal is stored in a binary format. Even though systemd can be configured to log into text files, the binary format of systemd allows to access and search the log based on powerful command line tools and it is suggested to use the tools to access the binary log instead. I will describe how to access the binary log in the following sections.

9.1. Accessing the Journal

Systemd provides the command **journalctl** to access the systemd journal from the console. The command is described in the man pages [64]. **Journalctl** is not the only command that can be used to access the systemd journal. Systemd also provides the **sd-journal** [65] api which will be described in a later section. In this section I will focus on the command line.

Calling `journalctl` without any arguments display the current log content. On my macbook with Ubuntu 18.04 it displays all messages in the log since I installed the operation system (which is only about a week ago at the time of writing). Using **journalctl** in this way is obviously not a good option because reading all log messages for the last several days takes a long time. Fortunately, **journalctl** provides various ways to filter the journal using command line options.

9.1.1. Showing the Log for specific Units only

Journalctl allows to filter the output so that the journal only displays log messages for a specific service unit. Calling `journalctl -u python-sample-1` will display only the log messages for the sample service I wrote for one of the earlier exercises. Because I did not implement any output to the command line, the output will only contain messages systemd add by default to any service it controls. On my system calling the command yields the following.

```
-- Logs begin at Wed 2018-10-03 01:34:34 EDT, end at Tue 2018-10-09 22:50:31 EDT. --
Oct 08 21:02:54 wotan systemd[1]: Started The sample service.
Oct 08 22:14:53 wotan systemd[1]: Stopping The sample service...
Oct 08 22:14:53 wotan systemd[1]: Stopped The sample service.
Oct 08 23:45:02 wotan systemd[1]: Started The sample service.
Oct 09 13:15:35 wotan systemd[1]: Stopping The sample service...
Oct 09 13:15:35 wotan systemd[1]: Stopped The sample service.
...
```

The output only contains log entries for starting and stopping the service created by systemd. These entries are automatically added by systemd when it starts or stops a service unit. It is easy to add new messages to the systemd log as I will demonstrate in one of the following sections.

[64] https://www.freedesktop.org/software/systemd/man/journalctl.html
[65] https://www.freedesktop.org/software/systemd/man/sd-journal.html

9.1.2. Filtering by Priority

Systemd uses the **syslog** message priorities or log level to indicate the message priority. The message priority or log levels are defined in the manual page for **syslog** [66] and can have the following values.

- **Emergency / LOG_EMERG / 0**

  Indicates that the system is unusable.

- **Alert / LOG_ALERT / 1**

  Indicates that an action has to be taken immediately.

- **Critical / LOG_CRIT / 2**

  Indicates a critical condition.

- **Error / LOG_ERR / 3**

  Indicates an error condition.

- **Warning / LOG_WARN / 4**

  Indicates a warning.

- **Notice / LOG_NOTICE / 5**

  Indicates a normal, but significant, condition.

- **Info / LOG_INFO / 6**

  Indicates an informational message.

- **Debug / LOG_DEBUG / 7**

  Indicates a debug level message.

Journalctl provides the command line option *-p* or *–priority* to filter the log by log levels or log level ranges. Calling the command with one log level, e.g. like `journalctl -p 2` will display all messages of log level 2 or lower. On my system that produces an output similar to the following.

```
-- Logs begin at Wed 2018-10-03 01:34:34 EDT, end at Tue 2018-10-09 23:52:52 EDT. --
Oct 04 20:01:03 wotan kernel: CPU3: Core temperature above threshold, cpu clock throttled (total events
Oct 04 20:01:03 wotan kernel: CPU1: Core temperature above threshold, cpu clock throttled (total events
Oct 04 20:01:03 wotan kernel: CPU1: Package temperature above threshold, cpu clock throttled (total ever
Oct 04 20:01:03 wotan kernel: CPU3: Package temperature above threshold, cpu clock throttled (total ever
Oct 04 20:01:03 wotan kernel: CPU2: Package temperature above threshold, cpu clock throttled (total ever
Oct 04 20:01:03 wotan kernel: CPU0: Package temperature above threshold, cpu clock throttled (total ever
...
```

Which shows me that my CPU gets to hot, but that is something to be investigated another time.

