EMBEDDED IN THIN SLICES

Embedded File Systems (Part 3)

Designing Robust Flash Memory Systems

This is the third installment in a multi-part article series discussing Linux embedded file systems. This article focuses on how to design robust flash memory file systems for embedded systems.

By Bob Japenga (USA)

COLUMNS

My house has too much stuff—especially electronic junk. I have various electronic gadgets that I just don't throw away. I still have the hard drive from my 1990 PC. It has 10 MB of data on it! Just recently, I was about to throw away—I mean, recycle—a PCMCIA Wi-Fi card. The built-in Wi-Fi on my wife's laptop recently stopped working. In 5 min. I was able to get it working from my stash of electronic "junk." See, I am not a pack rat. I am a supply chain warehouse!

Part of the reason I have so much electronic junk is that as a culture we don't keep electronics very long. How long have you had your cell phone? How long did you keep your last cell phone? How about your DVD player? Did you replace it with a Blu-ray player? Those industries don't have to design their embedded systems to last more than a few years. Yes, a Blu-ray player is an embedded system that is probably running Linux.

As embedded designers, we need to create more things to last. In many cases, the software we write and the hardware on which it runs still needs to be running 10, 20, even 30 years from now. Of course we know that deep-space systems need that kind of lifetime. But is that true for the systems we design? As more low-tech devices become "connected," consumers are not going to want to swap out their toilet or furnace every three to four years. My company is currently working on a smart electric meter. When was the last time you swapped out your electric meter? They are supposed to last 30 years.

How can you make things last with something as complex as a Linux flash memory system? This article examines how to do that. Although many things contribute to a robust system, I will only discuss making the flash memory system robust. That is why this column is called "Embedded in Thin Slices."

THE PROBLEM: DATA RETENTION

In the first article in this series, I mentioned that most NAND flash devices are specified to only retain data for 10 years. A 4-Gb SPI flash we use specifies 20 years for data retention. What this means is that, independent of the wear-leveling issues I discussed in Part 1, if you just write the data once to these devices, you can only count on them to keep that data for 10 or 20 years, respectively.

For the bulk of our systems, the file system uses wear leveling (even on read-only data), which greatly extends the data retention life of these devices. It does this by constantly moving data around the flash memory. However, in one of our designs that should last more than 20 years, there are three other regions in our NAND flash memory (called partitions) that are not part of the file system and are usually never changed for the products's life. They are the bootstrap loader (u-boot), the u-boot environment variables, and the Linux kernel. Thus the data retention problem is real for these partitions. In 10 years, these regions will start dying.

Some flash memory characteristics can cause other errors that can prevent us from creating a robust 24/7 long-life embedded system. For example, both NAND and NOR flash memory can exhibit an error called bit flipping. This happens when a single bit is reported as reversed or is actually reversed

from the required state. I have already talked about how flash memory has limited write/erase cycles. In addition, flash memory can wear out due to too many reads! I didn't know that until I dug deep into a Micron Technology Technical Note (see Resources). In most systems, this is not an issue because flash data is usually seldom read. In the rare case where you are designing a Linux system that executes instructions directly out of the flash memory or one that repetitively reads data directly from the flash memory, you should be aware of the fact that flash memory wears out on reads as well as writes. Finally, all flash memory devices now come from the factory with bad blocks. The manufacturers guarantee that the first block is not bad, but after that, it is up to your flash memory manager to handle these bad blocks.

SOLUTIONS

Fortunately, there are workable solutions to each of these issues to extend a flash memory system's life. Applying error-correcting code (ECC) is the first line of defense and is widely used with flash devices. In addition, a general redundancy is relatively inexpensive to implement in hardware and software if you plan for it up front. Let's look at each of these and see what we can learn.

ERROR-CORRECTING CODE

ECC is used to detect and correct errors. One-bit ECC can detect and correct any single-bit errors. This is useful for correcting bit-flipping errors. Four-bit ECC can detect and correct up to four bits of error in a single page (typically 512 bytes). Both techniques accomplish this by storing redundant data on the page that enables it to restore the data to its original value if a bit failure should occur. Flash memory devices come with extra space on each page to store ECC data.

Our company designed a major system for a customer and missed the fact that the flash chip required four-bit ECC while the Linux kernel and u-boot for the chip we were using only supported one-bit ECC. After about two years and 20,000 units were deployed, systems started failing. This was a costly error that affected a large percentage (relatively) of the fleet before we were able to remotely update the software. Once four-bit ECC was implemented, the failure rate went basically to zero in our test lab.

