AMATEUR RADIO · SEPTEMBER 10, 2026

Debugging, Disassembly and Troubleshooting Tools for Baofeng DM-1701 Custom Firmware Development

A working toolchain for DM-1701 firmware work: STM32CubeIDE headless (or the no-Xpresso patch kit) to build, OpenOCD plus a reset-capable probe on five PCB test points to debug, SEGGER RTT or the debug build's serial console for output, radio_tool or the Python DFU loader to move images, Ghidra at 0x0800C000 to disassemble, and an SDR plus MMDVMCal for RF.

24 min read · 41 sources · Updated Sep 10, 2026

2026 Baofeng DM-1701 OpenGD77 OpenRTX STM32 debugging firmware reverse engineering toolchain

CONTENTS

What the debugger is actually looking at 2
The bootloader is not the radio 3
Building an image you can actually debug 3
Debug output from a running radio 4
Hardware debug: SWD, probes and the five wires 5
Flashing, dumping and the DFU toolset 6
Recovery when the radio will not boot 8
Disassembly and reverse engineering 8
Emulation and platform tests before touching the radio 9
Radio-level test and measurement 10
Codeplug, calibration and memory-map tooling 11
Which firmware tree, which debugging story 12
Bottom line 13
Sources 13

The DM-1701 is an STM32F405 handheld built from the same silicon as the TYT MD-UV380 and Retevis RT-84, and every tool below assumes that. Two firmware families matter for development — OpenGD77 and its SMS fork on one side, OpenRTX on the other — and they have almost nothing in common in how you observe what the firmware is doing. Getting the toolchain right starts with understanding which of the radio's two USB personalities you are talking to, because one of them lies to you.

0x0800C000	851,968	48 kB	3 up / 3 down	±200 Hz
Firmware load address 48 kB bootloader below	Codeplug image size (bytes)	Bootloader dump `20.86 kB used`	SEGGER RTT buffers `1024-byte up buffer`	Hotspot Rx capture window

What the debugger is actually looking at

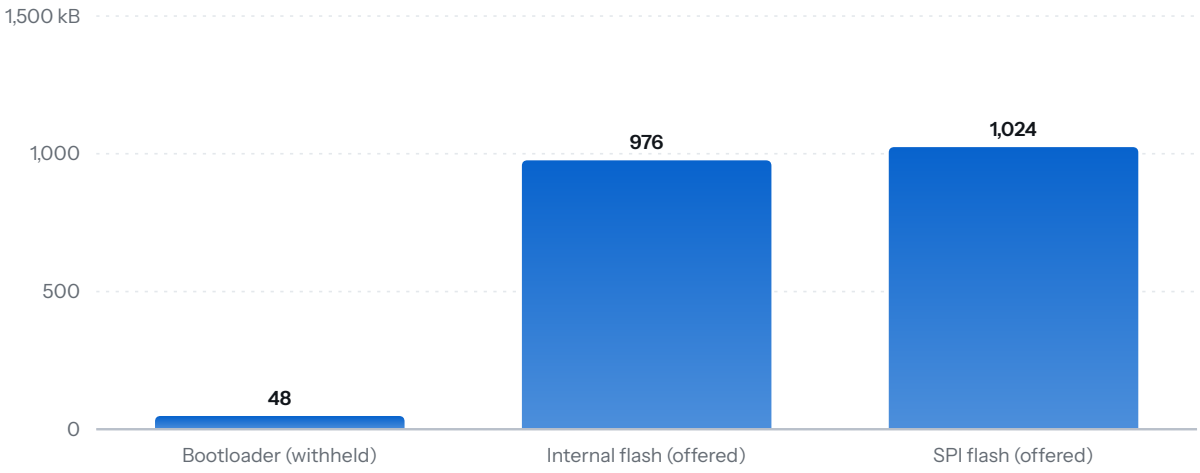
The MCU is an **STM32F405VGT6** — Cortex-M4, 1 MB internal flash, 192 KB RAM, LQFP100 — paired with an **AT1846S** RF chip and an **HR-C6000** DMR baseband that the CPU addresses over SPI [1] [31]. The codeplug is not in the MCU; it lives in an external SPI flash chip, which is why the radio has to describe two memories rather than one [1] [31].

It describes them itself, and that description is the single most useful fact in this document. Over DFU, a DM-1701 returns:

```
@Internal Flash   /0x0800C000/01*016Kg,01*064Kg,07*128Kg
@SPI Flash Memory /0x00000000/16*064Kg
```

Internal flash starts at **0x0800C000**, not at the chip's own 0x08000000, because the three 16 K sectors below it hold the bootloader and the radio is deliberately not offering them [1].

What the DM-1701 offers over DFU versus what the chip holds



Source: DFU interface string returned by the radio — 01*016Kg + 01*064Kg + 07*128Kg offered, three 16 K sectors below 0x0800C000 withheld [1].

A hardcoded STM32F40x map covering the whole chip from 0x08000000 will happily erase those withheld sectors on a radio that is trying to say no [1]. Read the layout from the device, not from a datasheet.

