Libsysfs - a programming interface to gather device information in Linux

Ananth N. Mavinakayanahalli
Linux Technology Center
IBM India Software Lab
ananth@in.ibm.com

Daniel Stekloff
Linux Technology Center
IBM Beaverton
dsteklof@us.ibm.com

Abstract

The Linux® 2.6 kernels include a new driver model, a set of common data structures and operations abstracted from various subsystems. Sysfs, a new RAM-based file system included in the 2.6 kernels, takes advantage of the new driver model by exposing system device information to User Space. Sysfs is intended to be the device tree for Linux.

We present Libsysfs, which is a User Space library that provides applications with a programming interface to the sysfs file system and system device data. The library reduces the need for each application to know how the file system is arranged and managed, providing common calls to query device attributes and relationships. The API is also meant to be stable, providing an interface that will not change while sysfs may change underneath it.

Libsysfs currently obtains system device information based on *bus*, *class*, and *devices*. The three categories reflect sysfs's device representation. The *bus* subsystem under sysfs represents a bus view of system buses and their devices. *Class* refers to system device classes like *net*, *scsi_host*, and *usb*. *Devices* represents the hierarchical view of all system devices. The library provides functions for listing devices by these categories as well as reading and writing to device attributes, represented as sysfs files.

There are three applications currently being developed that use Libsysfs to query system device information. Greg Kroah-Hartman's udev - a User Space replacement for devfs, Linux Event Logging project's Error Log Analysis, and Linux Diagnostic Tools project's sysdiag diagnostic command-line interface.

This paper especially discusses:

1. Need for Libsysfs
2. The APIs associated with Libsysfs
3. Design of Libsysfs
4. Current state of Libsysfs.

1 Introduction

The 2.6 Linux kernel includes a new feature called sysfs, a virtual file system that exposes system device information and attributes to User Space. Libsysfs's purpose is to provide a stable and consistant programming interface to the sysfs file system and device information. The library removes applications's need for common file system knowledge and code while providing interfaces for retrieving device information based on bus or class. Libsysfs's interfaces will be stable for applications to depend upon while sysfs matures.

Section 2 provides a brief background about the driver model and sysfs. Section 3 outlines Libsysfs's purpose. Section 4 gives an overview of aspects of sysfs that had to be considered while designing the

library. It also details the conventions that were followed to extract information from sysfs and how this information is presented to users. Section 5 describes the API naming conventions. Section 6 describes the data structures used in the library to represent sysfs's directory, file and link. Section 7 describes the basic abstraction of a device in sysfs as well as its Libsysfs representation. Section 8 describes Libsysfs's representation of buses. It also details how the library provides interfaces to determine device/driver details from a bus perspective. Section 9 describes the class representation in Libsysfs and functions available to users to access class information. Section 10 describes how Libsysfs implements access to drivers. Section 11 gives examples for using the library. Section 12 talks about the future of the library.

Note that we assume */sys* as the sysfs mount point for subsequent discussions.

2 Background

This section provides a brief history about the new Linux Driver Model and sysfs.

2.1 Driver Model

A new driver model [01] was adopted early in the Linux 2.5 development cycle. Its purpose was to unify drivers, abstracting data common to all drivers into general device structures. The model defined common interfaces as well.

Developed by Patrick Mochel, the initial intention in moving towards the new driver model was to make the power management tasks easy. The main structures that constitute the new driver model (defined in *include/linux/device.h* in the linux source code tree) are:

- device
- device_driver
- bus_type
- class
- class_device

The device structure (which will be statically defined as part of the bigger, bus specific structure) consists of, among other things, the device name, a pointer to the device's parent, a pointer to the device's device_driver structure, and a list of the device's children. It also contains the device's power state.

The device_driver structure contains a list of devices that use this driver. It also contains references to probe, remove, shutdown, suspend and resume routines for devices that use this driver.

The bus_type structure is used to represent a specific bus type. The structure consists, among other things, of lists of devices and drivers that are registered with the bus. It also contains functions pointers to bus specific routines to match a device and its driver, adding a device in the hierarchy, hotplug notification and power management.

The class and class_device structures are used to classify devices based on functionality. The class structure consists of a list of class_devices that belong to the class and functions for hotplug and class_device release notifications. The class_device structure consists of pointers to the class it belongs to, the physical or logical device that implements the class_device.

Attributes for each of these subsystems can be exported to userspace through show/store methods.

The design of the new Linux driver model lends itself to usage in different scenarios as

1. Hotplug
2. Device hierarchy representation
3. Power management

Detailed information about the driver model is available elsewhere [02, 04].

2.2 Sysfs

With the advent of the new driver model, a method was necessary to debug it. This necessity culminated in the creation of a new RAM-based filesystem called *driverfs*, which was later renamed *sysfs*. Sysfs provides a number of views of system devices:

- Hierarchical: The topology tree of devices in a system.
- Bus specific: What devices are connected to what bus.
- Class based : Classification based on functionality.

Sysfs is compiled into all 2.5/2.6 kernels and can be mounted using:

```
mount -t sysfs sysfs /sys
```

Salient features of sysfs includes:

1. A hierarchical view of the complete device tree where the various components of the driver model are organized as directories, attributes as files, and interconnections as links.
2. A type-safe kernel interface.
3. A facility by which any layer can export its attributes.
4. A true representation of the various relationships that exist between devices, buses, and drivers in the system.

More details about sysfs can be found elsewhere [03, 05].

3 Goals of Libsysfs

Sysfs already has an interface, a very easy to use file system. Commands like `echo`, `cat`, `ls`, and `find` are already available as are functions like `open()`, `read()`, and `close()`. Why is Libsysfs needed?

3.1 Encapsulate Sysfs

The first goal of Libsysfs's is to encapsulate sysfs-specific knowledge required for retrieving device information, reducing the need for applications to know how sysfs is structured. Applications can use Libsysfs to retrieve a device and its attributes without knowing specifically how that information is organized in sysfs. Encapsulation enables users to concentrate on what they're intended to do - work with devices. Users can leave the file system knowledge and how sysfs organizes device information to Libsysfs.

A user could, for example, use Libsysfs to retrieve specific device information using the sysfs path to the device. The library internally collects information including parent, children, bus, and driver name and returns it all to the user. Without the library, the user would need to know where to retrieve that information, like being able to discern what directories below the device directory are child devices and where to find bus information.

3.2 Reduce Duplicate Code

The library's second goal, which is related to the first, is to reduce duplicate code needed by applications to work with the file system and to retrieve sysfs specific information. Libsysfs includes functions for reading and writing files and traversing directories. Libsysfs also, as mentioned in the example above, includes code to easily wrap up device information rather than requiring each application to duplicate it.

