Linux USB gadget configured through configfs

25th April 2013

Overview

A USB Linux Gadget is a device which has a UDC (USB Device Controller) and can be connected to a USB Host to extend it with additional functions like a serial port or a mass storage capability.

A gadget is seen by its host as a set of configurations, each of which contains a number of interfaces which, from the gadget's perspective, are known as functions, each function representing e.g. a serial connection or a SCSI disk.

Linux provides a number of functions for gadgets to use.

Creating a gadget means deciding what configurations there will be and which functions each configuration will provide.

Configfs (please see *Documentation/filesystems/configfs.rst*) lends itself nicely for the purpose of telling the kernel about the above mentioned decision. This document is about how to do it.

It also describes how configfs integration into gadget is designed.

Requirements

In order for this to work configfs must be available, so CONFIGFS_FS must be 'y' or 'm' in .config. As of this writing USB_LIBCOMPOSITE selects CONFIGFS_FS.

Usage

(The original post describing the first function made available through configfs can be seen here: http://www.spinics.net/lists/linux-usb/msg76388.html)

```
$ modprobe libcomposite
$ mount none $CONFIGFS_HOME -t configfs
```

where CONFIGFS_HOME is the mount point for configfs

1. Creating the gadgets

For each gadget to be created its corresponding directory must be created:

```
$ mkdir $CONFIGFS_HOME/usb_gadget/<gadget name>
```

e.g.:

```
$ mkdir $CONFIGFS_HOME/usb_gadget/g1

...
...
...

$ cd $CONFIGFS_HOME/usb_gadget/g1
```

Each gadget needs to have its vendor id <VID> and product id <PID> specified:

```
$ echo <VID> > idVendor
$ echo <PID> > idProduct
```

A gadget also needs its serial number, manufacturer and product strings. In order to have a place to store them, a strings subdirectory must be created for each language, e.g.:

```
$ mkdir strings/0x409
```

Then the strings can be specified:

```
$ echo <serial number> > strings/0x409/serialnumber
$ echo <manufacturer> > strings/0x409/manufacturer
$ echo <product> > strings/0x409/product
```

2. Creating the configurations

Each gadget will consist of a number of configurations, their corresponding directories must be created:

$ mkdir configs/<name>.<number>

where <name> can be any string which is legal in a filesystem and the <number> is the configuration's number, e.g.:

```
$ mkdir configs/c.1

...
...
...
```

Each configuration also needs its strings, so a subdirectory must be created for each language, e.g.:

```
$ mkdir configs/c.1/strings/0x409
```

Then the configuration string can be specified:

```
$ echo <configuration> > configs/c.1/strings/0x409/configuration
```

Some attributes can also be set for a configuration, e.g.:

```
$ echo 120 > configs/c.1/MaxPower
```

3. Creating the functions

The gadget will provide some functions, for each function its corresponding directory must be created:

```
$ mkdir functions/<name>.<instance name>
```

where <name> corresponds to one of allowed function names and instance name is an arbitrary string allowed in a filesystem, e.g.:

```
$ mkdir functions/ncm.usb0 # usb_f_ncm.ko gets loaded with request_module()

...
...
...
```

Each function provides its specific set of attributes, with either read-only or read-write access. Where applicable they need to be written to as appropriate. Please refer to Documentation/ABI/testing/configfs-usb-gadget for more information.

4. Associating the functions with their configurations

At this moment a number of gadgets is created, each of which has a number of configurations specified and a number of functions available. What remains is specifying which function is available in which configuration (the same function can be used in multiple configurations). This is achieved with creating symbolic links:

```
$ ln -s functions/<name>.<instance name> configs/<name>.<number>
```

e.g.:

```
$ ln -s functions/ncm.usb0 configs/c.1

...
...
...
```

5. Enabling the gadget

All the above steps serve the purpose of composing the gadget of configurations and functions.

An example directory structure might look like this:

```
.
./strings
./strings/0x409
./strings/0x409/serialnumber
./strings/0x409/product
./strings/0x409/manufacturer
./configs
./configs/c.1
./configs/c.1/ncm.usb0 -> ../../../../usb_gadget/g1/functions/ncm.usb0
./configs/c.1/strings
./configs/c.1/strings/0x409
./configs/c.1/strings/0x409/configuration
./configs/c.1/bmAttributes
./configs/c.1/MaxPower
./functions
./functions/ncm.usb0
./functions/ncm.usb0/ifname
./functions/ncm.usb0/qmult
./functions/ncm.usb0/host_addr
./functions/ncm.usb0/dev_addr
./UDC
./bcdUSB
./bcdDevice
./idProduct
```

```
./idVendor
./bMaxPacketSize0
./bDeviceProtocol
./bDeviceSubClass
./bDeviceClass
```

Such a gadget must be finally enabled so that the USB host can enumerate it.