The bootloader is not the radio

The second thing to internalise is that a TYT-family radio speaks DFU in two quite different ways[1]:

- **Its own firmware**, when switched on normally. This is what implements the vendor commands — codeplug read/write, clock, reboot — and it is what the stock CPS talks to.
- **Its bootloader**, entered by holding PTT and the button above it. This flashes firmware and nothing else.

Under the bootloader only, codeplug writes are accepted and stored nowhere, erases are silently discarded, reboot is ignored, reads above 0x40000 return `0xff`, and the clock register returns the same four bytes as another register[1]. Every one of those looks like success or like empty memory. The same radio also reports a completely different memory map in the two modes — 976 kB of internal flash at 0x0800c000 under the bootloader, versus **48 kB at 0x08000000**, a 128 MB flash and 1 MB of SPI under stock firmware[1].

> CAUTION
>
> With OpenRTX installed there is no firmware DFU at all, only a serial port. Every vendor-specific command then fails quietly rather than erroring, which is the single most common source of "my codeplug write did nothing" reports[1].

Building an image you can actually debug

The mainstream path is **STM32CubeIDE** with the `DM1701_FW` build configuration[2][3]. Before any clean build you must run `prepare.bat`, which checks the source tree's integrity and calls `MDUV380_firmware/tools/codec_cleaner.exe -C` inside `application/source/linkerdata` to regenerate the codec and linker data[3]. If that step fails, further builds "usually do not make much sense" — check that `application/source/linkerdata` and `codec_cleaner.exe` both exist[3].

Outputs land in `MDUV380_firmware/Debug` as `.elf` and `.bin`, plus a cleaned image named for the target (`OpenDM1701.bin`, `OpenDM1701_Japanese.bin`, `OpenRT84.bin`)[3]. Eight configurations exist: `MDUV380_FW`, `JA_MDUV380_FW`, `DM1701_FW`, `JA_DM1701_FW`, `RT84_FW`, `JA_RT84_FW`, `MDUV380_10W_PLUS_FW`, `JA_MDUV380_10W_PLUS_FW`[2][4]. One naming trap is recorded in the source's own notes: not everything that looks like a configuration name in `.cproject` is one — the file also contains debug launch entries[3].

For scripted builds the IDE has a headless mode, which is how you reproduce a build without opening the GUI[4]:

```powershell
& "C:\ST\STM32CubeIDE_1.16.0\STM32CubeIDE\stm32cubeidec.exe" `
  -nosplash `
  -application org.eclipse.cdt.managedbuilder.core.headlessbuild `
```

```
    -data "C:\...\.headless-workspace" `
    -import "C:\...\V2\MDUV380_firmware" `
    -cleanBuild "MDUV380_firmware/DM1701_FW"
```

`BUILD_ALL.md` in the same repository carries the equivalent one-liner for all eight configurations, plus a PowerShell loop that runs the whole set [4]. The version string is generated with `git rev-parse --short HEAD`, so if git is not on `PATH` the build still succeeds but the version falls back to `UNKNOWN` [3].

Two workarounds exist if Eclipse is unwelcome. JG1UAA's **opengd77-noxpresso** is a patch kit that builds with a plain `arm-none-eabi` toolchain — "no f**king slow MCUXpresso required" — at the cost of English-only strings and everything forced to `-Os`, because newlib and picolibc are larger than the vendor's redlib [26]. The patch kit is based on an experimental Makefile that older OpenGD77 releases shipped and that recent releases abandoned [26]. The older MK22-based `open-ham/OpenGD77` tree, which predates the STM32 port, instead requires MCUXpresso plus the NXP MK22 SDK and a Windows `codec_dat_files_creator.exe` that emits `codec_bin_section_1.bin` and `codec_bin_section_2.bin` from official firmware 3.1.1 [25]. That utility now tries to download a codec blob from a repository that no longer exists, so those two files have to be obtained from another source or from a previous build [25].

A note on toolchain drift: the current release **R20260504** exists only because the AMBE `codec_interface` assembler code would no longer link under a newer GCC, and had to be rewritten to use C function pointers; the same change was back-ported to the MK22 version [27]. If you are building older source against a current toolchain, that is the failure you will hit, and it is worth knowing that the fix also allowed `codec_interface.c` to be compiled with `-Os` [27].

Debug output from a running radio

There are three ways to get information out of a running DM-1701, and the choice depends on which firmware tree you are in.

A debug build with a serial console. The SMS fork's author documents the contributor workflow directly: "Download putty and flash the debug build, connect your radio with your pc and start a raw serial connection on the right COM port. You will need this to debug, to see what the hotspot transmits" [9]. The same tree ships a `V2_DEBUG` configuration alongside the production ones [2] [10]. This is the cheapest instrumentation available. The USB identity it presents is worth recording, because it explains why a plain terminal works: with OpenGD77 installed the radio enumerates as `idVendor=1fc9, idProduct=0094`, product string **"OpenGD77 Transceiver"**, manufacturer `www.opengd77.com`, and the kernel binds it as a CDC-ACM device creating `/dev/ttyACM0` [39].

