ChatGPT

Firmware Patch Plan for SN9C292B (Disable Boot OSD & Ensure Valid CRC)

Step 1: Identify OSD Boot-Enable Instructions in Firmware

- **Disassemble the firmware**: Load the 128 KB firmware dump (`firmware_backup.bin`) into IDA Pro using the Intel 8051/8032 processor module. This allows analysis of the on-chip 8032 microcontroller code. Use IDA's MCP plugin ("windsurf") to assist in pattern searching and disassembly navigation.
- **Search for OSD init pattern**: Scan the code for the known instruction sequence that enables the On-Screen Display (OSD) at boot. In 8051 assembly, the pattern is `MOV DPTR,#0x0B7x` followed by `MOV A,#0x01` and `MOVX @DPTR,A` [1]. This corresponds to writing the value `0x01` to memory-mapped OSD control registers in XDATA (addresses 0x0B75–0x0B77). Use IDA's search (or the MCP agent's assistance) to find all occurrences of bytes `90 0B 7x 74 01 F0` in the firmware.
- **Locate all OSD enable writes**: Expect **four** instances of this pattern in the firmware's initialization routine [2]. Each instance writes `0x01` to one of the OSD enable registers. Confirm the addresses and context of each occurrence in the disassembly. They should appear in early startup code (near the reset vector or initialization functions) where the firmware sets up the real-time clock and OSD. The identified addresses (file offsets in the binary) and original bytes are:

Firmware Offset	Target Reg	Original Instruction	Original Bytes
0x0004D0	0x0B77	`MOV DPTR,#0x0B77; MOV A,#0x01; MOVX @DPTR,A`	`90 0B 77 74 01 F0` [3]
0x000AC4	0x0B76	`MOV DPTR,#0x0B76; MOV A,#0x01; MOVX @DPTR,A`	`90 0B 76 74 01 F0` [3]
0x000AFE	0x0B77	`MOV DPTR,#0x0B77; MOV A,#0x01; MOVX @DPTR,A`	`90 0B 77 74 01 F0` [4]
0x004522	0x0B75	`MOV DPTR,#0x0B75; MOV A,#0x01; MOVX @DPTR,A`	`90 0B 75 74 01 F0` [5]

- **Plan the patch**: To **disable the OSD overlay at boot**, change the immediate value `0x01` to `0x00` in each of the above instructions. This will prevent the firmware from turning on the timestamp/OSD during initialization. We will not alter the instruction flow or timing—only the value written—ensuring minimal impact on the rest of the code. After patching, each of the four instructions will write `0x00` to the OSD control registers (effectively leaving OSD disabled) [6] [7].
- **Double-check in context**: Use IDA to verify the surrounding code logic. Confirm that these addresses are in the startup configuration routine (likely just after the reset vector). If using the windsurf/MCP environment, you can set breakpoints or simulate the code to ensure these writes occur only during init. The OSD registers 0x0B75–0x0B77 should not receive any `0x01` writes

after our patch. (Other OSD-related writes, e.g. writing values like 0x19, 0x1A, 0x1B to 0x0B77 during runtime, can be left as-is; with the enable flag 0x0B76 held at 0, those writes will have no effect [8].)

