File Systems in Embedded Systems (Part 1)

Linux File Systems

This is the first installment in a multi-part article series about Linux file systems for embedded systems. This article introduces the topic and plunges into some important details about flash file systems.

By Bob Japenga (USA)

The closing installment of my article series about concurrency in embedded systems examined how files can be used in building such systems. If you missed that article, you may want to go back and read it.

Although you could build a Linux system without a file system, most Linux systems have some sort of file system. And there are various types. There are file systems that do not retain their data (volatile) across power outages (i.e., RAM drives). There are nonvolatile read-only file systems that cannot be changed (e.g., CRAMFS). And there are nonvolatile read/write file systems.

WHAT IS THIS ARTICLE SERIES ABOUT?

Linux provides all three types of file systems. This article series will address all of them. Part 1 begins by looking at read/write file systems that retain their data over a power down. In particular, this article will examine flash file systems. Although ruggedized hard drives are available for use in embedded systems, most embedded systems that have file systems use flash file systems.

When using a flash file system, it is important for the system designer to be aware of some of the system's limitations before choosing to use it. This article addresses two of the limitations to using a flash file system and explains how to overcome these restrictions. The next few articles will provide more details every designer should be aware of when using these flash file systems and discuss the other two types of Linux file systems.

FLASH FILE SYSTEMS

The embedded systems I designed in the 1970s used either ultraviolet (UV) erasable memory or one-time programmable (OTP) memory to store program and data. There were no file systems. For UV erasable memory, I would use a programmable read-only memory (PROM) programmer to program an erased memory chip. The previously programmed memory chips would be erased by placing the chip under a UV light. It usually took about 30 min to erase a chip.

Around 1982, I worked on my first project that had EEPROM (i.e., electrically erasable PROM). The software could be programmed without having to remove the memory chip. I thought I was in heaven. This was a giant breakthrough in developing embedded systems.

The UV erasable memory process was time consuming during development. Because these early EEPROMs only had chip erase capabilities and not individual byte or block erase capabilities, it was difficult to design a file system with such devices.

In 1980, Toshiba invented an electrically programmable memory chip that could be erased in blocks rather than requiring erasing the entire chip. Because the erasure was so fast, it was dubbed "flash," and the name stuck.

FLASH MEMORY TYPES

It was quickly evident that these flash memories could be used to create nonvolatile read/write file systems. However, the first flash file systems were not developed until the early 1990s, almost 10 years after the

technology was invented. In some ways, this was because there are two major limitations in the flash memory technology that affected its use in a flash file system. This was partly because there were different types of flash memory: NAND and NOR. **Table 1** provides a comparison of the two technologies.

Internally, NOR flash connects the memory cells in parallel, enabling random access to a memory cell. This parallel connection resembles a NOR gate's structure. This is why it is easy for you to execute your code directly out of the flash.

NAND gets its name because of the serial connection of the memory cells resembles the structure of a NAND gate. NAND flash does not permit random access. NAND's much smaller erase page size makes it look more like a hard drive with blocks and sectors. Thus, it is easier to create a file system out of NAND flash.

That said, one of the first flash file systems we designed used NOR. Also, booting a NOR flash is much easier than booting a NAND flash. Many systems mix NOR and NAND for that reason. However, the remainder of this article will concentrate on NAND flash memory technology since it is primarily used in flash file systems.

Type/Feature	NAND	NOR
Cost per bit	Low	High
Standby power	Medium low	Low
Active power	Low	Medium
Read speed	Medium high	High
Write speed	High	Low
Erase time	Fast (typically 2 ms)	Slow (900 ms)
Capacity	High	Low
Code execution	Very difficult	Easy
File system usage	Easy	Difficult

TABLE 1
Here is a comparison of NAND and NOR flash memory technologies.

LIMITATIONS OF USING FLASH MEMORY IN A FILE SYSTEM

There are several limitations when using flash memory in a file system. The first constraint is that data stored in flash memory has a limited life (i.e., the flash memory will retain the written data for a limited amount of time).

The datasheet of a device we are using in one of our most recent products specifies 10-year data retention. Even UV memories, if not exposed to any UV, could retain data for more than 200 years. Ten years is insufficient for most of the embedded systems we build. The first embedded system I designed using UV memories was still running 25 years after it was installed. Ten years of data retention may be fine for a cell phone, but most of our systems are required to last much longer than 10 years.

The second limitation to flash memory technology was that it had a limited number of write/erase cycles before it "wore out." The two types of NAND flash memory—single-level cell (SLC) and multi-level cell (MLC)—have different impediments. MLC memories have a memory cell that can store 2 bits of information. SLC memories only store 1 bit per cell. SLC chips are lower density and have better performance and longer life. The SLC flash memory used in our most recent product is limited to 100,000 cycles. The Toshiba 90-nm MLC memory chips are rated at only 10,000 cycles.

The way in which the file system handles repetitive writes to the same file is critical to how long the memory will last. File systems used on magnetic hard drives overwrote a file in the exact same sector on the disk. This would be unacceptable for a flash file system.

The first flash file system I used in the 1990s did not handle this, so we needed to keep track of how many times we wrote a particular file and then switched to a different file after a fixed number of writes. For example, if an application wrote to some file at the same location every second, an MLC chip would be worn out after 3 h. Before two days expired, an SLC flash memory chip's memory cells would be worn out.