3.3 Provide a Stable API

Libsysfs's final goal is to provide a stable programming interface that applications can depend upon to work with system devices. Sysfs is still maturing; new features are being added with each kernel release. Its structure, outside perhaps the bus, class, and devices subsystems, isn't firm. Block devices, currently, are considered a separate subsystem and aren't under the class subsystem. Libsysfs gives applications a stable interface to depend upon while allowing sysfs to evolve.

Libsysfs is intended to be the API to use to access sysfs information.

4 Design of Libsysfs

4.1 Modeling sysfs

Libsysfs is modeled directly from sysfs. Sysfs contains three major subsystems: *bus*, *class*, and *devices*. The library represents these subsystems and their devices as objects like *sysfs_device*, *sysfs_bus* and *sysfs_class*. The library represents the relationships between the sysfs objects - linking, for example, a *sysfs_class_device* under the *class* subsystem to a *sysfs_device* under the *devices* subsystem. Libsysfs objects have attributes like their sysfs counterparts. The library also contains objects for directories, files, and links because sysfs is, after all, a file system.

Libsysfs's design was influenced by application requirements. Libsysfs emerged from common requirements from three applications: Greg Kroah-Hartman's udev, Linux Diagnostic Tools project's sysdiag, and Linux Event Logging project's Error Log Analysis.

4.1.1 Devices, Classes, and Buses

The *devices* subsystem represents the physical layout of devices in a given system. The top-level directory under the subsystem contains root devices from which other devices hang. For instance, PCI devices on a small workstation all hang off PCI device *pci0000:00*. Libsysfs represents these root devices with *sysfs_root_device* objects. The devices that are represented as directories under the *devices* subsystem of sysfs are represented using the structure *sysfs_device* in Libsysfs.

In many cases, users are more interested in the function(s) devices perform rather than their physical location. The *class* subsystem is a representation of devices based on their functionality. Devices, once registered, can be associated with one or more classes based on the functions they perform. Libsysfs uses the *sysfs_class* structure to represent classes like *net* or *scsi_host*, and the *sysfs_class_device* to represent devices in those classes like *eth0* or *host1*.

Currently under sysfs there's a subsystem, on level with the three major subsystems, for block devices. Since the *block* subsystem describes a class of devices and since it is our understanding that it will be moved under the *class* subsystem, Libsysfs presently handles it as another *class*.

The *bus* subsystem represents how devices are connected, which devices are connected to which bus, and so on. More importantly, it provides information about drivers that are registered with the bus and the devices they manage. Libsysfs represents buses with the *sysfs_bus* structure and drivers with the *sysfs_driver* structure.

4.1.2 Directories, Files and Links

One of the important design considerations for Libsysfs was to faithfully represent the sysfs file system.

From the kernel's point of view, sysfs directories represent kobjects [07] and files in the directories represent attributes [05, 06]. Sysfs extensively uses links to faithfully bring out the relationships between the various subsystems. As an example, the *device* link under any sysfs driver directory (such as e100 for an Intel® Ethernet Pro adapter) points to the physical device under the *devices* subsystem (Example illustrated in figure 12).

Libsysfs uses the *sysfs_directory, sysfs_attribute* and *sysfs_link* structures to represent sysfs's directories, attributes, and links, respectively. Libsysfs provides functions to retrieve relevent information from these structures. Navigation of these structures directly by applications is discouraged.

To reduce the memory footprint of the library and to minimize the possibility of stale values being presented to the user, a conscious decision was made to classify the structure and dlist related elements of all subsystem structures as *private* data, available only on request. With this design, reading of directories, links, and attributes can be postponed until such time the user application explicitly requests the data.

The elements of structures that are classified as being for *internal use* may not contain valid data until such time that the user calls appropriate helper functions that populate them. These helper routines return handles to the *private* structure elements.

5 Calling conventions in Libsysfs

Libsysfs uses a very simple calling convention for its APIs. All *open* calls have a corresponding *close* API. An *open* call returns a reference to a structure that has been opened. Any structure that is *opened* must be closed with a call to its corresponding *close* function.

Libsysfs provides *get* calls for user applications to obtain references to the *private* elements of the data structures. References obtained from such calls need not be closed explicitly.

The following sections describe the various structures used in Libsysfs and the APIs associated with them. While most APIs are self explanatory, some explanation is provided for quick reference. Details about all of Libsysfs's APIs can be found in *libsysfs.txt* that is shipped with the sysfsutils and udev packages.

6 Working with Directories, Files and Links

Libsysfs contains functions and structures for working with directories, files, and links. The Libsysfs calls are modeled after the file system calls: directories, files (attributes), and links must be opened and closed. The library's functions include common code like getting a directory's files, subdirectories, and links or getting a link's target. Libsysfs also provides special sysfs domain information, such as checking whether a file or attribute is a *show* and/or *store* method.

6.1 Directories

Sysfs directories can represent subsystems (like bus or class), devices, or even sets of device attributes. Libsysfs provides functions to work with directories. Libsysfs also provides functions and structures to work with bus, class, or device objects, which all use the directory routines internally. The Libsysfs directory functions are meant to generalize retrieval of directory information including subdirectories, files, and links.

The *sysfs_directory* structure acts as a handle for working with sysfs directories. It contains lists of subdirectories, links, and attributes along with the directory's name and path.

```
struct sysfs_directory {
        unsigned char name[SYSFS_NAME_LEN];
        unsigned char path[SYSFS_PATH_MAX];

        /* Private: for internal use only */
        struct dlist *subdirs;
        struct dlist *links;
        struct dlist *attributes;
};
```

Figure 1: *sysfs_directory* in Libsysfs

The *sysfs_directory* structure contains:

- *name* - The directory name
- *path* - The absolute sysfs path to this directory
- *subdirs* - The list of subdirectories under this directory
- *links* - The list of links in this directory
- *attributes* - The list of attributes (files) in this directory

The following *sysfs_directory* functions exist:

- `struct sysfs_directory *sysfs_open_directory(const unsigned char *path);`
- `void sysfs_close_directory(struct sysfs_directory *sysdir);`
- `int sysfs_read_dir_attributes(struct sysfs_directory *sysdir);`
- `int sysfs_read_dir_links(struct sysfs_directory *sysdir);`
- `int sysfs_read_dir_subdirs(struct sysfs_directory *sysdir);`
- `dlist *sysfs_get_dir_attributes(struct sysfs_directory *dir);`
- `dlist *sysfs_get_dir_links(struct sysfs_directory *dir);`
- `dlist *sysfs_get_dir_subdirs(struct sysfs_directory *dir);`

