Debugging Large Embedded Memories

7th July 2023

It is common in the embedded space to have small, simple storage devices accessed address-wise. Chips can be down to kB in size and there may not be enough space on the device to accommodate a file system. Or, at the other end of the complexity spectrum, we have been handed code using very large storage devices devoid of file system because the design did not call for it. Testing these systems can be long and complicated. For instance, we worked on a product which would generate a few hundred bytes of storage a few times a week. How can we test that it correctly copes with a very large storage area (128MB)? There are two linked answers to this.

The first is to abstract away data accesses through a dummy file. It doesn't do much. Its true purpose is to act as a barrier between the rest of the code and the memory accesses. When running code for the MCU target, we compile the code which uses its accesses methods. When running on the desktop, we use file system accesses. Say for example we have a system which reads one byte from an address. We could directly code the register access sequence into the method, or we could have a HAL function pass it through to another. Both signatures look identical:

```
int HAL_ReadMemory(uint8_t * buffer, uint32_t address, uint32_t
count)
```

It doesn't matter to the function calling it if the underlying memory access is through an MCU or the file system. It wants a memory area to hold the result, a return code, a start address, and the number of bytes to read. When we need to know on the desktop that we can read it back, we can use f_read to read from a file. Heck for many systems it doesn't even need to be that complicated – we can store the whole thing in a uint8_t array. What's 128MB to a modern desktop? And it's *fast*. It's RAM, after all! We can run pass/fail tests much more quickly than if we need to wait for an embedded erase cycle.

So, why bother?

Think back to our system that generates a few hundred bytes per week. How do we test the memory accesses are good? First up we simulate the maths on the desktop. We mock out the accesses and substitute our desktop methods. Now we can fill the dummy memory chip simulator with anything we want, under our control, as fast as the desktop can do it. That's way faster than waiting months or years for the real thing to catch up. We can "fill" the chip and check it rolls over correctly. We can test the erase routines get called as expected. We can see what happens when the data contents change, if that's what's important to us. It gives us significant test capability.

The second answer is a little more devious. What if we could do all of that, but on the real target? After all, the desktop doesn't have to deal with the same embedded memory limitations. It doesn't have to deal with the same sector sizes, or erase times. Best do that on the target. But how? The answer here lies in integration test commands. We require a back channel onto the target. We send the target a specially crafted message, and it runs its "store some data" method in a loop. OK, it's not as neat as doing it on the desktop – the flexibility of the data we can store is limited now by whatever will fit in the target memory.

But we are on the target. We have to deal with things like the power gulp during sector erase. We can't hide from erase times on chip. If there's a problem in the layout and it interferes with the address lines, this is one way to find it. Maybe interrupt priorities matter? Maybe you can expose a race condition? Here is one way to debug it. If we have a set of integration tests running on the

target, we can design tests to expose timing dependencies. We can link as much of the application as we want. With an appropriate bidirectional communication link, we can collect results and treat the runtime like a unit testing environment.

For example, referring back to the original 128MB storage IC, filling this memory “naturally” would have taken ~21 months. Using a desktop-centric unit testing approach, the test runs in around 13 seconds. As an integration test on the real target, the memory fills in around 4 ¾ minutes. We must wait for the memory to be written to the chip in real time, with various page write times. We must wait for the chip to carry out sector erases in real time. All of this happens in the background while the device executes its normal scheduler, so it is dependent on ISRs and conflicts just like anything else.

As an aside, we find in 2023 tools are rarely the problem. Yes, historically we used to see toolchain problems – the compiler wouldn’t always generate the same code, copious compiler-specific directives, quirks of the devices worked around in unclear ways. Modern tools (particularly Cortex tools) are reliable. What you get on the embedded target is almost always what happens on the desktop. The argument that there’s significant differences between the desktop and embedded runtimes is largely now discredited. Beware the vendors pointing out all the clever ways their tools are “good with C” – there’s probably a reason the marketing department chooses that wording.

Here we have presented two ways of debugging large memory devices in small embedded systems. Both have merits. Both are applicable in different situations. We have used both to improve the functionality of real systems. Sometimes it means we expand our test coverage cheaply. Sometimes we use it to expose integration test failures in the final product. Either way, we learn something new about our product and how it behaves under stressful conditions.