SEGGER RTT. The same debug tree bundles the SEGGER RTT sources under `MDUV380_firmware/SeggerRTT`, configured with `SEGGER_RTT_MAX_NUM_UP_BUFFERS (3)`, `SEGGER_RTT_MAX_NUM_DOWN_BUFFERS (3)`, `BUFFER_SIZE_UP (1024)`, `BUFFER_SIZE_DOWN (16)`,

`SEGGER_RTT_PRINTF_BUFFER_SIZE (64u)` and `SEGGER_RTT_MODE_NO_BLOCK_SKIP` as the default mode [6]. RTT carries `printf` output over the same two SWD wires the debugger already uses, so no UART is sacrificed and no special debug build is required. The bundled configuration is SEGGER's stock file, and the vendor's own note in the header is explicit that the source should not be modified if you want to stay compatible with the RTT protocol and J-Link [6].

Screen capture. `gd-77_screen_grabber.py` in the OpenGD77 tools tree saves the radio's display to PNG, taking `-d` for the device, `-s` for a scale factor, `-o` for the output filename and `-f` for the foreground colour [43] [44]. Crude, but it is the only way to record UI state without a camera.

Hardware debug: SWD, probes and the five wires

The radio has no debug connector. Recovering a radio that will not enter the bootloader requires opening it: the OpenGD77 FAQ says it is possible "via a hardware debugger dongle, e.g. JLink Edu, but this requires opening the radio and attaching 5 wires to test points on the PCB" [8]. The patched-bootloader documentation is more specific about what goes where: **reset must be connected, not just GND, SW_CLK and SW_IO**, and "some of the cheaper JLink clones don't seem to have a reset connection" [10].

That same page describes the bootloader patch itself, which is the cheapest hardware-level diagnostic in the ecosystem. On the GD-77 the signature bytes sit at 0x7F800 and on the DM-1801 at 0x3800; if they are erased or corrupted the bootloader refuses firmware updates, and the symptom is diagnostic — the green LED flashes instead of staying lit, and the moment SK1 and SK2 are released the bootloader runs the main firmware anyway [10]. The patch that fixes it is a single instruction change, "changing a BEQ to a B assembler instruction," and the patched images are 16 kB [10]. Note that these are MK22-family bootloaders and are not the DM-1701's, so this is a technique to borrow rather than a file to flash.

The debug tree carries a working OpenOCD configuration, which is the fastest route to a first connection [5]:

```tcl
source [find interface/stlink.cfg]
set WORKAREASIZE 0x8000
transport select "hla_swd"
set CHIPNAME STM32F407VGTx
set STOP_WATCHDOG 1
set CLOCK_FREQ 8000
reset_config none
set GDB_PORT 3333
source [find target/stm32f4x.cfg]
```

It is the vendor-IDE-generated file and is worth reading for what it tells you: the ST-Link transport is `hla_swd`, the watchdog is stopped when the core halts, the debug clock is 8 MHz, and GDB attaches

on **port 3333** [5]. The chip name is nominally an F407 rather than the F405 fitted; the two share a target definition, and this is exactly the kind of detail that wastes an afternoon if you assume it is a typo and correct it [5]. `reset_config none` is also significant — it means the config expects reset to be driven by the probe rather than by the target's own reset network, which lines up with the "connect reset or it will not work" warning above [5] [10].

ST's application note AN6382 is the reference for SWD itself, covering the protocol, target-selection sequence and multi-drop constraints, which matters if you are probing alongside another device or need to understand why a probe will not attach [32]. Historically the whole path was opened by Travis Goodspeed's **PoC||GTFO 10:8** write-up, which broke the MD-380's readout device protection, dumped all of protected memory and patched the recovery bootloader so it would stop setting the protection flag [33]. The MD-380's CPU has both JTAG and a ROM bootloader, and both are protected by that RDP feature — the same architecture applies to the F405 in the DM-1701 [33].

KEY FINDING

Reset, SW_CLK, SW_IO and GND to five test points, plus a probe that actually drives reset. A J-Link clone without a reset line may appear to connect and still not recover a bricked radio [8] [10].

Flashing, dumping and the DFU toolset

These are the tools that put bytes on the radio and take them off again.

TOOL	PLATFORM	WHAT IT DOES	DM-1701 NOTES
radio_tool (C++)	Linux/macOS/Windows	Flash, wrap/unwrap, `--get-status`, `--dump-reg`, `--reboot`, `--dump-bootloader`	Fw read ✖, Fw write ✔, Fw wrap ✔ [29]
radio_tool (Rust port)	Cross-platform	Adds codeplug read, device-described memory layout, write verification	Verified on hardware [1]
opengd77_stm32_firmware_loader.py	Python 3	Standalone DFU flash, no CPS needed	1,077 lines; DM-1701 is platform 2 [30]
OpenGD77 CPS → Extras → Firmware Loader	Windows	Canonical flash path with donor-file merge	The sanctioned route [7]
dmr.tools, ClearDMR	Browser (WebUSB)	Web flashers; ClearDMR also reads a backup first	Chrome/Edge 89+ only [35]

The Python loader is the most portable option and worth reading rather than just running. It is a DFU state machine with `__DFU_DETACH` through `__DFU_ABORT`, a **4000 ms** USB timeout, `BLOCK_WRITE_SIZE = 1024`, a `FWPlatformOutput` enum where **DM-1701 = 2** and `MD_UV380 = 1`, and a 256-byte `MD9600_ENCODE_CIPHER` table [30]. It descends from `pydfu.py` and md380tools, and its

error codes are a usable diagnostic in themselves: `-1` missing firmware file, `-2` wrong SGL format, `-3` unsupported format, `-4` file too large, `-5` unknown device model, `-6` unable to connect, `-7` command-line parsing error, `-8` unable to patch[30]. A `__DFU_STATE_DFU_ERROR` from the state machine means the radio is wedged in a DFU error state, which is what the clear-status request exists to fix — a loop that calls abort instead will hang forever, because abort cannot clear that state[1][30].