WEAR LEVELING

Wear leveling was developed to address both of these limitations. Wear leveling describes an algorithm that, when applied to a flash memory, can dramatically level out the number of writes to any given memory block across the entire memory chip.

Simply put, imagine keeping a counter for every block that is written. When a new block needs to be written, the algorithm tells the memory controller (software or hardware) where to write the new block. It will write the block in such a way that levels or equalizes the number of writes.

After 10 years of use, all blocks should have been erased approximately the same number of times. The wear-leveling information is stored on the chip in a region associated with each page or block. With the current part we use, there are 64 bytes reserved for every 2,048-byte page.

Most good wear-leveling algorithms do this in a way that minimizes the need to erase a "dirty" or already used block at the time of the write. You don't want to delay the write. This means that wear leveling often takes place in the background. The memory controller (software or hardware) finds dirty

blocks and erases them in the background. Sometimes it finds blocks that are partially "dirty" and moves the data to an unused block and erases the rest. This is sometimes called "garbage collection."

Not all wear-leveling algorithms are created equal. In 2005, we used a file system on an embedded system that utilized an abysmal wear-leveling algorithm. When you would least expect it (running in the background), it would take over the file system and lock out all disk accesses while it prepared for the next write! And it would do this for 2 to 3 s!

The RTOS supplier refused to fix this. The wear-leveling algorithm was developed by a big-name RTOS supplier that claimed its OS was DO-178B certified (i.e., it was flight-worthy for flight-critical software). Hence, this became the primary motivator for us and our customer to turn to Linux. We could only imagine a pilot attempting to take corrective action and the system locking out all flash file access. To make the system robust, we finally had to create a mirror drive in RAM that was periodically written to flash in a non-time-critical fashion and add battery backup to the system.

HOW DOES WEAR LEVELING ADDRESS A FLASH FILE SYSTEM'S LIMITATIONS?

Obviously, by keeping track of the number of writes to every block in the system and writing to those blocks with the lowest write cycle, the flash can be easily optimized for wear. As a designer, you need to be careful about the literature that tells you that 10,000 life erase cycles is sufficient for use in your system. The literature says that 10,000 erase cycles would enable you to erase your complete USB stick once per day for 27 years. Therefore, you may think your system will be fine for the 10 to 15 years needed. Your system doesn't write the entire memory space every day. You only write a log file or a status file once per second.

Let's do the math on that. Imagine you write a small 8-Kb file every second. Because of wear leveling, the wear-leveling manager will keep moving that 8-Kb block to a new position every second. Furthermore, imagine your system has 64 MB of memory. At the end of the day, you will have written your entire memory space and you will have a 27-year life cycle. But if you are writing a 24-Kb file, that lifetime drops to nine years. And, if you are writing twice a second, the lifetime is 4.5 years. These are not ivory tower examples. We have several systems that are recording data 10 times a second and some that are logging data once per second.

The way in which wear leveling addresses the data retention limitation is a little trickier. In fact, not all wear-leveling algorithms address the data retention limitation. There are basically two types of wear-leveling algorithms: static and dynamic.

A dynamic wear-leveling algorithm only wear levels dynamic data. For example, if your 64-MB flash disk used 32 MB for the OS and 32 MB for dynamic data, only the dynamic portion would be wear leveled. This means that at the end of the flash's life, the 32 MB of data would be worn out and the 32 MB of program will be like new.

A static algorithm wear levels both dynamic and static data. This solves two problems. First, it uses all 64 MB of data (in our example) to level off the erase cycles. In addition, there are no data retention issues since every block gets rewritten thousands of times throughout its life.

We use static wear leveling in all our Linux systems. As a designer, you need to be aware that this is going on in the background. A software bug (like we found in JFFS2 used in version 2.6.12) can cause files that are read-only from the OS's perspective to be corrupted. Until we understood wear leveling (at least in these thin slices), we were scratching our heads as to how this could happen. It happened because read-only files are getting moved as part of the static wear-leveling algorithm. And the software contained a bug that occasionally did not properly copy these read-only files.

UBIQUITOUS FLASH FILE SYSTEMS

The moral of the story is that if you are a good designer, you understand the limitations of the devices in your systems. Flash file systems are everywhere in embedded systems. As designers, we need to understand the options to make wise choices going forward.

Knowing about the flash file systems' limitations and how they are corrected in software is important for anyone who designs robust embedded systems. In Part 2, I'll dig a little deeper into flash file systems and discuss power outage protection.

ABOUT THE AUTHOR

Bob Japenga has been designing embedded systems since 1973. In 1988, along with his best friend, he started MicroTools, which specializes in creating a variety of real-time embedded systems. With a combined embedded systems experience base of more than 200 years, they love to tackle impossible problems together. Bob has been awarded 11 patents in many areas of embedded systems and motion control. You can reach him at rjapenga@microtoolsinc.com.

RESOURCE

B. Japenga, "Concurrency in Embedded Systems (Part 8): Using Files in Concurrent Linux Designs," *Circuit Cellar* 277, 2013.

circuitcellar.com/articlematerials