DebugFS

Debugfs exists as a simple way for kernel developers to make information available to user space. Unlike /proc, which is only meant for information about a process, or sysfs, which has strict one-value-per-file rules, debugfs has no rules at all. Developers can put any information they want there. The debugfs filesystem is also intended to not serve as a stable ABI to user space; in theory, there are no stability constraints placed on files exported there. The real world is not always so simple, though [1]; even debugfs interfaces are best designed with the idea that they will need to be maintained forever.

Debugfs is typically mounted with a command like:

```
mount -t debugfs none /sys/kernel/debug
```

(Or an equivalent /etc/fstab line). The debugfs root directory is accessible only to the root user by default. To change access to the tree the "uid", "gid" and "mode" mount options can be used.

Note that the debugfs API is exported GPL-only to modules.

Code using debugfs should include <linux/debugfs.h>. Then, the first order of business will be to create at least one directory to hold a set of debugfs files:

```
struct dentry *debugfs_create_dir(const char *name, struct dentry *parent);
```

This call, if successful, will make a directory called name underneath the indicated parent directory. If parent is NULL, the directory will be created in the debugfs root. On success, the return value is a struct dentry pointer which can be used to create files in the directory (and to clean it up at the end). An ERR_PTR(-ERROR) return value indicates that something went wrong. If ERR_PTR(-ENODEV) is returned, that is an indication that the kernel has been built without debugfs support and none of the functions described below will work.

The most general way to create a file within a debugfs directory is with:

```
struct dentry *debugfs_create_file(const char *name, umode_t mode,
                                   struct dentry *parent, void *data,
                                   const struct file_operations *fops);
```

Here, name is the name of the file to create, mode describes the access permissions the file should have, parent indicates the directory which should hold the file, data will be stored in the i_private field of the resulting inode structure, and fops is a set of file operations which implement the file's behavior. At a minimum, the read() and/or write() operations should be provided; others can be included as needed. Again, the return value will be a dentry pointer to the created file, ERR_PTR(-ERROR) on error, or ERR_PTR(-ENODEV) if debugfs support is missing.

Create a file with an initial size, the following function can be used instead:

```
void debugfs_create_file_size(const char *name, umode_t mode,
                              struct dentry *parent, void *data,
                              const struct file_operations *fops,
                              loff_t file_size);
```

file_size is the initial file size. The other parameters are the same as the function debugfs_create_file.

In a number of cases, the creation of a set of file operations is not actually necessary; the debugfs code provides a number of helper functions for simple situations. Files containing a single integer value can be created with any of:

```
void debugfs_create_u8(const char *name, umode_t mode,
                       struct dentry *parent, u8 *value);
void debugfs_create_u16(const char *name, umode_t mode,
                        struct dentry *parent, u16 *value);
void debugfs_create_u32(const char *name, umode_t mode,
                        struct dentry *parent, u32 *value);
void debugfs_create_u64(const char *name, umode_t mode,
                        struct dentry *parent, u64 *value);
```

These files support both reading and writing the given value; if a specific file should not be written to, simply set the mode bits accordingly. The values in these files are in decimal; if hexadecimal is more appropriate, the following functions can be used instead:

```
void debugfs_create_x8(const char *name, umode_t mode,
                       struct dentry *parent, u8 *value);
void debugfs_create_x16(const char *name, umode_t mode,
                        struct dentry *parent, u16 *value);
void debugfs_create_x32(const char *name, umode_t mode,
                        struct dentry *parent, u32 *value);
void debugfs_create_x64(const char *name, umode_t mode,
                        struct dentry *parent, u64 *value);
```

These functions are useful as long as the developer knows the size of the value to be exported. Some types can have different widths on different architectures, though, complicating the situation somewhat. There are functions meant to help out in such special cases:

```
void debugfs_create_size_t(const char *name, umode_t mode,
                           struct dentry *parent, size_t *value);
```

As might be expected, this function will create a debugfs file to represent a variable of type size_t.