9.1.3. Filtering by Boot Id

Journalctl can filter the logs based on boot. Calling `journalctl --list-boots` lists all available boots. They are listed from the oldest to the newest boot, the latest boot at the bottom of the list. **Journalctl** displays for columns of information for each boot. The first column shows the position relative to the current boot. The current boot has the position 0, the boot before that -1, and so on. The second column contains the boot id, the third column shows the start date and time of the boot and the fourth column shows the end date of the boot. Start and end date are determined by the first record in the journal with the given boot id and the end date is the date of the last record in the journal with the boot id.

[66] http://man7.org/linux/man-pages/man3/syslog.3.html

Calling `journalctl --list-boots` yields the following output in my system.

```
-2 3313042ae14141989656c07cacbbeca2 Wed 2018-10-03 01:34:34 EDT-Wed 2018-10-03 11:31:44 EDT
-1 f67151aff1d942f3886acf0a953b9410 Wed 2018-10-03 11:32:41 EDT-Wed 2018-10-10 13:20:19 EDT
 0 f004794bfabd4cd2bf627556318e35b4 Wed 2018-10-10 13:21:15 EDT-Wed 2018-10-10 16:02:11 EDT
```

The relative boot position or the boot id can both be used to only display the records of a specific boot. The main difference is that the boot id is stable while the relative position for a given boot changes in between boots. So if I want to see the log messages for the boot with id *3313042ae14141989656c07cacbbeca2* I could either call `journalctl -b -2` or `journalctl -b 3313042ae14141989656c07cacbbeca2`.

9.1.4. Filtering by Journal Field

Systemd stores the journal in a binary file that constains fields. The systemd journal defines special journal fields in the special journal fields man page [67]. Applications can define their own fields. Some special fields should have the same meaning for each applications.

Journalctl provides the option to filter log entries by matches which reference fields and the values they should match. With matches it is possible to, for example, filter the log based on the systemd unit. Calling `journalctl _UID=1000` returns all log messages in relation to user 1000 which is myself on my system. The output is similar to the following.

```
-- Logs begin at Wed 2018-10-03 01:34:34 EDT, end at Wed 2018-10-10 00:48:53 EDT. --
Oct 03 01:34:48 wotan systemd[1820]: Starting D-Bus User Message Bus Socket.
Oct 03 01:34:48 wotan systemd[1820]: Listening on GnuPG cryptographic agent and passphrase cache.
Oct 03 01:34:48 wotan systemd[1820]: Reached target Timers.
Oct 03 01:34:48 wotan systemd[1820]: Listening on GnuPG cryptographic agent and passphrase cache (acces
Oct 03 01:34:48 wotan systemd[1820]: Listening on GnuPG cryptographic agent (ssh-agent emulation).
Oct 03 01:34:48 wotan systemd[1820]: Started Pending report trigger for Ubuntu Report.
Oct 03 01:34:48 wotan systemd[1820]: Reached target Paths.
Oct 03 01:34:48 wotan systemd[1820]: Listening on GnuPG network certificate management daemon.
...
```

Journalctl supports matches for all fields in the journal.

9.2. Implementing Logging

Systemd provides several ways to add messages to the log. In the following sections I will describe two of them, using the standard output of the process and using the **sd-journal** api.

9.2.1. Writing to the System Log using the Standard Output

The easiest way to write into the systemd journal with a unit is to write to the standard output or the error output. The following application demonstrates how to write something into the systemd journal.

```
#include <iostream>


using namespace std;

int main(int argc, char *argv[]) {
    cout << "Simple log message" << "\n";
```

[67] https://www.freedesktop.org/software/systemd/man/systemd.journal-fields.html

```
    cerr << "Simple error message" << "\n";
}
```

I use the following unit file to start the application as service unit. Note that the type of the application is *oneshot* in this case because the application runs and then exists.

```
[Unit]
Description=Example logging to systemd journal

[Service]
Type=oneshot
ExecStart=/usr/local/bin/logging-in-systemd-1
```

Stating the service unit using `systemctl start logging-in-systemd-1.service` and displaying the log using `journalctl -u logging-in-systemd-1.service` yields the following output.

```
-- Logs begin at Wed 2018-10-03 01:34:34 EDT, end at Wed 2018-10-10 16:48:25 EDT. --
Oct 10 16:44:58 wotan systemd[1]: Starting Example logging to systemd journal...
Oct 10 16:44:58 wotan logging-in-systemd-1[12834]: Simple log message
Oct 10 16:44:58 wotan logging-in-systemd-1[12834]: Simple error message
Oct 10 16:44:58 wotan systemd[1]: Started Example logging to systemd journal.
```