- `int sysfs_refresh_dir_attributes(struct sysfs_directory *sysdir);`
- `int sysfs_refresh_dir_links(struct sysfs_directory *sysdir);`
- `int sysfs_refresh_dir_subdirs(struct sysfs_directory *sysdir);`
- `int sysfs_read_directory(struct sysfs_directory *sysdir);`
- `int sysfs_read_all_subdirs(struct sysfs_directory *sysdir);`
- `struct sysfs_directory *sysfs_get_subdirectory(struct sysfs_directory *dir, unsigned char *subname);`
- `struct sysfs_link *sysfs_get_directory_link(struct sysfs_directory *dir, unsigned char *linkname);`
- `struct sysfs_link *sysfs_get_subdirectory_link(struct sysfs_directory *dir, unsigned char *linkname);`
- `struct sysfs_attribute *sysfs_get_directory_attribute(struct sysfs_directory *dir, unsigned char *attrname);`

The `sysfs_open_directory()` function takes a path to the directory to be opened and used. The function allocates the basic *sysfs_directory* structure, fills in path and name, and returns the directory structure to the caller. The lists of subdirectories, links, and attributes are left unpopulated.

The `sysfs_close_directory()` function closes the *sysfs_directory* structure provided. It goes through any lists of subdirectories, attributes, and links and closes all of them before deallocating the structure.

While individual lists of a sysfs_directory can be read using the appropriate `sysfs_read_dir_*` functions, the `sysfs_read_directory()` function reads the provided directory, opening subdirectories and links. Those objects are added to the *sysfs_directory*'s lists. This function doesn't recursively read subdirectories, only the directory.

The `sysfs_get_dir_*` functions can be used to retrieve references to the appropriate dlist of the sysfs_directory structure.

The `sysfs_refresh_dir_*` functions can be used in cases when the attributes, links and/or subdirectories of a given sysfs_directory need to be reread. The refresh functions need to be used with care since references held to elements of the lists prior to calling the refresh functions will not be valid upon return from these functions.

The `sysfs_read_all_subdirs()` function reads all subdirectories. It traverses the sysfs_directory's subdirectory list, individually reading every subdirectory it finds.

The `sysfs_get_subdirectory()` function retrieves a specific subdirectory by name from a *sysfs_directory*. The function travels the subdirectory list in the *sysfs_directory* structure and returns a reference to the indicated subdirectory. The returned subdirectory reference does not need to be closed, but will be closed when the parent's directory is eventually closed.

The `sysfs_get_directory_link()` retrieves a specific link from a *sysfs_directory*'s list of links. It doesn't search the directory's subdirectories. It returns a reference to the link if found. The reference doesn't need to be closed separately.

The `sysfs_get_subdirectory_link()` searches the current directory and all of its subdirectories for the given link name. If found, the function returns a reference to the link.

The `sysfs_get_directory_attribute()` returns a reference to a specific attribute if found in the given directory. As with the other get functions, the reference doesn't need to be closed.

6.2 Files

Sysfs files represent attributes for devices, drivers, classes, and buses. They are a means to communicate with kernel objects from User Space. Files can be readable, writeable, or both, reflecting the driver model *show* and *store* methods. If an attribute implements *show*, a user or application can read from the file to receive the *show* information. The *vendor* file in a device directory is a *show* method: by reading the file, a user can see the device's vendor information. A *store* method is represented with a writeable file. The SCSI host class device has a *scan* attribute; a user can invoke a scan by writing to the

file. Up to a page of data can be transferred between a user and the kernel through sysfs in either binary or ASCII form.

```
struct sysfs_attribute {
        unsigned char *value;
        unsigned short len;
        unsigned short method;
        unsigned char name[SYSFS_NAME_LEN];
        unsigned char path[SYSFS_PATH_MAX];
};
```

Figure 2: *sysfs_attribute* in Libsysfs

Libsysfs uses the *sysfs_attribute* structure to represents sysfs files. The attribute structure contains the following members:

- *value* - The value to be read from or written to the file
- *len* - The size of buffer *value*
- *method* - Bitfield to designate if attribute is a *show* method, *store* method, or both
- *name* - The name of the attribute file
- *path* - The absolute sysfs path to the attribute file

The library contains the following functions to work with attributes:

- `struct sysfs_attribute *sysfs_open_attribute (const unsigned char *path);`
- `void sysfs_close_attribute(struct sysfs_attribute *sysattr);`
- `int sysfs_read_attribute(struct sysfs_attribute *sysattr);`
- `int sysfs_read_attribute_value(const unsigned char *attrpath, unsigned char *value, size_t vsize);`
- `int sysfs_read_dir_attributes(struct sysfs_directory *sysdir);`
- `unsigned char *sysfs_get_value_from_attributes (struct dlist *attr, const unsigned char *name);`
- `int sysfs_write_attribute(struct sysfs_attribute *sysattr, const unsigned char *new_value, size_t len);`

The library contains a function to open and another to close an attribute. The `sysfs_open_attribute()` function takes the path to the file to open. It allocates the *sysfs_attribute* structure, fills in the file name and path information, and then uses `stat()` to determine if the file is a *show* and/or *store* method. The file isn't read or opened at this point. The `sysfs_close_attribute()` function deallocates the created *sysfs_attribute* structure.

The sysfs file is opened only when a read or a write is performed. The `sysfs_read_attribute()` function opens, reads, and closes the file pointed to by the opened *sysfs_attribute* structure. The data read from the file is assigned to the *value* field and its length to the *sysfs_attribute*'s *len* field. The `sysfs_write_attribute()` function performs the reverse: it opens the file, writes from the provided buffer, and then closes the file.

The `sysfs_read_attribute_value()` function is provided for convenience. The caller doesn't need to open and handle a *sysfs_attribute* structure, it merely supplies the file path, a buffer, and a buffer length. The function is a wrapper around the open, read, and close *sysfs_attribute* functions.

The `sysfs_read_dir_attributes()` function compliments the `sysfs_read_directory()` function, in that it reads the *sysfs_directory* provided and opens up the list of attributes for the directory.

The `sysfs_get_value_from_attributes()` function retrieves a specific attribute's value from a list of attributes. This function is useful if a user has opened a directory with a list of attributes and wishes to get a specific attribute's value from the list. It doesn't require the user to open or close attributes, which are already opened and contained in the attribute list. The function traverses the attribute list looking for the attribute with the provided name and returns its value.

6.2.1 Write attribute support in Libsysfs

As discussed earlier, attributes in sysfs are exported to userspace as normal *files*. Quite a few attributes of the driver model components are configurable, and sysfs provides a method to alter these attributes by providing *write* permission to such attribute files. Libsysfs provides the following function to modify

such attribute values:

- `int sysfs_write_attribute(struct sysfs_attribute *sysattr, const unsigned char *new_value, size_t len);`

`sysfs_write_attribute()` returns an *error* when:

- The *sysfs_attribute* supplied is not "writable".
- The return value on `write()` is not equal to the supplied *len*. In such a case, the initial value of the attribute will be restored before returning to the calling function.