The C++ `radio_tool` writes DM-1701 firmware and wraps images to a load address — `./radio_tool --wrap -o wrapped.bin -r DM1701 -s 0x0800C000:main.bin` — but cannot read firmware back and has no codeplug support at all[29]. The Rust rewrite's hardware-verified list is the best available statement of what the DFU interface can actually do on this radio[1]:

- identify, memory layout and register reads — works, and the registers match the C++ byte for byte
- write firmware — works; it flashed OpenRTX 0.4.4 and the radio booted
- **read codeplug — works: 851,968 bytes, parsed, two reads byte identical**
- write codeplug under stock firmware — works, read back with no difference
- dump bootloader — works: **48 kB, 20.86 kB used**, a valid vector table (the C++ marks this macOS-only)
- write stock Baofeng V02.06 back over OpenRTX — works, **869 kB in nine sectors**

Two further details from that work are worth carrying forward. Cross-checking against independat tooling, a codeplug read came out **851,966 of 851,968 bytes identical** to what dmrconfig reads from the same radio, with the two differing bytes being live state the radio changes between reads[1]. And the tool that reads the memory layout from the device rather than compiling it in exists precisely because "the same radio describes itself differently depending on what is running"[1].

One firmware-format trap belongs here because it costs real time. The TYT container's `Read` path does not decipher the payload, and `WriteFirmware` sends the payload exactly as it sits in the file — still enciphered, because the bootloader deciphers it itself[1]. A tool that deciphered on parse and then flashed would send a valid ARM vector table instead of the enciphered payload, take all 223 blocks, acknowledge every one, report success, and not start[1].

For Windows, a one-time driver swap is needed to use the browser-based flashers: the shipped STM32 DFU driver blocks WebUSB, so **Zadig** is used (Options → List All Devices → "Digital Radio in USB mode" → **WinUSB v6.1.7600.16385**) to replace it[35]. DFU mode is confirmed by the LED flashing red/green after holding **PTT + SK1** at power-on[35]. On Linux, the release packages include udev rules so the radio can be reached without root — `OpenGD77_20260131_linux_udev.zip` sits alongside the 69 kB Python tools archive and the 2.7 MB source archive in the R20260131 `sources_and_tools` folder[28].

Recovery when the radio will not boot

The OpenGD77 FAQ gives an escalation ladder, and it is worth following it in order: check the battery with a meter, remove and reinsert the battery, remove the battery and hold **SK2** at power-on, then remove the battery and hold **SK1 + SK2** at power-on, at which point the radio should enter the bootloader with the **green LED illuminated** [8]. From there, reload the firmware via the CPS. The FAQ adds one non-obvious caution: “Do not transmit any RF from other radios while reloading the firmware, because this can corrupt the firmware data being sent via the USB cable” [8].

BOOT COMBINATION	EFFECT
SK2	Settings reset
SK2 + Down	Clear boot melody, image and themes
SK1 + 0	Safe boot
SK1	Boot when Safe Pwr-On is enabled
SK1 + 5	Clear NMEA logs
#	Enable Voice Prompt L3

STM32-based handhelds including the DM-1701 [7].

The FAQ is blunt about the end of the ladder: “If the radio won’t even enter the bootloader, and the green LED does not illuminate, the radio probably has a hardware fault, or perhaps the MCU / CPU program ROM has been completely erased or corrupted” [8]. That is the point at which the five wires and a probe become the only option.

> **CAUTION**
>
> Two failure modes deserve respect. First, a radio in bootloader mode silently accepts codeplug writes, erases and reboots — read back and compare, which is the only thing that catches it. Second, because the C++ tool erases using a hardcoded chip map from 0x08000000, it can erase the bootloader sectors the radio is holding back; the corrected version refuses any image outside the device’s own reported layout, and there is a test using the layout string above as it came off the radio [1].

Disassembly and reverse engineering

The md380tools project is the reference method, even though it targets the MD-380 and MD-390 — a different MCU from the DM-1701 — rather than this radio. Treat it as a recipe, not a binary to flash.

The Ghidra workflow is documented step by step [11]. Unwrap the firmware image first (`md380-fw --unwrap`), then load it into Ghidra with the language set to `ARM:LE:32:Cortex` and a **loading address of 0x0800C000** — the images begin after the 48 kB bootloader [11]. Run the auto-analyzer

with defaults and you get hundreds of functions identified without effort; the exceptions are the initial stack address and RESET vector in the vector table at the start, which the analyzer misses because they are never called by the application's own code[11].