The output contains the two lines of text we printed on standard out and standard error. The journal added several standard fields to the two lines of text. I can display the additional fields by calling `journalctl -u logging-in-systemd-1.service -o verbose` which yields the following output on my machine. I removed all the other log entries except for the two that the application was written to the log.

```
...
Wed 2018-10-10 16:44:58.404196 EDT [s=818e0037db4b41fa904a12e6a823364b;i=3502d;b=f004794bfabd4cd2bf6275
    _MACHINE_ID=fb9b6434c01c48a39c38d044a835c0e2
    _HOSTNAME=wotan
    _UID=0
    _GID=0
    _CAP_EFFECTIVE=3fffffffff
    _SELINUX_CONTEXT=unconfined
    PRIORITY=6
    SYSLOG_FACILITY=3
    _SYSTEMD_SLICE=system.slice
    _PID=12834
    _TRANSPORT=stdout
    _BOOT_ID=f004794bfabd4cd2bf627556318e35b4
    _STREAM_ID=f1384d0e20574acab2fdbf7a6d6ace81
    SYSLOG_IDENTIFIER=logging-in-systemd-1
    MESSAGE=Simple log message
    _COMM=logging-in-syst
    _SYSTEMD_CGROUP=/system.slice/logging-in-systemd-1.service
    _SYSTEMD_UNIT=logging-in-systemd-1.service
    _SYSTEMD_INVOCATION_ID=0403b4a956044c1ab221d0d210977239
Wed 2018-10-10 16:44:58.404196 EDT [s=818e0037db4b41fa904a12e6a823364b;i=3502e;b=f004794bfabd4cd2bf6275
    _MACHINE_ID=fb9b6434c01c48a39c38d044a835c0e2
    _HOSTNAME=wotan
    _UID=0
    _GID=0
    _CAP_EFFECTIVE=3fffffffff
    _SELINUX_CONTEXT=unconfined
    PRIORITY=6
    SYSLOG_FACILITY=3
```

```
    _SYSTEMD_SLICE=system.slice
    _PID=12834
    _TRANSPORT=stdout
    _BOOT_ID=f004794bfabd4cd2bf627556318e35b4
    _STREAM_ID=f1384d0e20574acab2fdbf7a6d6ace81
    SYSLOG_IDENTIFIER=logging-in-systemd-1
    _COMM=logging-in-syst
    _SYSTEMD_CGROUP=/system.slice/logging-in-systemd-1.service
    _SYSTEMD_UNIT=logging-in-systemd-1.service
    _SYSTEMD_INVOCATION_ID=0403b4a956044c1ab221d0d210977239
    MESSAGE=Simple error message
...
```

The log entries show that systemd add several fields in addition to the *MESSAGE* field which contains the original message. Systemd add the following fields automatically which are described in the man page for special journal fields [68]. All fields starting with an underscore '_' are trusted fields only added by systemd and cannot be changed by user code.

- *_MACHINE_ID*

  The machine id of the system. The machine id is taken from the file **/etc/machine-id**.

- *_HOSTNAME*

  The hostname of the system.

- *_UID*

  The user id of the use running the program.

- *_GID*

  The group id of the user running the program.

- *_CAP_EFFECTIVE*

  The effective linux capabilities of the process executing the application. Linux capabilities are described in the linux capabilities man page [69].

- *_SELINUX_CONTEXT*

  The SELinux security context.

- *PRIORITY*

  The priority or log level of the message.

- *SYSLOG_FACILITY*

  Syslog compatibility fields.

- *_SYSTEMD_SLICE*

  The systemd slice of the process.

- *_PID*

  The process id.

- *_TRANSPORT*

  Describes how the log entry was received by the journal. The following lists valid transports.

[68] https://www.freedesktop.org/software/systemd/man/systemd.journal-fields.html
[69] http://man7.org/linux/man-pages/man7/capabilities.7.html

  - *audit*

    Messages from the kernel audit subsystem.

  - *driver*

    Message generated internally.

  - *syslog*

    Message received via the local syslog socket.

  - *journal*

    Message received via the native journal protocol.

  - *stdout*

    Message received via the standard output or error output.

  - *kernel*

    Message received from the kernel.

- *_BOOT_ID*

  The current boot id.

- *_STREAM_ID*

  The stream id only applies to messages received from the standard output. The stream id is a randomized 128 bit integer that is created when the connection was first created. This allows to select log entries based on the stream they were received from.