Upon return from this function, `sysattr->value` will contain the current value of the attribute.

6.3 Links

Links are used in sysfs to relate objects to one another, such as a physical device to its block device representation or a class device to its physical device. Libsysfs uses the *sysfs_link* structure to represent the important link information including the link's name, path, and target path.

```
struct sysfs_link {
        unsigned char name[SYSFS_NAME_LEN];
        unsigned char path[SYSFS_PATH_MAX];
        unsigned char target[SYSFS_PATH_MAX];
};
```

Figure 3: *sysfs_link* in Libsysfs

The *sysfs_link* structure contains:

- *name* - The name of the link
- *path* - The absolute sysfs path to the link
- *target* - The absolute sysfs path to where this link points to

Libsysfs has the following functions for working with sysfs links:

- `struct sysfs_link *sysfs_open_link(const unsigned char *lnpath);`
- `void sysfs_close_link(struct sysfs_link *ln);`
- `int sysfs_get_link(const unsigned char *path, unsigned char *target, size_t len);`

The `sysfs_open_link` function allocates and returns the *sysfs_link* structure. The `sysfs_close_link` function deallocates the opened *sysfs_link* structure.

The `sysfs_get_link` function takes a link path and returns its target in the supplied target buffer. The function uses `readlink()` to get the target.

7 Working with Devices

7.1 Definition

A *device* is a logical or physical system resource. It is associated with a bus and is usually managed by a driver.

7.2 Devices in sysfs

Every device has its own directory in sysfs under the *devices* subsystem. Device attributes are exposed as files in the device directory. The device's directory name is its bus_id or kernel name.

Links are used in the *bus* and *class* subsystems to refer to specific devices. Each bus directory contains device links to those devices registered with it. A device directory must have one link only from the *bus* subsystem. If class devices refer to a specific physical device, they will have a link to the device they represent.

7.3 Device Structure in Libsysfs

Figure 5 shows how the library represents a sysfs *device*.

The *sysfs_device* structure contains:

- *name* - The name of the device (same as *bus_id* as of now)

```
[stekloff@... sys]$ tree /sys/devices/pci0000:05/
0000:05:02.0/
/sys/devices/pci0000:05/0000:05:02.0/
|-- class
|-- config
|-- detach_state
|-- device
|-- irq
|-- power
|   `-- state
|-- resource
|-- subsystem_device
|-- subsystem_vendor
`-- vendor
```

Figure 4: Device Directory in sysfs Example

```
struct sysfs_device {
        unsigned char name[SYSFS_NAME_LEN];
        unsigned char bus_id[SYSFS_NAME_LEN];
        unsigned char bus[SYSFS_NAME_LEN];
        unsigned char driver_name[SYSFS_NAME_LEN];
        unsigned char path[SYSFS_PATH_MAX];

        /* Private: for internal use only */
        struct sysfs_device *parent;
        struct dlist *children;
        struct sysfs_directory *directory;
};
```

Figure 5: *sysfs_device* in Libsysfs

- *bus_id* - The kernel representation of this device
- *bus* - The *bus* this device is registered with
- *driver_name* - The name of the *sysfs_driver* used by this device
- *path* - The absolute path to the device
- *parent* - The reference to a device's parent
- *children* - The list of child devices that spawn off this device
- *directory* - The *sysfs_directory* representation for the device directory

7.4 Device Functions

Libsysfs provides the following functions to work with devices:

- `struct sysfs_device *sysfs_open_device_path(const unsigned char *path);`
- `struct sysfs_device *sysfs_open_device(const unsigned char *bus, const unsigned char *bus_id);`
- `void sysfs_close_device(struct sysfs_device *device);`
- `struct sysfs_device *sysfs_get_device_parent(struct sysfs_device *dev);`
- `int sysfs_get_device_bus(struct sysfs_device *dev);`
- `struct dlist *sysfs_get_device_attributes (struct sysfs_device`
- `struct dlist *sysfs_refresh_device_attributes (struct sysfs_device *device);`
- `struct sysfs_attribute *sysfs_get_device_attr(struct sysfs_device *dev, const unsigned char *name);`
- `struct sysfs_attribute *sysfs_open_device_attr (const unsigned char *bus, const unsigned char *bus_id, const unsigned char *attrib);`

The `sysfs_open_device_path()` takes the absolute sysfs *path* to the device as an argument and returns a reference to the *sysfs_device* structure.

The `sysfs_open_device()` function can be used to obtain a reference to the *sysfs_device* structure for a device whose kernel representation (*bus_id*) and *bus* are known.

The `sysfs_close_device()` function closes the given *sysfs_device*. It walks the list of device *children* and closes all of them before freeing the structure.

The `sysfs_get_device_parent()` function returns a reference to the sysfs_device structure of the parent of the device *dev*.

The `sysfs_get_device_bus()` function returns the name of the bus the device is registered on in the *bus* field of the sysfs_device structure upon successful return.

The `sysfs_get_device_attributes()` function returns a reference to a list of defined attributes for the given *sysfs_device*. The function

`sysfs_refresh_device_attributes()` is intended to be used in situations where the attributes need to be reread. Prior references to attributes from this list will not be valid upon return from the refresh function.

The `sysfs_get_device_attr()` function returns a reference to the *sysfs_attribute* structure for the requested *name* attribute and device *dev*.

The function `sysfs_open_device_attr()` returns a reference to the *sysfs_attribute* as defined for a device (*bus_id*) whose *bus* is known. The *sysfs_attribute* reference obtained upon successful return has to be closed with a call to `sysfs_close_attribute()`.

Libsysfs also provides for a *root* device representation. A *sysfs_root_device* is basically a representation for kobjects under the `/devices` directory. These kobjects are the nodes from which the device topology tree originate. Figure 6 shows the *sysfs_root_device* structure.

```
struct sysfs_root_device {
    unsigned char name[SYSFS_NAME_LEN];
    unsigned char path[SYSFS_PATH_MAX];

    /* Private: for internal use only */
    struct dlist *devices;
    struct sysfs_directory *directory;
};
```

Figure 6: *sysfs_root_device* in Libsysfs

The following functions work on the *root* kobjects:

- `struct sysfs_root_device *sysfs_open_root_device(const unsigned char *name);`
- `void sysfs_close_root_device(struct sysfs_root_device *root);`
- `struct dlist *sysfs_get_root_devices(struct sysfs_root_device *root);`

The `sysfs_open_root_device()` function returns a handle to the *sysfs_root_device* structure corresponding to `/sys/devices/`*name*, while `sysfs_close_root_device()` deallocates the structure, closing the devices list and its directory handle in the process.