We hope we see the same in the field. At the time our product was released, neither the Linux kernel nor the boot loader supported four-bit ECC. Thankfully, by the time the problem surfaced, these features were supported in the kernel and in u-boot. Thus we did not have to create these algorithms

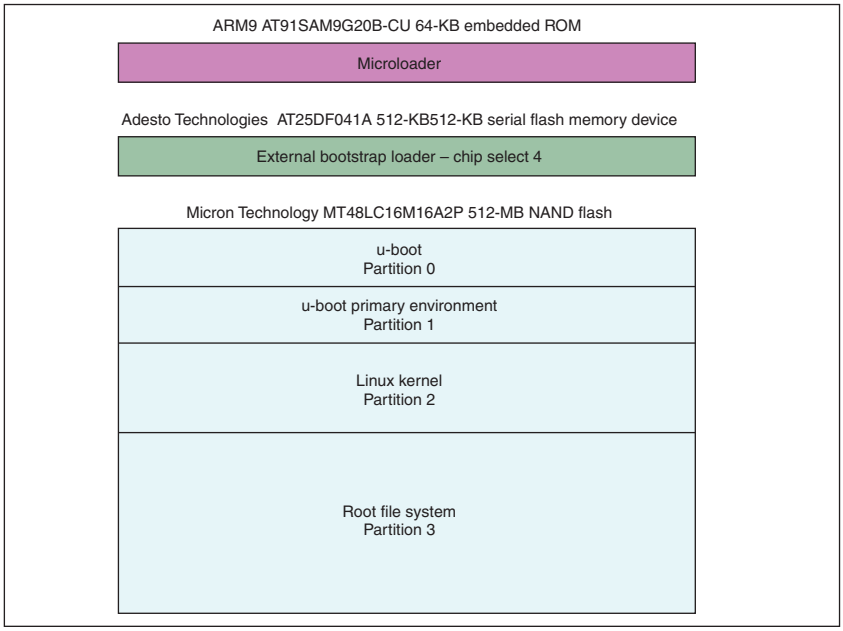

FIGURE 1
This is the memory layout of memory regions required to bring up one of my company's embedded Linux designs with no redundancy.

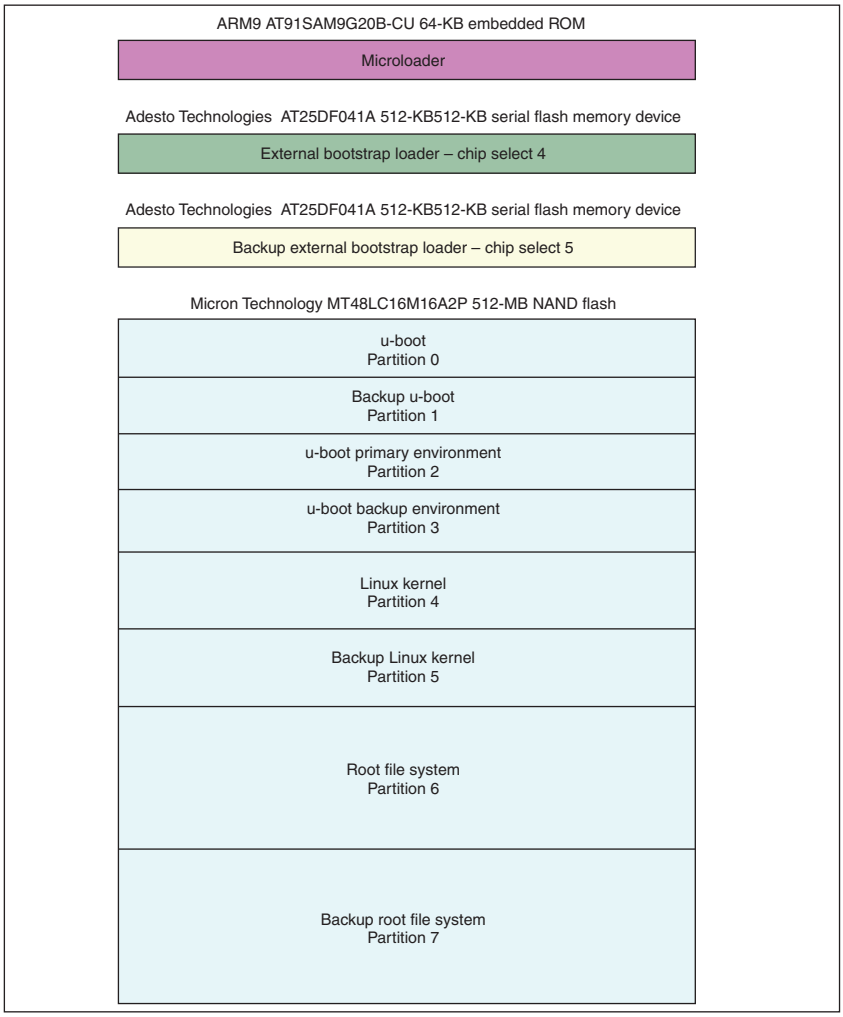

FIGURE 2
This memory layout creates the most reliable design by providing redundant memory regions for the external bootstrap, u-boot, u-boot's environment, the kernel, and the file system.

and write this code.

