Introduction
This tutorial is about the new submodule in Solid for handling basic partitioning operations. Althought each public method and class is documented in a doxygen style, this tutorial will guide you through the process of writing a partitioning application that uses the new available API.
All the classes in this module are inside the namespace Solid::Partitioner: other subnamespaces are used, and will be pointed out in the tutorial when necessary.

The VolumeManager class
Virtually everything in this submodule is indirectly managed by the VolumeManager singleton. Most of your interaction with the Solid will pass through this class.

Once instantiated, VolumeManager will detect all the information about drives and partitions, thus making them immediately available to you.

Getting information about devices.
The first thing any partitioning application needs to do is to display updated information about the drives and partitions present on the system.

This information is organized as a tree map: each tree's root node contains a disk, while its children are the partitions and free space blocks, sorted by initial offset (that is, their position inside the disk itself). Logical partitions, and free space blocks inside them, are inserted as children of the extended partition. A tree is often called a layout tree in the documentation. VolumeManager offers two methods:

VolumeTree diskTree(const QString& diskName); this returns the layout tree of the given disk.
VolumeTreeMap allDiskTrees(); this returns a map containing all the disk layouts. The map is indexed by disk name.

See the documentation for the VolumeTree to know all the methods available. Note that a VolumeTree object may be invalid (for example, if you specified an unexistent disk in the diskName parameter above); to be on the safe side, it's suggested you query the valid() method beforehand, to avoid asserts failing later. The VolumeTree class is implicitly shared, so copies are cheap; however, it's possible to obtain a hard copy of the disk and its contents with copy().

Each node in the tree is represented by a VolumeTreeItem object; however, these objects are rarely interesting for the application, as they only contain pointers to the node's parent and children. The actual payload is the DeviceModified object stored inside each item. DeviceModified is an abstract base class used to represent a device in the partitioning submodule. Tree classes inherits DeviceModified: Disk, Partition and FreeSpace. All these classes are contained in the Devices subnamespace; the pure virtual method deviceType() can be used to get the most specific type of an object.

Each of this class maintains several properties. Althought almost all setters are currently public for simplicity, calling them from your application is likely to result in nasty bugs or segmentation faults very soon. Among all the properties, the name property is particularly interesting, as it uniquely identifies a device inside the system. To maintain uniqueness, some names aren't very human-readable: on your interface you can use the description property instead.
Please note that not all DeviceModified objects represent a device you actually have in

your system: for example, a Partition object may be an actual partition, or a partition for which a creation request has been sent to Solid, but not confirmed yet. In that case, the property name holds a dummy name generated by Solid; this name is, like every other, a suitable identifier for the partition.

A note about partition labels
Unfortunately udisks allows some ambiguity when dealing with partition's labels. To avoid confusion, it was maintained in Solid, althought we hope to improve it later.
MBR tables don't support labeled partitions: if you call partition->label(), you will get the filesystem's label, if there is one. On the other hand, GPT tables support labels for partitions, so partition->label() will return the partition label.

There is no property we can call to know the filesystem label in GPT, so it will always be empty: this will be fixed in the future, using filesystem-specific tools.

In the next paragraph you will be introduced to an action that modifies a filesystem's label: if the table currently used is MBR, that action will also change the partition's label. To sum it up, if you want to display the labels in your application, you can use the label property of Partition regardless of which table is currently installed on the disk.

Registering actions
A typical execution of a partitioner involves two steps:

- requesting changes;
- confirming them.

When you first request the changes, you can see them on the widget that shows you the disks' layout, but nothing actually happens in your hardware until you give a confirmation (or press exit and forget about everything).

This partitioning library has been organized following exactly this philosophy. In this paragraph we will talk about actions, and how to register them. In the next, we will review some central points about execution.

Actions are registered to the system using the registerAction() method of VolumeManager. The only parameter is a pointer to the action. The destruction of this object is entirely managed by Solid: actions are destroyed when the library has no more use for them. Therefore, deallocating this objects in the application is likely to leave to dangling pointers in the library, and should be avoided.
Before modifying the layout trees, Solid checks if the action is correct: if false is returned, the action wasn't either applied nor registered. To get more detailed information about what issue arised, call error(). The PartitioningError object returned will give you an error code and a readable description.
When true is returned, everything went right. This means some tree layout has been modified to reflect the new changes you registered, but nothing actually happened on the system.
Furthermore, you can review the list of registered actions at any time, using registeredActions().

All actions are contained in the subnamespace Actions. Let's cover briefly each one:

Action
This is the abstract base class. The pure virtual method description() of this class returns

a string with a readable description of the action. To know the most specific type of an Action object use actionType() and the ActionType enumeration.

FormatPartitionAction
Changes the filesystem of a partition. You can set the new filesystem either with the class Utils::Filesystem, or with a simple QString containing its name. The Filesystem class allows you to set a filesystem label, when supported, and some flags like ownership.