Once a root device is opened, the function `sysfs_get_root_devices()` can be used to get the device tree under the root device.

8 Working with Buses

8.1 Definition

A *bus* is a medium used to connect a set of similar devices. It acts as a transport for transfering data between devices on the bus using a common protocol.

8.2 Buses in sysfs

Every bus that is supported on a system has its own directory under `/sys/bus/`. Devices that are registered with the bus are represented under `/sys/bus/xxx/devices` and drivers under `/sys/bus/xxx/drivers`. Entries under `/sys/bus/xxx/devices` are symbolic links to actual devices (physical or virtual) under the *devices* subsystem.

8.3 Bus Structures

The *sysfs_bus* structure (shown in Figure 8) acts as a handle to access bus related information. It contains lists of devices and drivers that are registered with this bus.

The *sysfs_bus* structure contains:

- *name* - The bus name
- *path* - The absolute sysfs path to the bus
- *drivers* - The list of drivers registered with this bus
- *devices* - The list of devices registered with this bus
- *directory* - The *sysfs_directory* representation for the bus kobject

```
[ananth@...sys]$ tree /sys/bus/
/sys/bus/
|- ide
|- pci
|- platform
|- pnp
|- pseudo
`- scsi
   |- devices
   |  |- 1:0:0:0 -> ../../../devices/
   |  | pci0000:02/0000:02:01.1/host1/1:0:0:0
   |  |- 1:0:1:0 -> ../../../devices/
   |  | pci0000:02/0000:02:01.1/host1/1:0:1:0
   |  |- 1:0:8:0 -> ../../../devices/
   |  | pci0000:02/0000:02:01.1/host1/1:0:8:0
   |  `- 4:0:3:0 -> ../../../devices/pci0000:05/
   |   0000:05:04.0/host4/4:0:3:0
   `- drivers
      |- sd
      |  |- 1:0:0:0 -> ../../../../devices/
      |  | pci0000:02/0000:02:01.1/host1/1:0:0:0
      |  |- 1:0:1:0 -> ../../../../devices/
      |  | pci0000:02/0000:02:01.1/host1/1:0:1:0
      |  |- 2:0:0:0 -> ../../../../devices/
      |  | pseudo_0/adapter0/host2/2:0:0:0
      |  `- 4:0:3:0 -> ../../../../devices/
      |    pci0000:05/0000:05:04.0/host4/4:0:3:0
      `-- sr
```

Figure 7: Bus example in sysfs

8.4 Bus Functions

The following functions are available to work with buses in Libsysfs:

- `struct sysfs_bus *sysfs_open_bus(const unsigned char *name);`
- `void sysfs_close_bus(struct sysfs_bus *bus);`
- `struct dlist *sysfs_get_bus_devices(struct sysfs_bus *bus);`
- `struct dlist *sysfs_get_bus_drivers(struct sysfs_bus *bus);`
- `struct sysfs_device *sysfs_get_bus_device(struct sysfs_bus *bus, unsigned char *id);`
- `struct sysfs_driver *sysfs_get_bus_driver(struct sysfs_bus *bus, unsigned char *drvname);`
- `struct dlist *sysfs_refresh_bus_attributes (struct sysfs_bus *bus);`

```
struct sysfs_bus {
        unsigned char name[SYSFS_NAME_LEN];
        unsigned char path[SYSFS_PATH_MAX];

        /* Private: internal use only */
        struct dlist *drivers;
        struct dlist *devices;
        struct sysfs_directory *directory;
};
```

Figure 8: *sysfs_bus* in Libsysfs

- `struct dlist *sysfs_get_bus_attributes (struct sysfs_bus *bus);`
- `struct sysfs_attribute *sysfs_get_bus_attr(struct sysfs_bus *bus, unsigned char *attrname);`
- `struct sysfs_device *sysfs_open_bus_device(unsigned char *busname, unsigned char *dev_id);`
- `int sysfs_find_driver_bus(const unsigned char *driver, unsigned char *busname, size_t bsize);`

The `sysfs_open_bus()` function returns the reference to a *sysfs_bus* structure corresponding to the bus *name*.

The `sysfs_close_bus()` function closes the given *sysfs_bus* structure. It walks the list of *devices* and *drivers* closing them in the process before freeing the structure.

The `sysfs_get_bus_devices()` and `sysfs_get_bus_drivers()` functions respectively return references to lists of devices and drivers registered with the *bus*.

Given a *sysfs_bus* structure, the `sysfs_get_bus_device()` function returns the *sysfs_device* structure reference corresponding to device *id*. Similarly, the `sysfs_get_bus_driver()` function returns the *sysfs_driver* structure reference for driver with name *drvname*.

The `sysfs_get_bus_attributes()` function returns a list of attributes for the given *bus* while the `sysfs_get_bus_attr()` function returns the *sysfs_attribute* structure corresponding to *attrname*.

The `sysfs_open_bus_device()` function returns the *sysfs_device* structure reference for a device whose

bus and its kernel name (*bus_id*) are known. The *sysfs_device* reference thus obtained must be closed with a call to the `sysfs_close_device()` function.

The function `sysfs_find_driver_bus()` can be used to determine what bus a *driver* is registered with. *bsize* is the size of buffer *busname* in which to return the name of the bus.

9 Working with Classes

9.1 Definition

A *class* represents an aggregation of similar objects.

9.2 Classes in sysfs

A sysfs *class* is a set of logical devices that perform a similar function. The sysfs *class* subsystem exports these device sets to User Space. For example, all network interfaces are represented under the *net* class and usb devices are represented under the *usb* class. One device may be associated with more than one class depending on the functions it performs.

Libsysfs defines devices under classes to be *sysfs_class_device*s, different from *sysfs_device*s. A class device represents one function of a logical device. The logical device may or may not point to a physical device. A class device's directory normally contains, in addition to attributes, a link to its physical device (under the *devices* subsystem) and possibly a link to its driver.

Block devices are currently represented under the *block* subsystem immediately under the sysfs root. Libsysfs, however, considers block devices to be part of the *class* subsystem. All class functions work with block devices.

9.3 Classes in Libsysfs

Libsysfs provides two structures for working with the class subsystem.

