Storage and Filesystems

Why a filesystem (1/2)

Usually, the applications needs to save some data

- Parameters
- Calibration
- Possibly log files

Defining magic offsets in flash only lasts so far

- After a few items you get lost
- You can't trade maintainability for laziness

Whatever your storage method, always include a version

- If the "storage version" in the app mismatches, you must reset the storage
- The filesystem layout will change over time
- And the application format new devices anyways

This is different from the typical FS use in bigger systems

Why a filesystem (2/2)

Sometimes, you use a filesystem to exchange data with the outside world

- mount an SD card to read/write data
- be mounted as USB storage device
- send acquired data through the network to a NAS device

To do this you need a known format

Unfortunately, even the simple (and stupid) FAT may be big

- I couldn't easily find figures, but at least a dozen kB for sure
- Clearly, a single-file approach like "firmware.bin" can be small
  - but then it is a file, not a filesystem
  - and the trick can't be applied to SD cards

A look at real filesystems (the Unix way)

A filesystem includes both data and metadata

- And directories obviously

Metadata is stored in the "inode" data structure

- In each filesytem, inodes are numbered

A directory is just a list of names and inode numbers

- Metadata is not directory-specific
- Most users don't care about metadata

A directory is a "special" file, like devices and sockets

- It doesn't feature an associated data area
- The behaviour of open(), read() etc is different

For "normal" files, the inode describes the on-disk mapping

- And any access is then routed to a storage device

Finally, the super-block offers summary information and stats

The Linux way to define a filesystems

The code is laid out as a set of "operations"

`struct super_operations;`

- **Mount, umount, ...**
- **These are the methods to act on the filesystem as a whole**

`struct inode_operations;`

- **Mkdir, rmdir, rename, unlink, ...**
- **It is the set of methods that act on the file-tree**

`struct file_operations;`

- **Open, release, read, write, mmap, ...**
- **The set of methods that act on data proper**
- **And they are different according to the file type**

Two simple examples

Minixfs (developed by Andrew Tanenbaum for his Minix OS)

- It was meant to be simple and small, and Unix-like
- 14byte filenames
- 64k inodes
- It shows its age for real systems, and it's too big of uC.

RomFS (introduced in Linux-2.2 for embedded systems)

- Read-only data structures, all files are concatenated
- No support for user and group
- No support for date and time
- No support for physical links.
- Created, as a file, by a user-space tool ("genromfs")
- Still maintained and useful in some situations
- It is around 5kB of compiled code

The microcontroller world is simpler

Our environment is smaller, but the basic ideas still apply

What we need for sure:

- A "superblock" (version, magic, size, ...)
- Some "filename" support, and the lenght information
- Mapping from the name to a data area

What we might need, or not:

- Subdirectories
- Creation of new files (i.e. write support for the structure)
- Write support (for the files themselves)

What we definitely do not need:

- Device files and sockets
- Owner, group, time, permission
- Complex enlarge/shrink policies for files

Possible filesystem layouts

Splitting metadata and data

- Metadata could just live within the application binary
- Data must definitely live on external storage
- The application needs to host a copy anyways for initial setup
- This should really be a structure, not a bunch of #define
- And it must include version and magic number in the data.

TLV (type length and value)

- All addresses are relatives
- Subdirectories are easy to add (as a special type)
- A writable file could be allocated to maximum-foreseen-size

Really, TLV is to be considered for a filesystem

- The concept is well known and appreciated
- The code size to access it is very small
- The main disadvantage is that access is sequential

Another option: SDB

SDB: Self Describing Bus

- This was born as a way do describe memory areas
- To make some order, and add autodetection in FPGA systems

It lends itself very well to make a simple filesystem

- We already used it to that aim, with a barebox patch
- We drafted a sdbfs library, but it's not really complete

All addresses are relative

- This allows the structure to be moved around
- And subdirectories use relative addresses too
  - You can easily plug pre-filled subdirs to a system

```
https://ohwr.org/project/sdb.git
https://people.mpi-sws.org/~mvanga/files/papers/sdb-1.1.pdf
```