You can get a list of supported filesystem names, as well as their properties, using the FilesystemUtils singleton in the Utils subnamespace. You should always use the names returned by supportedFilesystems(): the support for more names is in progress. There is also a method filesystemProperty(), to obtain the value of a particular property as a QVariant. Read to the udisks documentation for the KnownFilesystem property, or for the struct Filesystem defined in solid/backends/udisks/udisksfilesystem.h, to see the available properties. The property name is also the string you have to pass as a second parameter in the method.

As you can imagine, the first parameter is the partition to format. Important note, valid for all device-related parameters: always use the device UDI. A device UDI will be something like /org/freedesktop/UDisks/devices/sda. Different names will be rejected by the manager.

ModifyFilesystemAction
This action allows you to change a filesystem's label. If you use FormatPartitionAction for this, Solid will recreate the filesystem from scratch, erasing all the data. Use this if you just want to change the label.

ModifyPartitionAction
This action allows you to change the label and flags of a partition.

RemovePartitionAction.
This action is used to remove a partition.

ResizePartitionAction.
This action can be actually used to resize and/or move a partition: you can change both the offset and the size at the same time. Keep in mind that when you change only the offset, you're moving the partition backwards and forwards, while the size stays the same. The new offset and size must be specified in bytes.
Unfortunately for now resizing or moving a partition means deleting the filesystem contained, if any, so all the data are lost. This will change as soon as possible.

CreatePartitionAction.
Create a partition. You can set several properties, like the filesystem or label, directly when creating. Again the geometry properties, offset and size, must be specified in bytes.

CreatePartitionTableAction and RemovePartitionTableAction.
These actions are used to manage partition tables. Currently two partition table schemes are supported: MBR and GPT. The second action erases all table signatures from the disk. Keep in mind that with either actions all partitions (and so all the data) are lost. Transitioning from one scheme to the other keeping the current layout isn't

supported right now.
To know some information and properties of a partition table scheme, see the documentation of the Utils::PartitionTableUtils singleton. For example, you can obtain a list of supported flags for both schemes.

Conflicting actions
By default, conflicting actions are detected and erased from the list of registered actions (while all their changes are maintained). Two actions are called conflicting when their combined effect on the layout, in the given order, is void. At the moment conflicting actions are detected only when they immediately follow one another in the registration process. This is because when separated by other actions, that change the situation, those could not be conflicting anymore. You can disable or enable this behavior while the setDeletionOfConflictingActions method in VolumeManager.

Undo and redo
Undoing and redoing actions is supported by Solid using the undo() and redo() methods. These functions are always safe to call, because they don't do anything when there isn't any action to undo/redo. However, you can call isUndoPossible() and isRedoPossible() in advance to know which operations are available. The list of registered actions is changed accordingly.

Execution
The apply() method applies all the registered actions on disk, returning a boolean. In the case of a failure, the PartitioningError object can be obtained again to know the cause. It is impossible, for an application linked with Solid, to successfully call apply() while another application has already done so. If you call it, it will fail immediately with the error code BusyExecuterError. This prevents conflicts between applications, which can be nasty for your hardware.

After executing, all the hardware detection is repeated, so the layout is updated. At this point there is the possibility that you will see something a bit different from what you requested. This is because udisks makes some internal adjustments, for example it may align partitions to certain offsets for performance reasons.

The execution happens synchronously (althought the calls to udisks services are asynchronous), but nevertheless some signals are sent to report progress or errors. See the next section, which covers the subject of notifications.

After executing, all the registered actions are wiped automatically. This happens even if the execution wasn't successful (except when it fails because of the BusyExecuterError) because in the latter case we're not sure in what state we left the hardware, so some action could be out-of-context.

Notifications
VolumeManager employs some signals to report important notifications and status changes to the application.

<u>New devices/removed devices</u>
All changes to the part of the hardware that interests us are monitored. For example when a removable disk is added, detection is performed, and the new tree added to the map.
Solid also monitors changes coming from other partitioning applications. If another

application running on your system creates a partition, Solid catches this and updates the layout information. When the layout information of a disk changes in any way, particularly when the disk is removed from the system, all actions concerning that disk are deleted immediately. This avoids keeping actions that could have become incorrect in the new layout.
While Solid virtually manages everything in this scenarios, two signals are sent to the application to let it know that its layout and action data must be updated: deviceAdded and deviceRemoved.

Layout changes
Like we said before, registering actions applies changes on the layout trees. When a layout tree is changed as a result of a new action, the diskChanged signal is emitted.

Accessibility
Many volumes can be mounted or umounted. The accessibilityChanged signal is sent when the accessibility status of a volume changes, that is, after a partition is mounted or umounted.

Execution
Two signals are provided while applying actions: progressChanged and executionError. The first merely gives you the index of the last action that was executed successfully, starting from 0. The executionError signals gives you an error string to display. Althought this information can be retrieved using the apply() return value and error(), we kept this signal to follow the asynchronous nature of many applications.

Conclusions
If you have more questions regarding this module, feel free to contact me at <shainer@chakra-project.org>. Happy hacking!