The *sysfs_class* structure (shown in Figure 10) is used to represent a sysfs `class`.

```
[ananth@.. class] tree /sys/class/
/sys/class/
|- input
|- net
|  |- dummy0
|  |- eth0
|  |- eth1
|  |- eth2
|  `- lo
|- scsi_device
|- scsi_host
|  |- host0
|  |- host1
|  |- host2
|  `- host4
`- tty
```

Figure 9: Class example in sysfs

```
struct sysfs_class {
        unsigned char name[SYSFS_NAME_LEN];
        unsigned char path[SYSFS_PATH_MAX];

        /* Private: for internal use only */
        struct dlist *devices;
        struct sysfs_directory *directory;
};
```

Figure 10: *sysfs_class* in Libsysfs

The *sysfs_class* structure contains:

- *name* - The class name
- *path* - The absolute sysfs path to the class
- *devices* - The list of class devices under this class
- *directory* - The *sysfs_directory* representation for the bus kobject

The *sysfs_class_device* structure (shown in Figure 11) represents a device (*name*) belonging to a class (*classname*).

The *sysfs_class_device* structure contains:

- *name* - The name of the class device.
- *classname* - The class name to which this class device belongs to
- *path* - The absolute sysfs path to the class

```
struct sysfs_class_device {
        unsigned char name[SYSFS_NAME_LEN];
        unsigned char classname[SYSFS_NAME_LEN];
        unsigned char path[SYSFS_PATH_MAX];

        /* Private: for internal use only */
        struct sysfs_class_device *parent;
        struct sysfs_device *sysdevice;
        struct sysfs_driver *driver;
        struct sysfs_directory *directory;
};
```

Figure 11: *sysfs_class_device* in Libsysfs

- *parent* - The *sysfs_class_device* for the parent of this class device
- *sysdevice* - The *sysfs_device* reference for this class device
- *driver* - The *sysfs_driver* reference for this class device
- *directory* - The *sysfs_directory* representation for the bus kobject

9.4 Class Functions

The following functions are provided to work with sysfs classes and class devices:

- `struct sysfs_class *sysfs_open_class (const unsigned char *name);`
- `void sysfs_close_class(struct sysfs_class *cls);`
- `struct dlist *sysfs_get_class_devices(struct sysfs_class *cls);`
- `struct sysfs_class_device *sysfs_get_class_device(struct sysfs_class *class, unsigned char *name);`
- `struct sysfs_class_device *sysfs_open_class_device_path(const unsigned char *path);`
- `struct sysfs_open_class_device(const unsigned char *class, const unsigned char *name);`
- `void sysfs_close_class_device(struct sysfs_class_device *dev);`
- `struct sysfs_device *sysfs_get_classdev_device(struct sysfs_class_device *clsdev);`
- `struct sysfs_driver *sysfs_get_classdev_driver(struct sysfs_class_device *clsdev);`
- `struct sysfs_class_device *sysfs_get_classdev_parent(struct sysfs_class_device *clsdev);`
- `struct dlist *sysfs_get_classdev_attributes (struct sysfs_class_device *cdev);`
- `struct dlist *sysfs_refresh_classdev_attributes (struct sysfs_class_device *cdev);`
- `struct sysfs_attribute *sysfs_get_classdev_attr (struct sysfs_class_device *clsdev, const unsigned char *name);`
- `struct sysfs_open_classdev_attr(const unsigned char *classname, const unsigned char *dev, const unsigned char *attrib);`

The `sysfs_open_class()` returns a *sysfs_class* structure reference for the given class (*name*).

The `sysfs_close_class()` walks the list of *devices* and closes them in turn. It then frees the *sysfs_class* structure.

The `sysfs_get_class_devices()` function returns a list of class devices belonging to the given *class*.

The `sysfs_get_class_device()` function returns a *sysfs_device* structure reference for the class device with the given *name*.

Given the absolute sysfs *path* to a class device (*name*), the `sysfs_open_class_device_path()` returns its *sysfs_class_device* structure reference.

If the class device's *classname* and its kernel representation (*dev*) is known, the function `sysfs_open_classdev_attr()` returns the reference for attribute with name *attrib*. In keeping with Libsysfs's convention with "open" calls, the attribute reference thus obtained must be closed with a call to the `sysfs_close_attribute()` function.

The `sysfs_close_class_device()` closes the device and the driver before freeing the *sysfs_class_device* structure.

Given a sysfs_class *class*, the `sysfs_get_class_device()` looks for the *sysfs_class_device* structure corresponding to *name* from among its list of *devices*.

The `sysfs_get_classdev_device()` and `sysfs_get_classdev_driver()` functions can be used to obtain references to *sysfs_device* and *sysfs_driver* for the given class device (*clsdev*).

The `sysfs_get_classdev_parent()` function returns a valid reference to the parent *sysfs_class_device* of *clsdev*, if one is available. This function is particularly useful in cases when a block partition is considered as a class device. For example, if this function is called with a *clsdev* reference of sda1, it returns the *sysfs_class_device* for sda.

The reference to a list of attributes for a given class device (*cdev*) can be obtained using the function `sysfs_get_classdev_attributes()`. The `sysfs_refresh_classdev_attributes()` function rereads the attributes list for the class device *cdev*. Prior references held to attributes from the list will not be valid upon return from the refresh routine.

The `sysfs_get_classdev_attr()` function returns a *sysfs_attribute* structure corresponding to *name* of sysfs_class_device *clsdev*.

If the device's *class* and its kernel representation (*name*) is known, `sysfs_open_class_device_by_name()` can be used to get its *sysfs_class_device* structure reference.

10 Working with Drivers

10.1 Definition

Device drivers are software modules that manage logical and physical devices. Drivers provide an interface between applications and a device. They are responsible for registering managed devices with sysfs and implementing the callbacks that expose a managed device's information and configuration.

10.2 Drivers in sysfs

Drivers are represented in sysfs under the *bus* subsystem. Each bus directory contains a `drivers` directory. That directory contains directories for each registered driver on that bus. Each driver directory contains a link or links to devices registered with the driver. The driver directory also contains any driver attributes exposed through sysfs. Figure 12 is an example directory listing for a pci e100 driver.

```
[stekloff@.. eth0]$ tree /sys/bus/pci/drivers/
e100/
/sys/bus/pci/drivers/e100
|-- 0000:01:08.0 -> ../../../../devices/
| pci0000:00/0000:00:1e.0/0000:01:08.0
|-- 0000:01:0e.0 -> ../../../../devices/
| pci0000:00/0000:00:1e.0/0000:01:0e.0
‘-- new_id
```

Figure 12: Driver Directory in sysfs Example

The e100 device driver is currently managing two devices, the first device is at pci address 0000:01:08.0 and the second is at 0000:01:0e.0. The e100 driver's directory under bus contains a link for each device. It also includes a driver attribute called *new_id*.

The driver's device links point to the device's directory under the *devices* subsystem. There currently isn't a link back from a device to its driver, so referencing a device's driver must be done from the *bus* subsystem or the *class* subsystem.