Then add the RAM: a core dump loaded at **0x20000000** lets you follow pointers that the flash alone cannot resolve[11]. Two corrections make the decompiler behave — mark the flash page as non-writable so it stops treating literal pools as mutable variables, and import symbols with Ghidra's `ImportSymbolsScript.py`, fed by md380tools' `symbols2ghidra.py`[11]. One genuine advantage over IDA and Binary Ninja is noted: because Ghidra knows the CPU from the language setting, it knows the entire image is Thumb2, with no classic ARM instructions and no virtual register to configure[11].

The MD-380 emulator is the other half of the method, and its build recipe doubles as a map of the memory layout: `objcopy -I binary firmware.img --change-addresses=0x0800C000 --rename-section .data=.firmware` and the same for a RAM core at `--change-addresses=0x20000000`, linked with `--just-symbols` from a symbol file[14]. The result is a 32-bit ARM/Linux executable that runs on amd64 through qemu and is used mainly for AMBE encode and decode against `.amb` and `.wav` files[14]. It will not run on macOS or Windows[14].

Elsewhere in the toolset: `md380-tool dmesg` dumps the firmware's log buffer over USB, `md380-tool calllog` prints incoming calls with names, and `md380-tool spiflashid` reports the external flash type and size — the last of which is worth knowing because some VHF hardware shipped with only 1 MB of SPI flash instead of 16 MB, which changes what can be stored[12]. The applet that makes this possible is worth studying as an instrumentation technique in its own right: it hooks the UPLOAD and DNLOAD handlers on an unused DFU block to add its own protocol extensions, keeps its own global and static variables, and routes `printf` into a heapless dmesg buffer rather than to `stdio`[13]. It also exposes the stock firmware's graphics routines, with the recorded caveat that other threads will draw to the screen and may overwrite your changes because the vendor's graphics semaphores are not understood[13].

For quick work, `r2 -a arm -b 16 -e asm.cpu=cortex` puts radare2 into Cortex-M Thumb mode, and `objcopy` with `--change-addresses` and a section rename will bind a firmware image into an ELF. The DM-1701-specific gift is the `sources_and_tools` folder in the R20260131 release, which shipped the firmware sources alongside a Python tools archive and Linux udev rules[28].

Emulation and platform tests before touching the radio

OpenRTX has the strongest story here, and it applies even if you never run OpenRTX on the radio.

The **Linux build** (`meson setup build_linux` then `meson compile -C build_linux openrtx_linux`) validates UI changes without hardware, requiring SDL2, Codec2 and readline[15]. It can also demodulate a pre-recorded baseband file: the build runs a file source driver that reads a raw **16-bit little-endian mono 24 kHz** file from `/tmp/baseband.raw` on loop[17]. The bundled test samples are at 48 kHz and must be decimated first:

```bash
sox -r 48000 -e signed-integer -b 16 -c 1 tests/unit/assets/M17_test_baseband.raw \
    -r 24000 -e signed-integer -b 16 -c 1 /tmp/baseband.raw rate -v
./build_linux/openrtx_linux
```

The documented result is a receive screen showing callsign **OPNRTX**, and the page notes that a named pipe at that path fed by GNU Radio should work in principle [17]. It also warns that because the Linux build has no audio driver, received audio cannot be heard — the test confirms the demodulator, not the speaker path [17].

Platform tests are standalone binaries flashed to the radio in place of the firmware — build with `meson configure -Dtest=<name> build_cm4`, then `meson compile -C build_cm4 openrtx_mduv3x0_flash` [16]. The directory contains `calib_read.c`, `nvm_dump.c`, `stm32f405_flash.c`, `display_test.c`, `keyboard_test.c`, `mic_test.c`, `tonegen.c`, `osp_demo.c` and `gpio_demo.c`, and you can write your own from any of them as a starting point [16] [20]. The OpenRTX documentation describes one of their primary uses precisely: “they can be used for example to dump the calibration data of a radio” [16]. Note the header on the flash test — “It will mess-up the data stored in flash, use at your own risk!!” [22] — which is worth reading literally: that test erases from 0x8040000 over an area of 0x40000, deliberately writes over it, and then checks that sixteen illegal read, write and erase permutations are correctly refused [22]. It is a driver conformance test, not a safe thing to run on a radio you care about.

Building with `-Dasan=true` turns on the address sanitizer for buffer-overrun hunting, at the cost of speed [15]. Finally, **rtxlink** is OpenRTX’s host-to-radio debug protocol and is worth knowing even as a specification: SLIP-framed packets with a protocol ID (0x00 stdio redirection, 0x01 CAT, 0x02 FMP, 0x03 data transfer) and a CRC-16 using the CCITT polynomial **0x1021** [18]. Its CAT protocol includes a **peek command that reads a 32-bit word from any address**, which is a memory inspector over a serial cable, and an operating-mode set that includes DMR and M17 as well as FM [18].

Radio-level test and measurement

When the bug is in the RF path rather than the C, you need instruments.