- *SYSLOG_IDENTIFIER*

  Compatibility field for syslog.

- *_COMM*

  The name of the process.

- *_SYSTEMD_CGROUP*

  The systemd cgroup of the process.

- *_SYSTEMD_UNIT*

  The systemd unit.

- *_SYSTEMD_INVOCATION_ID*

  The invocation id for the runtime cycle the message was created in.

As mentioned before, the fields starting with an underscore '_' cannot be changed by the logging process and will always be added by systemd. The process should be able to change the priority though. If no priority is supplied systemd creates the log message with priority 6 or *Info*.

9.2.2. Writing to the System Log using the Standard Output with Priority

Fortunately systemd allows adding priorities to log message by simply adding a prefix to the message string. The prefix is of the form . The following code adds log levels to the application from the previous chapter.

```
#include <iostream>

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

```
using namespace std;

int main(int argc, char *argv[]) {
    cout << "<" << LOG_NOTICE << ">" << "Simple log message" << "\n";
    cerr << "<" << LOG_ERR << ">" <<"Simple error message" << "\n";
}
```

I changed the output to the standard error to add a prefix for log level *Error* and added a prefix for log level *Notice* to the output on standard output. The following service unit is used to start the test application.

```
[Unit]
Description=Example logging to systemd journal

[Service]
Type=oneshot
ExecStart=/usr/local/bin/logging-in-systemd-2
```

After the service is started using `sudo systemctl start logging-in-systemd-2.service`, displaying the journal using `journalctl -u logging-in-systemd-2.service -o verbose --output-fields=PRIORITY,MESSAGE` yields the following result. Note that the option *–output-fields=PRIORITY,MESSAGE* instructs **journalctl** to only display the fields I am interested in.

```
-- Logs begin at Wed 2018-10-03 01:34:34 EDT, end at Thu 2018-10-11 12:54:05 EDT. --
Thu 2018-10-11 12:54:05.061492 EDT [s=818e0037db4b41fa904a12e6a823364b;i=35be2;b=f004794bfabd4cd2bf6275
    PRIORITY=6
    MESSAGE=Starting Example logging to systemd journal...
Thu 2018-10-11 12:54:05.063716 EDT [s=818e0037db4b41fa904a12e6a823364b;i=35be3;b=f004794bfabd4cd2bf6275
    PRIORITY=5
    MESSAGE=Simple log message
Thu 2018-10-11 12:54:05.063716 EDT [s=818e0037db4b41fa904a12e6a823364b;i=35be4;b=f004794bfabd4cd2bf6275
    PRIORITY=3
    MESSAGE=Simple error message
Thu 2018-10-11 12:54:05.064108 EDT [s=818e0037db4b41fa904a12e6a823364b;i=35be5;b=f004794bfabd4cd2bf6275
    PRIORITY=6
    MESSAGE=Started Example logging to systemd journal.
```

The output shows that the priorities in the journal are updated.

9.2.3. Writing to the System Log using sd-journal

While it is certainly nice that systemd allows easy logging by using standard output in a program, it prevents us from using the more advanced features of the journal. To make the use of the journals improvements, e.g. by using additional fields, we have to use the **sd-journal** api to send log messages to the journal. To send log messages, **sd-journal** provides the function **sd_journal_send()** [70]. The function **sd_journal_send()** is used to add structured messages to the journal. It takes a list of format strings in the form *FIELD=value* and writes all the specified fields into the journal. The strings can include format parameters like *%s* and *%i*. If they include format parameter the function takes the values immediately after the format string as additional arguments.

In the following program I reimplemented the previous example to add the messages with the given priority to the journal. Additionally, I added a custom field named *EXAMPLE* with the value *systemd*.

```
#include <iostream>

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

[70] https://www.freedesktop.org/software/systemd/man/sd_journal_send.html

```
using namespace std;

int main(int argc, char *argv[]) {
    sd_journal_send(
            "MESSAGE=Simple log message",
            "PRIORITY=%i", LOG_NOTICE,
            "EXAMPLE=systemd",
            nullptr);

    sd_journal_send(
            "MESSAGE=Simple error message",
            "PRIORITY=%i", LOG_ERR,
            "EXAMPLE=systemd",
            nullptr);
}
```

The application is calling the journal directly, that means that we do not need to define a service unit to start the application. But because the application does not have a service file anymore, I will have to filter the log differently in order to find only the log messages I am interested in.