In some cases, the *class* subsystem contains links to drivers from class devices. The *net* class is an example, where, as shown in figure 13, the device driver for network interface eth0 is *e100*.

```
[stekloff@... eth0]$ tree /sys/class/net/eth0
/sys/class/net/eth0
|-- addr_len
|-- address
|-- broadcast
|-- device -> ../../../devices/pci0000:00/
| 0000:00:1e.0/0000:01:08.0
|-- driver -> ../../../bus/pci/drivers/e100
|-- features
|-- flags
|-- ifindex
|-- iflink
|-- mtu
|-- statistics
|-- tx_queue_len
‘-- type
```

Figure 13: Class Device Driver Reference in sysfs Example

Eth0 is being managed by the e100 driver.

10.3 Driver Structure in Libsysfs

Libsysfs implements a *sysfs_driver* structure to represent a driver in sysfs. The structure can be used as a handle to retrieve driver information including driver attributes and devices.

```
struct sysfs_driver {
        unsigned char name[SYSFS_NAME_LEN];
        unsigned char path[SYSFS_PATH_MAX];

        /* internal use only */
        struct dlist *devices;
        struct sysfs_directory *directory;
};
```

Figure 14: *sysfs_driver* in Libsysfs

The structure includes the following:

- *name* - Name of the file
- *path* - Absolute path to the file
- *devices* - List of devices the driver manages
- *directory* - Directory handle for internal use

10.4 Driver Functions in Libsysfs

Libsysfs contains the following functions for working with drivers:

- `struct sysfs_driver *sysfs_open_driver_path(const unsigned char *path);`
- `struct sysfs_driver *sysfs_open_driver(const unsigned char *bus_name, const unsigned char *drv_name);`
- `void sysfs_close_driver(struct sysfs_driver *driver);`
- `struct dlist *sysfs_get_driver_devices(struct sysfs_driver *driver);`
- `struct dlist *sysfs_refresh_driver_devices (struct sysfs_driver *driver);`
- `struct sysfs_device *sysfs_get_driver_device(struct sysfs_driver *driver, const unsigned char *name);`
- `struct sysfs_attribute *sysfs_get_driver_attr(struct sysfs_driver *drv, const unsigned char *name);`
- `struct dlist *sysfs_get_driver_attributes (struct sysfs_driver *driver);`
- `struct dlist *sysfs_refresh_driver_attributes (struct sysfs_driver *driver);`
- `struct dlist *sysfs_get_driver_links(struct sysfs_driver *driver);`
- `struct sysfs_attribute *sysfs_open_driver_attr(const unsigned char *bus, const unsigned char *drv, const unsigned char *attrib);`

As with other objects in Libsysfs, the *sysfs_driver* must be opened to be used and closed when finished. The `sysfs_open_driver_path()` function opens the driver at the supplied path, allocates the structure, and returns it to the caller. The `sysfs_open_driver()` function takes the bus name the driver is registered with and the driver name as parameters and returns a *sysfs_driver* reference for the driver.

The `sysfs_close_driver()` function deallocates the driver structure, closing its internal directory handle and all the devices that had been added to the devices list.

The `sysfs_get_driver_devices()` function returns a reference to the list of all devices that this driver manages. The `sysfs_refresh_driver_devices()` function can be used to reread the list of devices managed by this driver. Please note that prior references to the devices the driver manages will not be valid upon return from the refresh function.

The `sysfs_get_driver_device()` function returns a *sysfs_device* reference for the device with name *name* using the driver *driver*.

The `sysfs_get_driver_attr()` function takes a *sysfs_driver* and an attribute name and returns a reference to the driver's requested *sysfs_attribute*. The returned attribute reference doesn't need to be closed, it will be closed when the driver object is closed.

The `sysfs_get_driver_attributes()` function returns a reference to the list containing all of the driver's attributes. The

`sysfs_refresh_driver_attributes` function rereads the attributes list for the given *driver*. Prior references to any attributes for the driver will not be valid upon return from this function. The `sysfs_get_driver_links()` function, similarly, returns a reference to a list containing all of the driver's links. The driver's links represent the devices it manages. As above and as convention with Libsysfs's *get* functions, the referenced dlist returned with these functions doesn't need to be closed.

The `sysfs_open_driver_attr()` function retrieves a specific driver's attribute using the driver's name, its bus, and the name of the needed attribute. This function allows the caller to work with a driver's attribute without working with the *sysfs_driver* structure. Both of these functions require the caller to either close the *sysfs_driver* or close the *sysfs_attribute* when finished.

11 API Usage Example: Accessing Network Device Eth0

This example outlines the steps necessary for opening and obtaining information from network class device eth0. We can use the class device handle to get information from eth0 including its phsyical device, its attributes like statistics or MAC address, and its controlling driver.

The first task is to open the class device handle for eth0, which is done using the function `sysfs_open_class_device`. The function takes the name of the class the device is in and the class device name. It returns a *sysfs_class_device* structure that can is to be used as a handle for working with the class device. This structure must eventually be closed.

```
class_device = sysfs_open_class_device
                          ("net", "eth0");
```

Figure 15: Opening a *sysfs_class_device* in Libsysfs

Once a handle has been opened, we can use *get* functions to retrieve the class device's information. If, for example, we wanted to retrieve the class device's physical device, we would use the `sysfs_get_classdev_device` function. This function returns a handle to the class device's sysfs device. The *sysfs_device* handle that's returned doesn't need to be closed, it is referenced through the class device's handle and will be cleaned up when the class device is closed.

```
device = sysfs_get_classdev_device
                          (class_device);
```

Figure 16: Getting a Class Device's *sysfs_device* in Libsysfs

We can also use the class device's handle to get a specific attribute, like eth0's address. We would use the `sysfs_get_classdev_attr` function that takes the class device and the name of the desired attribute. It returns a *sysfs_attribute* handle.

```
attr = sysfs_get_classdev_attr
             (class_device, "address");
```

Figure 17: Getting a Class Device's *sysfs_attribute* in Libsysfs

Once we've finished accessing the class device, we must close it using the `sysfs_close_class_device` function. This function cleans up and closes everything that was referenced with the *get* functions.

12 Future

Libsysfs was envisioned to be the library to use for accessing sysfs information. As sysfs evolves, there will be changes to its structure and hence Libsysfs has to keep pace with these changes.

Many applications already are using Libsysfs. Examples are Greg Kroah-Hartman's udev - the User Space implementation of devfs, Patrick Mansfield's scsi_id utility, IBM LTC's sysdiag which acts as a single interface to a number of diagnostic utilities and IBM LTC's Event Log Analyzer (ELA). Other applications that use the library include Christophe Varoqui's multipath utility and the OpenHPI project.