An SDR is the primary one, and the community’s reasoning for preferring it over a counter is specific: “most counters only seem to resolve to 1kHz, which isn’t good enough when calibrating multiple radios to work with the same hotspot, as the Hotspot Rx window is only around **± 200Hz** at most” [36]. On the DM-1701 specifically, one user’s calibration on the radio’s own screen resolved a transmit-to-hotspot failure, with the practical procedure being to zoom the SDR spectrum as far as it will go, note the deviation from centre frequency on a radio that does work with the hotspot, then use the radio’s calibration page to match it [36].

DSD-FME decodes DMR, P25, NXDN, YSF and M17 from an SDR or a recorded audio stream and serves as an independent check that what your firmware transmits is actually well-formed; it also

accepts OP25 capture files, which makes it useful against recorded test vectors rather than only live RF[34][34]. **MMDVMCal** is the BER tool of choice against a hotspot: set the frequency with `E` (nine digits, so 441.5 MHz is `441500000`), enter DMR BER mode with `b`, then raise and lower the offset with `F` and `f` until the reported bit error rate is minimised; `q` quits and prints the offset to paste into MMDVMHost's RXoffset and TXoffset[37]. Commands are case-sensitive and nothing is transmitted over the network while it runs[37]. Running the DM-1701 in OpenGD77 **hotspot mode** — the General Options → Hotspot setting, `Off` / `MMDVM` / `BlueDV` — turns the radio itself into the modem and is the configuration many testers use for DMR and SMS work[7].

On the radio itself, the **calibration screen** is a diagnostic tool as much as a tuning one. Press **SK2 + Up/Down** to page between sections[7]. **Cal Freq** selects the calibration point and there are **5 points for VHF and 8 for UHF**, with the firmware interpolating between them — so calibrating 2 m means adjusting 136.0, 145.5 and 155.0 MHz together, because 144–148 MHz spans multiple points[7]. **Cal Power** holds four power levels per point[7]. Power calibration needs a power meter and a dummy load: connect the radio to the meter, the dummy load to the meter, select the point and power level, key the PTT and adjust until the meter reads the target[7]. Pressing Green exits keeping values in RAM only; **SK2 + Green** persists them, and the guide notes you can exit with Green, test on the VFO, and return to the calibration screen with the adjusted values still in use[7].

Codeplug, calibration and memory-map tooling

The codeplug is the radio's configuration memory, and its layout for this device is documented in libdmrconf[24].

REGION	START	END	ALIGNMENT
Segment 1	0x002000	0x040000	0x400 (1024 bytes)
Segment 2	0x110000	0x1a0000	0x400 (1024 bytes)

DM1701Codeplug, libdmrconf — codeplug structure reverse engineered, not published[24].

dmrconf (part of the qdmr suite, built on the same libdmrconf) reads and writes codeplugs from the command line. It detects the radio automatically, writes YAML or a binary image according to the file extension, and can decode a manufacturer's own `.rdt` file with `-m --radio=uv390`; `--verbose` gives detailed logging, and omitting the output filename plus `--yaml` or `--bin` lets you pipe the decoded codeplug into another program[23]. Since version 0.8.1 the decode command has accepted manufacturer CPS files[23]. The manual also carries a section the authors themselves label "Danger zone" — worth reading before pointing it at a radio you need[23].

Two DM-1701-specific offsets will save real time, both of them off-by-a-few-bytes problems in third-party tooling rather than in the radio[1]:

- **Contact records begin at image offset** `0x70014`, while dmrconfig looks at `0x70000`. It therefore finds no contacts and every digital channel fails to resolve its talkgroup. The records themselves are fine, on a correct **36-byte stride** from that offset[1].
- **A zone has an A list and a B list in different places** — A channels in the zone record at `0x149e0`, B channels in the zone extension at `0x31000`. Fill in only one and the radio shows "Unprogram" on the other VFO[1].

CHIRP has no DM-1701 driver; the request (issue #7175) was closed as a duplicate of an older one, and the note on it records that while the radio "does have its own software, the porting of data between CHIRP and that program is very much a manual process"[38]. **editcp** covers the MD-380/MD-390/MD-UV380/MD-UV390 and the Alinco DJ-MD40, edits `.rdt` and md380tools `.bin` files, converts between the two, and can upload over USB — but the DM-1701 is not on its supported list, so treat it as adjacent rather than applicable[21][40].

Which firmware tree, which debugging story

	OpenGD77 (incl. the SMS fork)	vs	OpenRTX
Serial console debug build	Yes — V2_DEBUG tree	·	No
SEGGER RTT bundled	Yes	·	No
OpenOCD config in tree	Yes	·	Not shipped
Linux / emulator build	No	·	Yes — openrtx_linux
Host peek/poke over serial	No	·	Yes — rtxlink
Hardware mod for the main mode	No	·	Yes, for M17
Primary target mode	DMR + FM	·	M17 + FM

OpenGD77 is the tree with the developer infrastructure: headless STM32CubeIDE builds, a debug configuration, SEGGER RTT, an OpenOCD config, Linux udev rules and a maintained user guide[2][3][6][28][7]. The SMS fork is smaller and messier but is where the active debugging happens, and it is the only tree that ships a debug build with a serial console[9][2]. Its author is candid about the state of it — "The repo is a mess. I don't have time to clean things up now" — and it documents the fixes that came out of that debugging, including moving SMS alert handling out of the interrupt handler so it runs from the main loop, correcting the HR-C6000 SPI frame loading order, and fixing USB capture[10]. Its feature documentation is precise enough to be useful as a test specification: maximum **64 characters** per message, ASCII only, **8 messages** of inbox storage, **8 of sent**, **10 quick-text templates** with 16-character titles, and a fixed set of user-visible failure messages (`DMR only`, `Select private call`, `SMS busy`, `Empty message`, `Message too long`, `ASCII only`)[41].