The moral of the story: Thoroughly read all datasheets and their associated errata for every device in your system. Know what your OS does underneath you. Even if Linux says: "We'll take care of it for you." Don't trust them! Tests run at the Jet Propulsion Laboratory showed that without ECC, one-bit errors started occurring at one fifth the life of a particular memory device.

ECC can be supported in either software or hardware if your processor and the OS support this feature. The ARM ARM9 processor we used on the previously mentioned product only supports one-bit ECC and is not useful with the memory chips we use.

REDUNDANT FLASH REGIONS

Before I describe a typical Linux system with redundancy that we design, let me describe how the system would start if it *didn't* have redundant flash regions. This description applies to an ARM9 architecture.

Figure 1 shows different types of memory and the roles they play at startup. The purple memory is ROM stored in the ARM9. The green memory is SPI flash memory stored in an external chip. The blue memory is the NAND flash memory stored in one or more chips.

At power-up, the system runs a microloader stored in ROM in the ARM9. This microloader can load a bootstrap loader into internal SRAM from any of several devices. Our system loads the bootstrap loader from a 4-Gb SPI flash on chip select 4. Once loaded, the microloader starts the bootstrap loader running out of internal static RAM (SRAM). The bootstrap loader loads our Linux boot loader (u-boot) from NAND flash Partition 0 into dynamic RAM (DRAM). Once complete, it transfers control to u-boot, which sets up the hardware as specified in the configuration environment block (stored in Partition 1 of the NAND flash). It then loads the Linux kernel from another NAND flash Partition (2) into DRAM. Control is then transferred to the Linux kernel, which mounts the file system in Partition 3 and then starts the various threads from applications stored on the file system.

There are two ways this can get you in trouble. Since the data retention lifetime is only 10 years for the NAND flash, Partitions 0–2 never get "wear leveled" and thus will start "failing" after 10 years. The life of these partitions can simply be extended by rewriting the partition. Squirreling away copies of these partitions (compressed or not) can enable the system to occasionally (once per year) rewrite them, which will dramatically extend the life of these regions. In the end, you end up with something like what is shown in **Figure 2**. Here the microloader attempts to load the bootstrap loader from chip select 4. If it fails to load, it will load it from another SPI device (yellow) on chip select 5. The loader, running in internal RAM, will attempt to load and run u-boot from either Partition 0 or Partition 1. Once running, u-boot will attempt to load the configuration from either Partition 2 or Partition 3. Finally, u-boot will attempt to load the kernel from either Partition 4 or Partition 5. All of these features are built into the open-source software for these packages.

Additional logic, which is not built into Linux, requires you to check the file system's integrity at start up to a previously defined "manifest" of CRC or MD5 sums for each file. Should there be any issues, the shadow file system partition (Partition 7) is mounted and the damaged files corrected.

Building the logic into each of the loaders to check data integrity and loading the alternate partition can enable the design to be even more robust than before. If one of the partitions is corrupted for any reason, the loader will choose the backup partition (or device) to boot. Once loaded, Linux can be flagged to rewrite the bad region. Out of the box, Linux even provides a means to write the bootstrap loader with a file image. This scheme also provides power-down protection for power outages that occur when the partition is being written (since it is not being used).

Since flash memory devices can cascade failures (i.e., failures in one page can cause failures in another page), the embedded designer needs to keep that in mind when designing a robust system. Sometimes this

circuitcellar.com/ccmaterials

RESOURCES

Y. Chen "Flash Memory Reliability: NEPP 2008 Task Final Report," NASA Jet Propulsion Laboratory, 2008.

B. Japenga, "Embedded File Systems (Part 1): Linux File Systems," *Circuit Cellar* 279, 2013.

———, "Embedded File Systems (Part 2): File System Integrity," *Circuit Cellar* 281, 2013.

Micron Technology, Inc., "NAND Flash Design and Use Considerations Introduction," Technical Note TN-29-17, Rev B 2010.

SOURCES

AT25DF041A 512-KB Serial interface flash memory device
Adesto Technologies Corp. | www.adestotech.com

ARM9 processor
ARM, Ltd. | www.arm.com

MT48LC16M16A2P single data rate synchronous dynamic RAM
Micron Technology, Inc. | www.micron.com

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.

can be solved by keeping your redundant data on a separate memory device rather than just a separate partition. Without knowing the physical layout of the chip, the designer doesn't know which partitions are physically next to each other.

REDUNDANCY AND ERROR-CORRECTING CODE

Some embedded systems are throwaways. We have designed some systems that are only run once. Other systems must last for more than 10 years. Flash memory devices are becoming more dense and less perfect. Not long ago, memory device manufacturers would not ship bad blocks in their devices. Now the devices routinely have many bad blocks. As embedded systems designers, we need to be aware how our flash memory systems work, what the failure modes are, and how they need to be reliably interfaced. Through redundancy and ECC you have added two tricks to your playbook. But just a thin slice!

Next time I will describe two alternate file systems available under Linux: a compressed ROM file system (cramfs) and a RAM file system.