Step 2: Locate Firmware CRC/Checksum Verification Routine

- **Understand the bootloader's role**: The SN9C292B has a mask ROM bootloader that loads and verifies the external SPI flash content before handing control to the firmware. Typically, the boot ROM will calculate a checksum or CRC of the firmware image and compare it to an expected value stored in the image [9]. If the check fails, the ROM will not jump to the firmware (the device stays in bootloader mode). Our task is to identify how this integrity check works (e.g. CRC algorithm or simple checksum).
- **Identify checksum constants or code**: In the 8051 firmware dump, look for any constants or routines related to image length or polynomial. Common patterns include references to the image size (0x20000 bytes) or known CRC polynomials (e.g. 0x1021 for CRC-16). Since the bootloader's code is in ROM (not in the dump), the firmware binary itself might not explicitly contain the CRC routine. **Clue:** Check the end of the firmware image for a dedicated checksum value. Oftentimes the last bytes of the flash store the expected checksum [10] [11]. Use a hex viewer or IDA's loaded bytes to inspect the last 2–4 bytes of the 0x20000-byte file. If they are non-FF values (in an otherwise padding-filled region), that's likely the stored checksum.
- **Analyze prior attempt data**: Previous analysis indicates the firmware uses a **16-bit little-endian checksum at the end of the image** [11]. In particular, it was observed that modifying bytes in the image without updating a two-byte footer caused the device to stay in bootloader (brick) because the "CRC" check failed [12]. The **last 2 bytes (addresses 0x1FFFE–0x1FFFF)** appear to hold this checksum word [13]. The boot ROM likely computes the sum of all preceding bytes and compares it against this footer.
- **Confirm the algorithm (via IDA or testing)**: Use IDA (with the MCP scripting) to assist in confirming the checksum logic. One approach is to compute a checksum of the firmware file and see if it matches the stored value. For example, calculate a 16-bit sum of all bytes from 0x00000 up to 0x1FFD and compare it to the 16-bit value at 0x1FFE. In our case, the boot ROM uses a simple **additive checksum**: it sums all 0x1FFE bytes and uses the last two bytes such that the total sum modulo 0x10000 equals zero [14]. This is effectively a two's complement checksum (a common integrity check for firmware). If you have IDA Python or an external script, you can verify this by summing the known-good firmware's bytes: the result should be 0x0000 when the footer is included.
- **Leverage disassembled flasher code (if available)**: The reversed Windows flasher (`sonix_burn.exe`) or provided source may define a verify operation. Look for any mention of a checksum or CRC in that code. For instance, constants for a "VERIFY" command or code that reads 2 bytes at the end of the image. According to analysis, the flasher/bootloader protocol has a verify step that can use either a full read-back or a device-side CRC check [15]. In this camera's case, the device's ROM likely does the checksum internally and just reports pass/fail. Knowing this, we focus on reproducing the checksum calculation ourselves.

Step 3: Recalculate and Patch the Firmware CRC

- **Update the checksum after patching**: After modifying the OSD bytes, we must recalc the 16-bit checksum so that the firmware will pass the bootloader's integrity check. The rule is: **the sum of all 0x20000 bytes should equal 0 (mod 0x10000)** [14]. Equivalently, the 16-bit footer at 0x1FFE is chosen such that it equals the two-byte negative sum of bytes 0x00000–0x1FFD.
- **Implement a checksum calculator**: Create a small script (Python recommended for cross-platform use on Windows) to automate the fix. For example:

- Read the 128 KB firmware file into a byte array.
- Compute the 16-bit sum of bytes `data[0x00000:0x1FFE]` (i.e. all but the last two bytes).
- Calculate the two-byte checksum as `chk = (-sum & 0xFFFF)` (two's complement of the sum) [16].
- Overwrite the last two bytes `data[0x1FFE:0x20000]` with `chk` in little-endian order. This ensures that `sum(all_bytes) mod 0x10000 = 0`. (In practice, if the image was correct before patch, you can simply adjust the checksum by subtracting the exact differences introduced by your edits. But recomputing from scratch is straightforward and foolproof.)
- **Verify on known-good image**: Test the script on the original unmodified firmware dump to ensure it produces the same checksum already present. For example, on the original file the script should calculate a checksum equal to the existing footer (confirm no change). This validates that our method matches the manufacturer's. According to the prior analysis, the factory image's footer was already correct (and allowed it to boot) [17].
- **Integrate with IDA/MCP workflow**: If using the IDA MCP agent, you can invoke the checksum fix within the IDA environment after patching bytes. For instance, use an IDAPython snippet or windsurf automation to run the above steps on the in-memory image and update the last two bytes. This keeps the entire patching workflow in one place (but be mindful to only calculate on the **file bytes**, not IDA's disassembly which may treat data as code). Alternatively, run the external Python script on the binary after exporting the patched file from IDA.