OpenRTX gives you a real emulator, flashable test binaries and a documented memory-access protocol, but pays for it with a soldering iron. The DM-1701 is listed on OpenRTX's own M17 page as a radio the firmware implements M17 on, alongside the MD-380/390, MD-UV380/390, CS7000-M17 and others, with the note that "for the M17 protocol stack to work properly, a hardware modification is required" and an explicit "**Proceed at your own risk**"[19]. The project rates the DM-1701 mod's difficulty as **Medium** and links the M17-Project's modification guide[19]. OpenRTX's own development-status table lists DMR RX, TX and SMS as future work — all currently unsupported on every platform — so the DM-1701's OpenRTX path is an M17 path, not a DMR one[20].

KEY FINDING

If your goal is DMR and a working debug loop, stay on the OpenGD77 tree and use `V2_DEBUG` for instrumentation. Choose OpenRTX only if you specifically want M17 — and budget for the mod, for a debugging workflow that is emulator-and-serial rather than DFU-based, and for the fact that once OpenRTX is installed every vendor DFU command fails silently[1][19][20].

Bottom line

A working DM-1701 debug setup is roughly six pieces, and they are all obtainable. Build with **STM32CubeIDE headless** or the **noxpresso** patch kit using the `DM1701_FW` configuration. Attach an **ST-Link or J-Link with reset wired** to five PCB test points and drive it from **OpenOCD on port 3333**. Get text out through **SEGGER RTT** or the debug build's raw serial console on the radio's CDC-ACM device. Move images with **radio_tool** (Rust port) or the standalone **Python DFU loader**, keeping **dmrconf** for codeplug work and **Ghidra** at 0x0800C000 for disassembly. Test without hardware where you can, using **OpenRTX's Linux emulator and platform tests**. Measure the RF side with an **SDR plus MMDVMCal**, not a frequency counter.

Two traps will cost you the most time, and both are about trusting a success message. The bootloader **silently discards** codeplug writes, erases and reboots while reporting success — always read back and compare. And because internal flash for firmware purposes starts at **0x0800C000** rather than the chip's 0x08000000, a tool using a hardcoded chip map can erase the bootloader sectors the radio is deliberately holding back. Ask the radio what it has before you write to it.

SOURCES

1 radio_tool Rust port — hardware-verified DM-1701 operations, DFU memory layout, bootloader vs firmware
https://github.com/v0l/radio_tool/blob/main/rust/README.md

2 opengd77-dm1701-ci — DM-1701 headless build, V2_DEBUG tree, SEGGER RTT
https://github.com/868meshbot/opengd77-dm1701-ci

3 OpenGD77 V2_DEBUG README — build configurations, prepare.bat, build outputs
https://github.com/868meshbot/opengd77-dm1701-ci/blob/main/V2_DEBUG/README.md

4 BUILD_ALL.md — exact headless build commands for all eight configurations
https://github.com/868meshbot/opengd77-dm1701-ci/blob/main/BUILD_ALL.md

5 MDUV380_firmware.cfg — the OpenOCD debug configuration shipped with the source
https://raw.githubusercontent.com/868meshbot/opengd77-dm1701-ci/main/V2_DEBUG/MDUV380_firmware/MDUV380_firmware.cfg

6 SEGGER_RTT_Conf.h — RTT buffer configuration used by the OpenGD77 debug build
https://github.com/868meshbot/opengd77-dm1701-ci/blob/main/V2_DEBUG/MDUV380_firmware/SeggerRTT/Config/SEGGER_RTT_Conf.h

7 OpenGD77 User Guide — calibration screen, hotspot mode, boot key combinations, DFU entry
https://github.com/LibreDMR/OpenGD77_UserGuide/blob/master/OpenGD77_User_Guide.md

8 OpenGD77 FAQ — recovery ladder, J-Link and five test wires, bootloader recovery
https://opengd77.org/viewtopic.php?t=7

9 SMS experiment github repo — PD0FR's debug-build and raw serial console workflow
https://www.opengd77.org/viewtopic.php?p=190

10 OpenGD77 patched bootloaders — reset line requirement, J-Link clone caveat
https://www.opengd77.com/downloads/patched_bootloaders/

11 Reversing MD380 firmware with Ghidra — load address, language, core dump, symbol import
https://github.com/travisgoodspeed/md380tools/wiki/GHIDRA

12 md380-tool / md380-dfu reference — dmesg, calllog, spiflashid, wrap and unwrap
https://github.com/travisgoodspeed/md380tools/blob/master/docs/md380tool.md

