Devil's Details: Writing a sFLASH

Serial FLASH are a convenient and cheap way to add nonvolatile memory to a modern microcontroller. They originated in the SPI-interfaced EEPROMs, and still share some characteristics with them - the 8-pin form factor (although larger capacity sFLASH of course have larger die and need a larger package, and now they come also in various modern leadless packages), and partially also the marking - mainly the "25" prefix, having its roots in early pre-EEPROM/FLASH Intel marking (think of the 21xx RAMs and 27xx EPROMs) and the unpleasant habit of memory makers to give memory capacity in bits rather than bytes.

sFLASH also share the basic working principle with the 25-series EEPROMs: they use the ubiquitous SPI interface - Data In, Data Out, Clock and Chip Select (Latch) - and they use a simple *1 byte command - several bytes address - several bytes data* "protocol", with a polled "status" register to indicate write-finish. The most prominent difference comes from the NOR-FLASH nature of the memory array, i.e. sFLASH is erased in relatively large chunks of several kB to allow then bytewise write (several bytes within a page may be written at the same time, but the granularity is still one byte); whereas EEPROMs are fully bytewise erasable/writable. The requirement to erase in chunks is the cost of larger memory density, but by now we've learned to live with this. sFLASH these days comes in densities ranging from a couple of megabytes to a couple of gigabytes, the basic erase chunk (sector, although terminology varies - nobody uses "chunk" though, that's exactly why I use it here) is 4kBytes, endurances range from 10k to 100k and retention is around 10-20 years. The SPI speed is in tens to a hundred of MHz, and all newer sFLASH support a quad-SPI mode, where data can be transferred through 4 pins simultaneously.

Manufacturers tend to add features - e.g. various locks, security features, JEDEC-mandated and largely useless info structures, one-time programmable memory portions etc. - but luckily, sFLASH remain largely mutually compatible, at least in the basic operations.

This all sounds quite simple and indeed, it's not that much complicated. But, we all know too well, where is the devil lurking...

A couple of years ago, for a project, I used a 32Mbit (i.e. 4MByte) sFLASH - initially a ST/Numonyx/Micron[1] part, but after finding out it's a rather old piece featuring a rather large 64kByte minimum erase chunk, I quickly migrated to an AMD/Spansion/Cypress part[2]. Things went initially well - I wrote some basic "drivers", transporting data from/to SPI using DMA and handling the protocol's state machine in the DMA's end-of-transfer interrupt, upon which I wrote a simple file system, a setup-data-backup machine employing some simple wear-reduction, and a few other features needed for the given project.

However, after exhaustive testing, one weakness was revealed - upon rapid power-down-up cycles, the sFLASH locked itself preventing further writes. Careful reading of the DS revealed a requirement for a controlled power supply ramp, violation of which results in incomplete internal reset (and unforeseable consequences of course). Luckily, this could be solved in software (as is often the requirement these days) - sFLASH do have a software reset command. In fact, it's a couple of commands to avoid invoking the softreset inadvertently: 66h first "triggers" the reset and subsequent 99h "fires" the reset. So, these were quickly added as commands performed at the startup of the state machine.

But I wanted to make it more resilient. Remember, that the old EEPROM protocol, required, after writing, to perform a series of reads - polls - to a particular register - Status register, which in one bit indicated whether the write is still in progress or has already finished (in

1 Yes, also memory manufacturers like to play the "this week, who buys whom" game: http://www.efton.sk/t0t1/semic_change.htm

2 That particular part was then a brand new respin of their then low-mid-class offering in sFLASH. After some 5 years it's already obsolete. Let this serve as a warning, the memory business has no respect for conservative thinking.

sFLASH, the same mechanism is used not only for Write, but also for Erase). This Status register is still present in the sFLASH, but during the years, several bits were added - most of them being write locks to various areas of the memory[3]. It means, that the Status register is no longer one byte (it's 3 bytes in that particular part). Also, Status register is now not only a "dynamic read-only bit", but contains also bits which are stored in internal non-volatile memory cells; and then there are special commands which allow to write to these internal memory cells.

I realized that the lockup during testing mentioned above resulted from inadvertent rewrite of these lockbits. After further reading of the datasheet, I realized that Status registers contain a bitfield, which determines, whether the Status register itself can be written or not. It has several settings, one of which is "Status register write disabled until reset", and that was what I wanted. This bitfield spans two bytes - but that is no problem, the "Write Status Register" command, 01h, can write several consecutive bytes, in the same way as the memory write command does. There is also a "prefix command", 50h, after which the write to Status register does not write the non-volatile bits, but rewrites only the volatile "working copy" of the Status register (into which the non-volatile bits are copied after each reset). Both these went after the 66h/99h reset commands into the state machine startup. The datasheet mentions that the part needs some time after the softreset to start operation, a few us, this is solved using a trivial loopdelay which waits for cca 6.5 µs.

This all worked well, and after finishing the project it went into production and worked successfully in the field. Fast forward several years, and I work on a new project with similar requirements for non-volatile storage. Requirement for amount of stored data has "of course" increased meantime, so I decided to use a 128Mbit = 16MByte Adesto (ex-Atmel) part. I pulled the old sources, downloaded datasheets and carefully checked SPI/signals timing, and that all used commands are present also in the new part - yes, they are.