It is our endeavour to provide applications a stable API irrespective of changes to the basic sysfs structure.

Many of the additions/changes to Libsysfs are a direct consequence of requirements of applications

that are using it. This is an ongoing activity.

13 Conclusion

The authors of the library set out with the goal of making sysfs access easy. Libsysfs is still evolving and it is hoped that the library is viewed as the the gateway to sysfs. Applications that use the library and other apps that ship with the library (such as systool) serve as examples for Libsysfs usage. As more and more applications start using the library and with active community involvement, it is hoped that Libsysfs will be more efficient and robust in the days to come.

14 Availability

The latest version of the library and utilities that come with it (as part of the sysfsutils package) can always be found at:

`http://linux-diag.sourceforge.net`

References

[01] Patrick Mochel, *The Linux Kernel Device Model,* Ottawa Linux Symposium, 2002

[02] Patrick Mochel, *Documentation/driver-model/**

[03] Patrick Mochel, *Documentation/filesystems/sysfs.txt*

[04] Jonathan Corbet, *Driver porting: Device model overview*, `http://lwn.net/Articles/31185`

[05] Jonathan Corbet, *Driver porting: kobjects and sysfs*, `http://lwn.net/Articles/54651`

[06] Jonathan Corbet, *Driver porting: Devices and attributes*, `http://lwn.net/Articles/31220`

[07] Jonathan Corbet, *Driver porting: The zen of kobjects*, `http://lwn.net/Articles/51437`

15 Trademarks and Disclaimer

Appendix

A Utility Functions

Libsysfs provides a set of utility functions. Some of them are listed here:

- `int sysfs_get_fs_mnt_path(const unsigned char *fs_type, unsigned char *mnt_path, size_t len);`
- `int sysfs_get_mnt_path(unsigned char *mnt_path, size_t len);`
- `int sysfs_get_link(const unsigned char *path, unsigned char *target, size_t len);`
- `void sysfs_close_list(struct dlist *list);`
- `struct dlist *sysfs_open_subsystem_list (unsigned char *name);`
- `struct dlist *sysfs_open_bus_devices_list (unsigned char *name);`
- `int sysfs_path_is_dir(const unsigned char *path);`
- `int sysfs_path_is_link(const unsigned char *path);`

- `int sysfs_path_is_file(const unsigned char *path);`

The function `sysfs_get_fs_mnt_path()` can be used to obtain the mount point of *any* file system. It takes the *fs_type* as argument and returns the mount point in buffer *mnt_path* which is a user supplied buffer of size *len* bytes. If there is more than one filesystem of the same fs_type, this function returns the first successful match on a `getmntent()` call.

The `sysfs_get_mnt_path()` function is used to find the sysfs mount point on the system. Parameters to the function are *mnt_path* which is a buffer of size *len* to return the sysfs mount point.

The function `sysfs_get_link()` uses `readlink()` to retrieve the symbolic link *path* and returns its absolute sysfs path in buffer *target* which is of size *len* bytes.

`sysfs_close_list()` is a generic list close function. This function can be used in all cases where the list elements are simple character pointers.

The `sysfs_open_subsystem_list()` function returns a reference to a list of kobjects under the *name* subsystem in sysfs. For example, `sysfs_open_subsystem_list(bus)` returns a reference to a list containing supported buses (such as *pci, scsi ..*). The list obtained upon successful return has to be closed by a call to `sysfs_close_list()`.

The `sysfs_open_bus_devices_list()` function returns a reference to the list of devices under bus *name*. As before, the list thus obtained has to be closed using `sysfs_close_list()`.

The `sysfs_path_is_*` functions can be used to validate if the given *path* is a directory, a link or a file.

B dlist usage

Libsysfs implements its own double linked list mechanism that's lightweight and full featured. Dlist users allocate their own data and give it to dlist. Dlist supports Perl style list semantics including `dlist_push()`, `dlist_pop()`, `dlist_shift()`, and `dlist_unshift()`. There are inserts and deletes that take a directional parameter to specify in which direction the data should be added or deleted. Dlist also includes several `dlist_for_each()` macros for working on each list member depending on direction and working only with dlist data. Finally, dlist supports custom delete functions for complex data or simply uses `free()`.

Dlist defines the *Dlist* type for managing lists. The structure is used as a handle for working with linked lists. It includes:

- *marker* - Internal pointer to keep track of current list position
- *count* - Number of list members
- *data_size* - Size of data kept in the list
- *del_func* - Reference to custom delete function, if necessary
- *headnode* - List head node
- *head* - Pointer to head of list

```
typedef struct dlist {
        DL_node *marker;
        unsigned long count;
        size_t data_size;
        void (*del_func)(void *);
        DL_node headnode;
        DL_node *head;
} Dlist;
```

Figure 18: *Dlist* Handle in Libsysfs

Dlists are created either simply by `dlist_new()` for simple data or `dlist_new_with_delete()` for complex data. The latter takes a reference to a delete function that matches this prototype `(void *)(del(void*))`.

- `Dlist *dlist_new(size_t datasize);`
- `Dlist *dlist_new_with_delete(size_t datasize,void (*del_func)(void*));`

There are numerous methods for inserting data into or removing it from a dlist. There are the Perl push, pop, shift, and unshift functions, that have already been mentioned. Dlist also includes the following functions and macros for inserting and deleting elements:

- `void *dlist_insert(Dlist *,void *,int);`
- `void *dlist_insert_sorted(struct dlist *list, void *new, int (*sorter)(void *, void *));`
- `void dlist_delete(Dlist *,int);`
- `dlist_insert_before(A,B)`
- `dlist_insert_after(A,B)`
- `dlist_delete_before(A)`
- `dlist_delete_after(A)`

Dlist implements multiple macros for traversing and working with lists. The `dlist_prev()` and `dlist_next()` macros help navigate through lists while the `dlist_for_each()` macros help iterate through the list working with each element.

- `dlist_prev(A)`
- `dlist_next(A)`
- `dlist_for_each(list)`
- `dlist_for_each_rev(list)`
- `dlist_for_each_nomark(list, iterator)`
- `dlist_for_each_nomark_rev(list, iterator)`
- `dlist_for_each_data(list, data_iterator, datatype)`
- `dlist_for_each_data_rev(list, data_iterator, datatype)`
- `dlist_for_each_data_nomark(list, iterator, data_iterator, datatype)`
- `dlist_for_each_data_nomark_rev(list, iterator, data_iterator, datatype)`

Finally, to clean up a dlist a user must call `dlist_destroy()`. The destroy function will iterate through the list removing members. If the list was created with a custom delete function, that function will be called to remove the data. Otherwise, it will use `free()`.