In order to enable the gadget it must be bound to a UDC (USB Device Controller):

```
$ echo <udc name> > UDC
```

where <udc name> is one of those found in /sys/class/udc/* e.g.:

```
$ echo s3c-hsotg > UDC
```

6. Disabling the gadget

```
$ echo "" > UDC
```

7. Cleaning up

Remove functions from configurations:

```
$ rm configs/<config name>.<number>/<function>
```

where <config name>.<number> specify the configuration and <function> is a symlink to a function being removed from the configuration, e.g.:

```
$ rm configs/c.1/ncm.usb0

...
...
...
```

Remove strings directories in configurations:

$ rmdir configs/<config name>.<number>/strings/<lang>

e.g.:

```
$ rmdir configs/c.1/strings/0x409

...
...
...
```

and remove the configurations:

```
$ rmdir configs/<config name>.<number>
```

e.g.:

```
rmdir configs/c.1

...
...
...
```

Remove functions (function modules are not unloaded, though):

$ rmdir functions/<name>.<instance name>

e.g.:

```
$ rmdir functions/ncm.usb0

...
...
...
```

Remove strings directories in the gadget:

```
$ rmdir strings/<lang>
```

e.g.:

```
$ rmdir strings/0x409
```

and finally remove the gadget:

```
$ cd ..
$ rmdir <gadget name>
```

e.g.:

```
$ rmdir g1
```

Implementation design

Below the idea of how configfs works is presented. In configfs there are items and groups, both represented as directories. The difference between an item and a group is that a group can contain other groups. In the picture below only an item is shown. Both items and groups can have attributes, which are represented as files. The user can create and remove directories, but cannot remove files, which can be read-only or read-write, depending on what they represent.

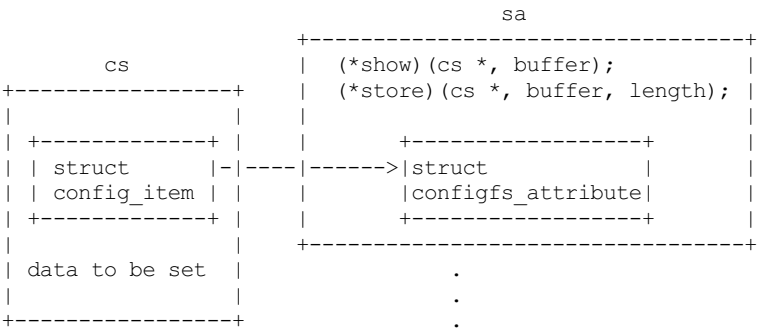
The filesystem part of configfs operates on config_items/groups and configfs_attributes which are generic and of the same type for all configured elements. However, they are embedded in usage-specific larger structures. In the picture below there is a "cs" which contains a config_item and an "sa" which contains a configfs_attribute.

The filesystem view would be like this:

```
./
./cs        (directory)
   |
   +--sa    (file)
   |
   .
   .
   .
```

Whenever a user reads/writes the "sa" file, a function is called which accepts a struct config_item and a struct configfs_attribute. In the said function the "cs" and "sa" are retrieved using the well known container_of technique and an appropriate sa's function (show or store) is called and passed the "cs" and a character buffer. The "show" is for displaying the file's contents (copy data from the cs to the buffer), while the "store" is for modifying the file's contents (copy data from the buffer to the cs), but it is up to the implementer of the two functions to decide what they actually do.

```
typedef struct configured_structure cs;
typedef struct specific_attribute sa;
```



The file names are decided by the config item/group designer, while the directories in general can be named at will. A group can have a number of its default sub-groups created automatically.

For more information on configfs please see *Documentation/filesystems/configfs.rst*.

The concepts described above translate to USB gadgets like this:

1. A gadget has its config group, which has some attributes (idVendor, idProduct etc) and default sub-groups (configs, functions, strings). Writing to the attributes causes the information to be stored in appropriate locations. In the configs, functions and strings sub-groups a user can create their sub-groups to represent configurations, functions, and groups of strings in a given language.

2. The user creates configurations and functions, in the configurations creates symbolic links to functions. This information is used when the gadget's UDC attribute is written to, which means binding the gadget to the UDC. The code in drivers/usb/gadget/configfs.c iterates over all configurations, and in each configuration it iterates over all functions and binds them. This way the whole gadget is bound.

3. The file drivers/usb/gadget/configfs.c contains code for

   - gadget's config_group
   - gadget's default groups (configs, functions, strings)
   - associating functions with configurations (symlinks)

4. Each USB function naturally has its own view of what it wants configured, so config_groups for particular functions are defined in the functions implementation files drivers/usb/gadget/f_*.c.

5. Function's code is written in such a way that it uses

usb_get_function_instance(), which, in turn, calls request_module. So, provided that modprobe works, modules for particular functions are loaded automatically. Please note that the converse is not true: after a gadget is disabled and torn down, the modules remain loaded.