Systemd adds the name of the executable as value to the field *SYSLOG_IDENTIFIER*. I can display all the log messages in relation the my test application using the following command.

```
journalctl -t logging-in-systemd-3
```

This yields the following output which shows us that I started the application seven times so far.

```
-- Logs begin at Wed 2018-10-03 01:34:34 EDT, end at Thu 2018-10-11 18:32:30 EDT. --
Oct 11 15:03:39 wotan logging-in-systemd-3[24038]: Simple log message
Oct 11 15:03:39 wotan logging-in-systemd-3[24038]: Simple error message
Oct 11 17:20:47 wotan logging-in-systemd-3[24631]: Simple log message
Oct 11 17:20:47 wotan logging-in-systemd-3[24631]: Simple error message
Oct 11 17:34:41 wotan logging-in-systemd-3[24898]: Simple log message
Oct 11 17:34:41 wotan logging-in-systemd-3[24898]: Simple error message
Oct 11 17:41:43 wotan logging-in-systemd-3[25068]: Simple log message
Oct 11 17:41:43 wotan logging-in-systemd-3[25068]: Simple error message
Oct 11 17:44:26 wotan logging-in-systemd-3[25127]: Simple log message
Oct 11 17:44:26 wotan logging-in-systemd-3[25127]: Simple error message
Oct 11 18:26:39 wotan logging-in-systemd-3[25387]: Simple log message
Oct 11 18:26:39 wotan logging-in-systemd-3[25387]: Simple error message
Oct 11 18:31:07 wotan logging-in-systemd-3[25436]: Simple log message
Oct 11 18:31:07 wotan logging-in-systemd-3[25436]: Simple error message
```

Calling **journalctl** like this `journalctl -t logging-in-systemd-3 -o verbose --output-fields=EXAMPLE,MESSAGE` will show the new field we added to all of the log entries and the message. I shortened the output below so that only two lines are shown.

```
...
Thu 2018-10-11 18:26:39.288403 EDT [s=569e2e1dcd0747f7ad9c27b31e5a982a;i=35fd2;b=f004794bfabd4cd2bf6275
    MESSAGE=Simple log message
    EXAMPLE=systemd
Thu 2018-10-11 18:26:39.288441 EDT [s=569e2e1dcd0747f7ad9c27b31e5a982a;i=35fd3;b=f004794bfabd4cd2bf6275
    EXAMPLE=systemd
    MESSAGE=Simple error message
...
```

Because we added the field *EXAMPLE* to all of the log entries, we can also search for them by using the field with the **journalctl** command like this `journalctl EXAMPLE=systemd`, which yields the output below.

```
-- Logs begin at Wed 2018-10-03 01:34:34 EDT, end at Thu 2018-10-11 18:32:30 EDT. --
Oct 11 15:03:39 wotan logging-in-systemd-3[24038]: Simple log message
Oct 11 15:03:39 wotan logging-in-systemd-3[24038]: Simple error message
Oct 11 17:20:47 wotan logging-in-systemd-3[24631]: Simple log message
Oct 11 17:20:47 wotan logging-in-systemd-3[24631]: Simple error message
Oct 11 17:34:41 wotan logging-in-systemd-3[24898]: Simple log message
Oct 11 17:34:41 wotan logging-in-systemd-3[24898]: Simple error message
Oct 11 17:41:43 wotan logging-in-systemd-3[25068]: Simple log message
Oct 11 17:41:43 wotan logging-in-systemd-3[25068]: Simple error message
Oct 11 17:44:26 wotan logging-in-systemd-3[25127]: Simple log message
Oct 11 17:44:26 wotan logging-in-systemd-3[25127]: Simple error message
Oct 11 18:26:39 wotan logging-in-systemd-3[25387]: Simple log message
Oct 11 18:26:39 wotan logging-in-systemd-3[25387]: Simple error message
Oct 11 18:31:07 wotan logging-in-systemd-3[25436]: Simple log message
Oct 11 18:31:07 wotan logging-in-systemd-3[25436]: Simple error message
```

10. Journal Message Catalogs

Message catalogs store additional explanatory texts for messages in the log. To add a message catalog entry to a message, the message needs a message id and a catalog entry associated with that message id has to be created.