I then thought about ways to program the sFLASH - in the old project, I had first to program a bootloader into the mcu (an STM32) through SWD (using STLink), then load a special application through ETH/TFTP, which contained a webserver and that in turn allowed to program the sFLASH. That's a bit complicated, and won't suit the new application's development, where it's anticipated that sFLASH content will change very often. So, using the same "driver" as in the application, I wrote a loader which allows to read-erase-program sFLASH using STLink utility (many thanks to Clive on the STM32 Forum for helping with this). This worked well and I used it during several weeks with prototype A, and I also successfully modified it and tested on the old project's hardware, as well as on a different, unrelated hardware, with a different sFLASH.

As it often happens, prototype A had several issues, completely unrelated to sFLASH, so after some time I was given prototype B. I confidently run the sFLASH programming, without even looking at the output window, then programmed the mcu and... nothing. Debugging revealed that an open() function on the filesystem in sFLASH failed to open a file, the caller function then called close() which returned error as the file was not open, and the caller function "output" an error message to the yet uninitialized display and went to an infinite loop. Oh, the sFLASH must be faulty, a short on the prototype is nothing unusual, I said to myself, while moving the offending function call beyond the display initialization, so that at least the error message is visible. Confidently again then I started to inspect the board for shorts, bad solder joints, PCB errors, etc... just to find nothing. With diminishing selfconfidence I pulled the oscilloscope and started to look at the signals - but all of them were perfectly normal. At this PCB, the sFLASH sits next to the mcu, with very short traces and probing all signals to LA is an unpleasant task, so I decided to go stare-at-the-code mode.

The first thing to do was to try manual mode. In debugger, after running enough of the application that ports and SPI are initialized, it's just wiggling the /CS pin and writing/reading data to/from SPI's data register, that's easy enough to do manually. Reads return all FFs, but

3 So, most of the content of Status register is not status anymore, but the register's name remained.

what if STLink never wrote anything into the sFLASH and it's in its pristine state? Let's try to write - the application wouldn't have attempted writing until after complete startup, and it has never gotten there. Writing means to issue a Write Enable 06h and then the write itself - there's no need for the write-finished poll with manual writing, it's blazing fast compared to a human working through the debugger - but we can check the Write Enable Latch bit in Status register and it's there. And several bytes are written as commanded.

Okay so what's the issue? Shut down debugger, fire up STLink - yes it reads the sFLASH, as the bytes we just wrote witness. Okay so let's erase - it goes suspiciously fast, and... nothing happened, the written bytes are still there. Programming fails too - there are two options, the verify-while-writing fails immediately, and the verify-after-write fails, well, when it tries to verify after the write...

It took me around a hundred of experiments and two halfdays to find out what's wrong.

Turns out, in the Adesto part, Status Register still has 3 bytes as the older Spansion part, with roughly the same bit layout (at least the bits we are interested in here have the same function and same position); and there still is the 50h "prefix" command allowing to write only to the volatile "working copy". But, contrary to the Spansion part, the 01h command to write into Status Register takes only 1 byte; to write to other bytes of the Status Register one has to use different commands. And, as an undocumented (but not surprising) "feature", if one attempts to write two bytes after command 01h, subsequent writes to the memory are disabled.

After removing the "extra security" provided by writing into Status Register from the STLink loader, Erase and Write work in STLink as expected.

But how comes that write worked OK with the application? Remember, that before outputting the 50h/01h command combo, there was the 66h/99h erase commands combo, and after them a 6.5 µs loopdelay. Further reading of the Adesto datasheet revealed, that contrary to the Spansion part, here, after softreset, the part is unresponsible for approx. 30 µs. This means, that the 50h "prefix" command was simply ignored, and subsequently the 01h command, too - so it then couldn't have caused the write disable. However, while the same "driver" and state machine is used also in the STLink loader, there are subtle but important differences - the code runs from RAM rather than from FLASH and that's slower[4], and also STLink doesn't really allow interrupts in the loader, so the interrupt flag is polled in main() and then the ISR is called "manuall", this again adding up to the delays. So, the loopdelay calibrated on the application to be around 6.5µs may in the STLink loader quite well turn out to be long enough and close enough to the required 30µs so that the 50h/01h command combo is then executed (and causes the write disable).

But how comes that write from STLink worked OK on prototype A? I can only speculate here, but I believe that the 30µs is not a precise value and is given by an internal RC oscillator of the sFLASH. Thus, it may quite well be, that the sFLASH specimen on prototype A was a tad bit slower thus even the longer delay did not allow it to have the reset finished when the 50h command was sent to it.

Is this all my fault? Yes, definitely; but be sure the devil was there, too, looking and laughing...

4 In STM32F4, unintuitively, code running from FLASH with several waitstates imposed, may run faster than from RAM with zero waitstates; the former benefiting among other things from simple prefetch and jumpcache, a.k.a. ART accelerator. This is a complex issue with lengthy explanation; some simple benchmark results can be seen in the STM32 Technical Updates, Issue 1, pp41-45 https://community.st.com/s/question/0D50X00009Xkba4SAB/stm32-technical-updates