13 MD380 Tool Applet README — dmesg buffer, printf hooks, DFU protocol extensions
https://raw.githubusercontent.com/travisgoodspeed/md380tools/master/applet/README.md

14 MD380 Emulator — firmware at 0x0800C000, SRAM at 0x20000000, qemu-arm
https://github.com/travisgoodspeed/md380tools/wiki/MD380-Emulator

15 OpenRTX compilation instructions — Miosix toolchain, meson targets, radio_tool, asan
https://openrtx.org/#/compiling

16 OpenRTX platform tests — building and flashing standalone test binaries
https://openrtx.org/#/tests

17 OpenRTX Linux testing — emulator, baseband file demodulation, sox decimation
https://openrtx.org/#/linux

18 rtxlink — OpenRTX host communication protocol, SLIP framing, CAT peek command
https://openrtx.org/#/rtxlink

19 OpenRTX M17 — DM-1701 listed as supported with a required hardware modification
https://openrtx.org/#/M17/m17

20 OpenRTX development status — supported devices and per-mode feature table
https://openrtx.org/#/dev_status

21 OpenRTX tests/platform — calibration read, NVM dump, flash, display, keyboard, mic tests
https://github.com/OpenRTX/OpenRTX/tree/master/tests/platform

22 stm32f405_flash.c — destructive internal-flash driver test for the STM32F405
https://github.com/OpenRTX/OpenRTX/blob/master/tests/platform/stm32f405_flash.c

23 dmrconf manual — reading, writing and decoding codeplugs, manufacturer file support
https://static.dm3mat.de/qdmr/manual/ch07.html

24 libdmrconf DM1701Codeplug — codeplug segment addresses and alignment
https://static.dm3mat.de/qdmr/libdmrconf/classDM1701Codeplug.html

25 OpenGD77 BUILD.md — MCUXpresso and MK22 SDK toolchain, codec binary extraction
https://github.com/open-ham/OpenGD77/blob/main/BUILD.md

26 opengd77-noxpresso — building OpenGD77 with a plain arm-none-eabi toolchain
https://github.com/jg1uaa/opengd77-noxpresso

27 OpenGD77 R20260504 changelog — codec_interface rework for a newer GCC
https://www.opengd77.com/downloads/MDUV380_DM1701/Firmware/20260504_ebd7100bb4c930e6ff0c24202474b5b72e7d6beb/changelog.txt

28 OpenGD77 R20260131 sources and tools — udev rules, Python tools and source archive
https://www.opengd77.com/downloads/releases/MDUV380_DM1701/R20260131/sources_and_tools/

29 radio_tool (C++) README — supported radios, wrap/unwrap, register and bootloader dump options
https://github.com/v0l/radio_tool/blob/main/README.md

30 opengd77_stm32_firmware_loader.py — standalone DFU flash tool with DM-1701 platform support
https://github.com/jamarju/opengd77/blob/fix_vol/opengd77/MDUV380_firmware/tools/opengd77_stm32_firmware_loader.py

31 MD380 hardware notes — HR-C5000 baseband, SPI flash, display and CPU port usage
https://www.qsl.net/dl4yhf/RT3/md380_hw.html

32 Reverse Engineering the Tytera MD380 — CPU, RDP, DFU variants and flash dumping
http://md380.blogspot.com/2016/01/reverse-engineering-tytera-md380.html

33 ST AN6382 — SWD multi-drop for STM32 with ST-LINK
https://www.st.com/resource/en/application_note/an6382-introduction-to-swd-multidrop-for-stm32-with-stlink-stmicroelectronics.pdf

34 DSD-FME — digital speech decoder for DMR, P25, NXDN, YSF and M17
https://github.com/lwvmobile/dsd-fme

35 MMDVMCal instructions — DMR BER test mode and hotspot offset tuning
https://www.m7spi.co.uk/mmdvmcal/

36 DM-1701 transmit to hotspot fails — SDR versus counter for frequency calibration
https://www.opengd77.com/archive/viewtopic3f1d.html?f=19&t=4765

37 ClearDMR web flasher — DFU mode entry, Zadig WinUSB driver swap, firmware backup
https://cleardmr.com/

38 CHIRP issue #7175 — Baofeng DM-1701 new-model request, closed as duplicate
https://chirpmyradio.com/issues/7175

39 Editcp — codeplug editor for the MD-380 and MD-UV380 families
https://www.farnsworth.org/dale/codeplug/editcp/

40 CPS on Linux — USB vendor and product IDs and the ttyACM0 CDC-ACM device
https://www.opengd77.com/archive/viewtopicf80f.html?f=12&t=2091

41 HOW_TO_USE_SMS.md — SMS feature limits and on-radio menu structure
https://github.com/868meshbot/opengd77-dm1701-ci/blob/main/DOCUMENTATIE/HOW_TO_USE_SMS.md

Researched and written with deepseek/deepseek-v4-flash-0731, from the sources listed above. Figures are as reported by those sources. Exported November 29, 2026 from https://intel.innerpulse.net/doc/debugging-disassembly-and-troubleshooting-tools-for-baofeng-dm-1701-custom, where the current version lives.