Step 4: Validate CRC Region Boundaries and Logic

- **Confirm the covered region**: It's critical to use the correct range when computing the checksum. Based on reverse-engineering clues, the checksum covers **all bytes except the checksum itself**. That means offset `0x00000` through `0x1FFD` (inclusive) are summed, and the 2-byte result is stored at `0x1FFE–0x1FFF` [18]. No portion of the image is excluded except the checksum word. Validate this by inspecting the disassembly or memory map: the image is exactly 0x20000 bytes (131,072 bytes) [19], which fits this scheme. There are no extra metadata blocks outside this range.
- **Check for multi-byte alignment**: The checksum is 16 bits, so it aligns with the 2-byte footer. Ensure little-endian ordering when writing the bytes (the least significant byte goes to 0x1FFE, next to 0x1FFF). If you see an obvious pattern in the stored checksum (for example, it might be 0xFFFF in an empty image, or some predictable value), that can corroborate the little-endian assumption.
- **Look for constants in firmware**: Although the boot ROM does the check, sometimes the firmware or updater contains references to the image size or checksum for double-checking. For instance, the Windows flasher might use a constant 0x1FFE in its code (or 0x20000 image length) when performing verify. If you have the disassembled flasher code open in IDA, search for `0x1FFE` or `0x20000` in the x86 code. Any occurrence can reinforce our understanding of the boundary. (The summary of tools indicated that 0x20000 was indeed used as the flash size in the Kurokesu utilities [20].)
- **Boundary testing**: It's wise to test the checksum logic on a small scale if possible. For example, take a short segment of bytes and manually compute a two-byte sum to see that our method holds. However, given we trust the prior analysis and the official process, focusing on the full-image checksum is sufficient.

Step 5: Prepare the Patched Firmware (128 KB, Corrected CRC)

- **Apply the OSD patches**: Using IDA Pro's patching features or a hex editor, flip the four identified bytes from `0x01` to `0x00` at offsets 0x4D0, 0xAC4, 0xAFE, and 0x4522. In IDA, you can right-click on the byte in the Hex View and edit it, or use the built-in assembler (since the change

doesn't alter instruction length). If using the MCP agent, you could even command it to perform these byte modifications sequentially.

- **Recompute and insert checksum**: Run the checksum script from Step 3 on the modified binary. This will update the last two bytes of the file to the new correct value [16]. Ensure the script reports a sensible checksum (not something outlandish like 0x0000 every time—unless the sum of the body truly happened to be 0xFFFF, which is unlikely). The script provided in earlier research output will print the new checksum and a SHA-256 of the file for verification [21].
- **Verify file size and format**: The patched firmware file **must remain exactly 131,072 bytes (128 KiB)** [19]. After patching and checksum correction, double-check the file size. No bytes should be added or removed; we are only altering existing bytes. If using IDA, ensure you **do not** inadvertently change the file length (IDA's `Edit->Patch Program->Apply to input file` will preserve size). If using a separate script, it should assert the input size is 0x20000 and keep it unchanged [22].
- **Hash and compare**: Compute a hash (e.g., SHA-256) of the patched file and compare it to the original. Aside from the four OSD bytes and the 2-byte checksum footer, there should be no differences. It's good practice to record these hash values and offsets changed. This helps in future to confirm if a camera is running the patched firmware or if a read-back matches exactly (which is crucial for trust in the flashing process) [23].
- **Table: Summary of Bytes Changed** (for record-keeping):

Offset	Original Byte → New Byte	Description
0x0004D4	`0x01 → 0x00`	OSD enable bit for reg 0x0B77 (init) [24]
0x000AC8	`0x01 → 0x00`	OSD enable bit for reg 0x0B76 (init) [25]
0x000B02	`0x01 → 0x00`	OSD enable bit (second write to 0x0B77) [26]
0x004526	`0x01 → 0x00`	OSD enable bit for reg 0x0B75 (init) [27]
0x1FFEE–0x1FFF	`0xYY 0xZZ → Checksum`	16-bit checksum footer (recomputed) [14]

(Note: Offsets are file positions; 0xYY 0xZZ represent whatever original checksum was, replaced by new values.)

- **Preserve a backup**: Keep an untouched copy of the original firmware dump and the patched version with a clear naming scheme (e.g., `firmware_backup_original.bin` and `firmware_backup_osd_off.bin`). This allows quick recovery if needed and helps in diffing files if something goes wrong.