Message ids are 128 bit integer that can be randomly created using **journalctl**. To create a new message id I call `journalctl --new-id` which creates a new random 128 bit id. The 128 bit ids are described in the **sd-id-128** man page [71]. Calling **journalctl** on my system for this example yields the following output.

```
As string:
69fb66e0503d48df8b6a81981a30d3d7

As UUID:
69fb66e0-503d-48df-8b6a-81981a30d3d7

As man:sd-id128(3) macro:
#define MESSAGE_XYZ SD_ID128_MAKE(69,fb,66,e0,50,3d,48,df,8b,6a,81,98,1a,30,d3,d7)

As Python constant:
>>> import uuid
>>> MESSAGE_XYZ = uuid.UUID('69fb66e0503d48df8b6a81981a30d3d7')
```

The output can then be copied and added to the application's code. The following code creates a message with the *MESSAGE_ID* from above.

```
#include <iostream>

#include <systemd/sd-journal.h>
#include <systemd/sd-id128.h>


#define EXAMPLE_MESSAGE SD_ID128_MAKE(69,fb,66,e0,50,3d,48,df,8b,6a,81,98,1a,30,d3,d7)
```

[71] https://www.freedesktop.org/software/systemd/man/sd-id128.html

```
using namespace std;

int main(int argc, char *argv[]) {
    char messageId[33];
    sd_id128_to_string(EXAMPLE_MESSAGE, messageId);
    sd_journal_send(
            "MESSAGE_ID=%s", messageId,
            "MESSAGE=Message with the example message id",
            "PRIORITY=%i", LOG_ERR,
            nullptr);
}
```

We can now use the field *MESSAGE_ID* to find messages in the journal. To use the *MESSAGE_ID* to its fullest, I have to create a custom message catalog for the example application. Message catalogs are described in files ending with the extension *.catalog*. The format is described in the journal message catalogs documentation [72]. The catalog file I created for the example is shown below.

```
-- 69fb66e0503d48df8b6a81981a30d3d7
Subject: The message was created to demonstrate the message catalog
Defined-By: Systemd by Example
Support: https://gitlab.com/franks_reich/systemd-by-example

This is a potentially longer text to describe the message even better.
```

A catalog entry for a message is started with – . The message id has to be formatted as a hex string. The message id can also be followed by a locale as *de* to enable the translation of catalog entries for messages. The message id is followed by a list of headers with the following semantics:

- **Subject**

  A short one line description of the message.

- **Defined-By**

  Describes who defined the message. That is typically the name of the application or package.

- **Support**

  A uri that helps in getting support, e.g. a git repository or something like that.

- **Documentation**

  A uri to the documentation.

If the entry for the message should also contain an entry payload, a multi-line message for human consumption, it should follow the headers after a new-line.

I copy the file to **/usr/lib/systemd/catalog** and the request **journalctl** to update the binary index of the catalog by calling `journalctl --update-catalog`. After **journalctl** updated the catalog, the catalog now shows up in the catalog list after calling `journalctl --list-catalog`.

```
...
5eb03494b6584870a536b337290809b3 systemd: Automatic restarting of a unit has been scheduled
641257651c1b4ec9a8624d7a40a9e1e7 systemd: Process @EXECUTABLE@ could not be executed
69fb66e0503d48df8b6a81981a30d3d7 Systemd by Example: The message was created to demonstrate the message
6bbd95ee977941e497c48be27c254128 systemd: System sleep state @SLEEP@ entered
7b05ebc668384222baa8881179cfda54 systemd: Unit @UNIT@ has finished reloading its configuration
...
```

[72] https://www.freedesktop.org/wiki/Software/systemd/catalog/

After I execute the example application from above, adding a message with our message id to the journal, I can call **journalctl** with the option *-x* to display the information of the catalog in addition to the message itself. Calling **journalctl** using `journalctl MESSAGE_ID=69fb66e0503d48df8b6a81981a30d3d7 -x` yields the following output.

```
-- Logs begin at Wed 2018-10-03 01:34:34 EDT, end at Thu 2018-10-11 21:32:47 EDT. --
Oct 11 21:18:45 wotan message_catalog_1[29584]: Message with the example message id
-- Subject: The message was created to demonstrate the message catalog
-- Defined-By: Systemd by Example
-- Support: https://gitlab.com/franks_reich/systemd-by-example
--
-- This is a potentially longer text to describe the message even better.
Oct 11 21:19:07 wotan message_catalog_1[29658]: Message with the example message id
-- Subject: The message was created to demonstrate the message catalog
-- Defined-By: Systemd by Example
-- Support: https://gitlab.com/franks_reich/systemd-by-example
--
-- This is a potentially longer text to describe the message even better.
```

We can see that I called the application that logs with the new message id twice. Additionally, **journalctl** added the *Subject*, the *Defined-By* and the *Support* header from the catalog to the output.