Similarly, there are helpers for variables of type unsigned long, in decimal and hexadecimal:

```
struct dentry *debugfs_create_ulong(const char *name, umode_t mode,
                                    struct dentry *parent,
                                    unsigned long *value);
void debugfs_create_xul(const char *name, umode_t mode,
                        struct dentry *parent, unsigned long *value);
```

Boolean values can be placed in debugfs with:

```
void debugfs_create_bool(const char *name, umode_t mode,
                         struct dentry *parent, bool *value);
```

A read on the resulting file will yield either Y (for non-zero values) or N, followed by a newline. If written to, it will accept either upper- or lower-case values, or 1 or 0. Any other input will be silently ignored.

Also, atomic_t values can be placed in debugfs with:

```
void debugfs_create_atomic_t(const char *name, umode_t mode,
                             struct dentry *parent, atomic_t *value)
```

A read of this file will get atomic_t values, and a write of this file will set atomic_t values.

Another option is exporting a block of arbitrary binary data, with this structure and function:

```
struct debugfs_blob_wrapper {
    void *data;
    unsigned long size;
};

struct dentry *debugfs_create_blob(const char *name, umode_t mode,
                                   struct dentry *parent,
                                   struct debugfs_blob_wrapper *blob);
```

A read of this file will return the data pointed to by the debugfs_blob_wrapper structure. Some drivers use "blobs" as a simple way to return several lines of (static) formatted text output. This function can be used to export binary information, but there does not appear to be any code which does so in the mainline. Note that all files created with debugfs_create_blob() are read-only.

If you want to dump a block of registers (something that happens quite often during development, even if little such code reaches mainline. Debugfs offers two functions: one to make a registers-only file, and another to insert a register block in the middle of another sequential file:

```
struct debugfs_reg32 {
    char *name;
    unsigned long offset;
};

struct debugfs_regset32 {
    const struct debugfs_reg32 *regs;
    int nregs;
    void __iomem *base;
    struct device *dev;     /* Optional device for Runtime PM */
};

debugfs_create_regset32(const char *name, umode_t mode,
                        struct dentry *parent,
                        struct debugfs_regset32 *regset);

void debugfs_print_regs32(struct seq_file *s, const struct debugfs_reg32 *regs,
                     int nregs, void __iomem *base, char *prefix);
```

The "base" argument may be 0, but you may want to build the reg32 array using __stringify, and a number of register names (macros) are actually byte offsets over a base for the register block.

If you want to dump an u32 array in debugfs, you can create file with:

```
struct debugfs_u32_array {
    u32 *array;
    u32 n_elements;
};

void debugfs_create_u32_array(const char *name, umode_t mode,
                    struct dentry *parent,
                    struct debugfs_u32_array *array);
```

The "array" argument wraps a pointer to the array's data and the number of its elements. Note: Once array is created its size can not be changed.

There is a helper function to create device related seq_file:

```
void debugfs_create_devm_seqfile(struct device *dev,
                             const char *name,
                             struct dentry *parent,
                             int (*read_fn)(struct seq_file *s,
                                     void *data));
```

The "dev" argument is the device related to this debugfs file, and the "read_fn" is a function pointer which to be called to print the seq_file content.

There are a couple of other directory-oriented helper functions:

```
struct dentry *debugfs_rename(struct dentry *old_dir,
                              struct dentry *old_dentry,
                              struct dentry *new_dir,
                              const char *new_name);

struct dentry *debugfs_create_symlink(const char *name,
                                      struct dentry *parent,
                                      const char *target);
```

A call to debugfs_rename() will give a new name to an existing debugfs file, possibly in a different directory. The new_name must not exist prior to the call; the return value is old_dentry with updated information. Symbolic links can be created with debugfs_create_symlink().

There is one important thing that all debugfs users must take into account: there is no automatic cleanup of any directories created in debugfs. If a module is unloaded without explicitly removing debugfs entries, the result will be a lot of stale pointers and no end of highly antisocial behavior. So all debugfs users - at least those which can be built as modules - must be prepared to remove all files and directories they create there. A file can be removed with:

```
void debugfs_remove(struct dentry *dentry);
```

The dentry value can be NULL or an error value, in which case nothing will be removed.

Once upon a time, debugfs users were required to remember the dentry pointer for every debugfs file they created so that all files could be cleaned up. We live in more civilized times now, though, and debugfs users can call:

```
void debugfs_remove_recursive(struct dentry *dentry);
```

If this function is passed a pointer for the dentry corresponding to the top-level directory, the entire hierarchy below that directory will be removed.

[1] http://lwn.net/Articles/309298/