Step 6: Inject and Test the Patch in IDA (Safe Rebasing & Simulation)

- **Rebase if necessary**: The 8051 code in the firmware might logically be mapped to address 0x0000 in the 8032's code space. In IDA, ensure that the segmentation/base is correct so that addresses in code correspond to the physical offsets (often IDA will load a raw binary at offset 0 by default for 8051, which is fine). If the disassembler had used a different base (e.g., some analysts load at 0x8000), rebase the program to 0x0000 for consistency with actual addresses. This avoids confusion when patching specific offsets.
- **Apply patches in IDA**: With the MCP (IDA Pro AI interface), you can script the patching process: locate the four addresses and change the bytes. For instance, use an IDAPython snippet to patch and then re-run IDA's analysis to ensure the instructions now show `MOVX @DPTR,A` with A loaded with 0x00. Verify that no unintended changes occurred (the disassembly around those

addresses should be identical except for the immediate value). IDA will not automatically recalc the checksum for you – that we handle externally or via script as described. If you haven't already, update the last two bytes in IDA's Hex View to match the new checksum as well, so IDA's memory matches the final binary exactly.

- **Leverage windsurf plugin for simulation**: The windsurf/MCP setup can be used to **simulate execution or at least examine state** around our patched code. For example, set a breakpoint (or use a conditional breakpoint script) on any `MOVX @DPTR,A` that targets 0x0B75–0x0B77, then "run" the firmware in the IDA emulator (if available) to ensure A now contains 0x00 when those instructions execute. While a full 8051 emulator might not be built into IDA, the MCP agent could assist by symbolically executing or using an external 8051 emulator to run through the reset sequence. The goal is to confirm that after reset vector, the OSD registers remain zeroed out. If such dynamic testing isn't feasible, rely on the static analysis: our patch is small and well-defined, so static verification (checking the instructions and cross-references) is usually sufficient.
- **Dry-run CRC check**: If possible, simulate what the boot ROM would do for the checksum. For instance, you could write a small IDAPython routine (or use the earlier script within IDA) to sum the bytes of the IDB and assert that the sum is 0. This is a sanity check that you've inserted the correct checksum. Given the simplicity, an external Python is often easier.
- **Maintain alignment and timing**: We specifically chose a **minimal patch** (changing data written, not removing code) to avoid any side effects on code flow. There's no need to reassemble or insert NOPs in this plan, but if you ever consider NOPing out the instructions entirely, be careful: on 8051, replacing an instruction with 0x00 is actually a NO-OP (`NOP` opcode is 0x00), so one could neutralize an instruction by overwriting opcodes with 0x00. In our case, writing 0x00 to the immediate achieves the desired effect without altering code length or timing. Thus, no further rebasing or jump fix-ups are needed. The patched firmware remains binary-aligned with the original.
- **Export the final binary**: Once satisfied, use **IDA's "Save file" or patch export** to get the final binary, or take the output from your Python script. This is the file you will flash to the device. Double-check its hash one more time and maybe even run a diff against the original to ensure only the expected six bytes have changed (4 OSD bytes + 2 checksum bytes). Having this final confirmation prevents accidents like an incomplete patch or an extra byte.

Step 7: Flashing the Patched Firmware and Recovery Considerations

- **Flash the new firmware**: Use the official Kurokesu firmware update tool (or the SONiX TestAP `--sf-w` command) to write the 128 KB patched file to the camera's SPI flash. On Windows, the Kurokesu GUI (flash_public.zip) is straightforward: it will handle rebooting the device into bootloader mode and programming the flash via USB [28]. If using the command-line `SONiX_UVC_TestAP` utility, invoke it with the `--sf-w` option (provide the binary and device path as needed) as shown in the documentation. Ensure the tool reports a successful write and verify.
- **Power-cycle and observe**: After flashing, unplug and replug the camera. On boot, the OSD should now remain **off by default**. The overlay with date "2013" will no longer appear on the video feed at startup. You can confirm by querying the camera's UVC Extension Unit for OSD status (e.g., using `--xuget-os` or the Windows UVC XU tool); it should report 0 (disabled) by default [29]. This indicates the patch is effective. Also verify the camera enumerates properly as a UVC device with the correct PID (0x6366) and no USB errors.
- **In-case of failure (device doesn't boot)**: Don't panic – the SN9C292B is designed with a fallback **ROM bootloader** that makes it *almost impossible to truly brick* the device [30] [31]. If the patched

firmware had an issue (e.g., a wrong checksum), the camera will simply not run that firmware and instead enumerate in **bootloader mode**. Signs of this include the device showing up with a different USB PID or as an unknown device, and not presenting the usual video interfaces.