10.1. Using Message Variables in the Catalog

When **journalctl** processes messages, it will exchange all message variables in the message with the respective field in the journal entry. Variables are created by putting the field name enclosed between *@* in the catalog entry of the message, for example including* @MESSAGE_ID@* would show the message id at this position in the message.

```
-- ddc85d3a25204e41a38fb1d9b4ce2a8f
Subject: Another sample message for the systemd by examples book
Defined-By: Systemd by Example
Support:  https://gitlab.com/franks_reich/systemd-by-example

This is the example message created for message id @MESSAGE_ID@, it
demonstrates how variables are used and replaced in the catalog entry.
The message was created on host @_HOSTNAME@ by the user with the user
id @_UID@.
```

The message shown above includes three variables, the *MESSAGE_ID*, the *_HOSTNAME* and the *_UID*. I implemented the following program to add a message with that message id to the journal.

```
#include <iostream>

#include <systemd/sd-journal.h>
#include <systemd/sd-id128.h>


#define MESSAGE_WITH_VARIABLES SD_ID128_MAKE(dd,c8,5d,3a,25,20,4e,41,a3,8f,b1,d9,b4,ce,2a,8f)

using namespace std;

int main(int argc, char *argv[]) {
    char messageId[33];
    sd_id128_to_string(MESSAGE_WITH_VARIABLES, messageId);
    sd_journal_send(
```

```
            "MESSAGE_ID=%s", messageId,
            "MESSAGE=Message with the example message id",
            "PRIORITY=%i", LOG_ERR,
            nullptr);
}
```

After calling the application and copying and rebuilding the binary index for the message catalogs, I can display the messages for the new message id using `journalctl MESSAGE_ID=ddc85d3a25204e41a38fb1d9b4ce2a8f -x`. This produces the following output.

```
Oct 11 22:01:39 wotan message_catalog_2[30563]: Message with the example message id
-- Subject: Another sample message for the systemd by examples book
-- Defined-By: Systemd by Example
-- Support:  https://gitlab.com/franks_reich/systemd-by-example
--
-- This is the example message created for message id ddc85d3a25204e41a38fb1d9b4ce2a8f, it
-- demonstrates how variables are used and replaced in the catalog entry.
-- The message was created on host wotan by the user with the user
-- id 1000.
```

Journalctl replaced all the variables in the message with the respective fields.

11. Appendix

11.1. Using Systemd with CMake

All the C/C++ examples in this book were build with **cmake** [73]. Cmake is controlled with **CMakeLists.txt** files. I discuss an example of one of the **CMakeLists.txt** files in the following and show how systemd can be used as a library in cmake.

Systemd provides pkg-config files for the pkg-config tool [74]. This allows retrieving the relevant paths and command line options for compiling applications using **libsystemd** being retrieved by pkg-config. I can for example call `pkg-config --libs libsystemd` and pkg-config returns the libraries that have to be linked by the linker.

Cmake has a plugin that allows **cmake** to use pkg-config to find dependencies for application compiled with **cmake**. The following **CMakeLists.txt** includes an example of how to use the pkg-config plugin to build an application using **libsystemd** with **cmake**.

```
find_package(PkgConfig)

pkg_check_modules(
        Systemd REQUIRED
        IMPORTED_TARGET
        libsystemd)

add_executable(d-bus-activation-example
        d-bus-activation-example.cpp)

target_link_libraries(d-bus-activation-example
        PkgConfig::Systemd)
```

[73] https://cmake.org/
[74] https://www.freedesktop.org/wiki/Software/pkg-config/

First, the package providing the pkg-config plugin has to be loaded using `find_package(PkgConfig)`. The module provides the function **pkg_check_modules()** which uses pkg-config to search for the given module. The script does this by calling `pkg_check_modules(Systemd REQUIRED IMPORTED_TARGET libsystemd)`. The script exports the library target **PkgConfig::Systemd** which can be added to executables and libraries using the function **target_link_libraries()**. After exporting the systemd module it creates an executable by calling the function **add_executable()** and finally adds the imported target **PkgConfig::Systemd** to the executable by calling **target_link_libraries()**.

11.2. Resources and References