- **Recovery via USB bootloader**: The Kurokesu firmware tool can recover the device when it's in bootloader mode (sometimes called ISP mode). Simply open the tool and reflash the original known-good firmware binary. The tool will detect the bootloader and program the flash over USB [32]. According to Kurokesu, the CPU's ROM will always be there to accept new firmware, so you can restore the camera even after a bad flash. Once re-written with a good image and power-cycled, the camera should function normally again.
- **Forcing bootloader mode**: If the device does not enumerate at all (rare, but could happen if USB descriptors are corrupted or the firmware crashes the USB interface), you can manually force the bootloader. **Method**: Disconnect the camera, then physically disable the SPI flash and reconnect USB. For example, hold the flash chip's CS line high or remove its VCC, so the SN9C292B can't read the firmware and thus *reverts to boot ROM*. The camera should then appear on USB (often with a default VID/PID like 0x0C45:**** that differs from 0x6366) [33]. At that point, run the flasher to write a good image, then restore the flash connections and reset the device.
- **Direct SPI programming (last resort)**: If USB recovery fails or is not an option (say the USB interface is unresponsive), you can remove or connect to the SPI flash chip directly and program it with an external programmer (e.g., CH341A USB programmer or Raspberry Pi SPI) [34]. Ensure to write the full 128 KB image. This bypasses the SN9C292B entirely. After re-soldering or reconnecting the chip, the camera should boot that firmware.
- **Testing on multiple units**: If you have more than one SN9C292B-based camera, test the patched firmware on each (when possible) to ensure consistent results. Because we only changed the OSD default, there should not be unit-specific differences. Still, verifying that each camera accepts the firmware and boots without OSD will build confidence. Use USB Tree Viewer or Device Manager to confirm each camera enumerates correctly (no "Code 10" errors). If any unit fails to boot, recheck if its firmware was identical to the one patched (some cameras might have different firmware versions). The procedure to fix would remain the same: adjust the correct bytes and checksum for that firmware version.
- **Maintain a recovery image**: Keep the **original firmware dump** safe as a recovery image. Its checksum/footer is known good. Label it clearly. In case of any issues with future patches, you can always flash back the original to get to a baseline.
- **Conclusion & next steps**: With the OSD disabled by default and the checksum corrected, the cameras should now power on with no overlay and no boot issues. Future firmware modifications (for example, more extensive changes to remove OSD code entirely or other feature tweaks) should always be followed by the checksum update step as we've implemented – that two-byte fix-up is what separates a successful boot from a "partial brick" [35]. Using this workflow (IDA + MCP for analysis, careful byte patches, and Python for checksum and verification), you can confidently develop and apply firmware patches on Windows without needing WSL. Always test incrementally and utilize the built-in safety net of the SN9C292B's bootloader for recovery.

References and Footnotes

- SN9C292B Datasheet and Register Map – for understanding OSD registers and MCU features.
- Prior attempt logs & analysis (ChatGPT-assisted) – provided crucial insight into OSD control addresses and checksum mechanism [2] [11].
- Kurokesu Wiki – notes on the firmware update tool and the "hardcoded bootloader" recovery method [31] [33].
- StackOverflow discussion on bootloader CRC – explains storing checksum in image for verification [9].

- *GitHub: jhdkrwmc/sonix_flasher2* – reversed Sonix flasher code (for reference on USB commands and potential CRC usage in host software).
- *GitHub: Kurokesu/C1_SONIX_Test_AP* – UVC control tool source (useful for understanding runtime commands like `--xuset-os`).
- *GitHub: mrexodia/ida-pro-mcp* – IDA MCP plugin used to integrate the analysis process with an AI assistant (facilitated disassembly queries and scripting in this plan).

---

1 2 8 23 24 25 26 27 sonix_patch_attempts_summary.md
file://file-7tG5RmB2izfTNKpaA5XeqH

3 4 5 6 7 11 12 13 14 15 16 17 18 19 20 21 22 28 29 30 31 32 33 34 35 prev attempts 2.txt
https://drive.google.com/file/d/1fvvBjFUbwtDlMyw5UptZSCmJ6McxQ7e3

9 10 c - How does a bootloader know the "expected" CRC value? - Stack Overflow
https://stackoverflow.com/questions/70598589/how-does-a-bootloader-know-the-expected-crc-value