DRACO7RT · FLIGHT SOFTWARE GUIDE · 28 OCTOBER 2026

Flight Software: Writing and Debugging Code

How a change to the cFS flight software is made, tested, built, flown and proved on DRACO7RT, and how the running vehicle is debugged

The flight software is real NASA cFS, cross-compiled for a PowerPC e500 running RTEMS 6, booted on an emulated board. Every change goes through the same loop: edit, unit tests in seconds, static analysis, a three-minute build and reboot, proof on the ground system, and a requirements trace. A debugger reaches the live vehicle.

Slides 2 to 5 explain what the flight software is, the shape of an app, the two ways to run it and the loop. Slides 6 to 17 are the work instructions: set up (with the VS Code task names), write, unit-test, analyse, build and restart, verify, debug on the vehicle, debug natively, trace. Slides 18 and 19 are two worked examples from this project. Slides 20 and 21: troubleshooting, the rules, and working with Claude Code in the loop.

WHAT IT IS

Real cFS on a real RTOS on an emulated flight computer

The stack, bottom up

- **QEMU** board model **draco-s706**: MPC8548-class PowerPC e500, big-endian, eTSEC Ethernet, serial UARTs, NAND, a GDB server on port 1234
- **U-Boot** fetches the image **files/sp0** by TFTP at every boot; the start-up script and tables come from **/cf**, an NFS share of **root/cf_rtems/**
- **RTEMS 6**, the real-time OS; the cFS OSAL and PSP sit on it
- **cFE**: Executive, Software Bus, Events, Tables, Time
- **Apps**, all statically linked into one image

The apps

- **Mission apps, written here:** ACS (attitude control, MEKF, mode logic), NAV (GPS ingest and spoof gate), IMU (sensor ingest and validation gate)
- **cFS apps, configured here:** LC watchpoints and actionpoints, SC stored sequences, CF file transfer, SCH scheduler, HK, HS, CS, DS, FM, MD, MM
- **Lab I/O:** CI_LAB uplink, TO_LAB downlink
- Mission config and tables: **cFS/draco_rtems_defs/**; each app in **cFS/apps/<app>/**

Numbers

- 3 mission apps, **2,671** lines of flight C
- **5,143** lines of unit tests for them
- **83** requirements, 83 traced, 0 untagged
- Build and reboot: about **3 minutes**
- Unit tests: **10 s** pure module, **30 s** app layer

The vehicle is 10.5.5.38 on the Docker network fogi_net; CI_LAB listens on UDP 1333, TO_LAB sends to the downlink tee at 10.5.5.201, which feeds COSMOS and Yamcs. The console UART is logged to files/console.log; front door › Demo control › Serial console tab (job pane) shows it live.

THE SHAPE OF AN APP

One pattern for every mission app: a pure module, an app shell, a table, tests and tags

nav_app.c, top to bottom (the template)

- **NAV_Init**: register with Events, create the command pipe, subscribe, register and load **NAV.CFG_TBL**, get its address, send the init event
- **NAV_AppMain**: the run loop, wait on the pipe with a timeout, dispatch by message id, housekeeping once per second, **NAV_ManageTable** once per housekeeping period
- **NAV_ProcessCmd**: one case per function code, length checked, counters, events
- **NAV_ProcessGpsFix → NAV_AcceptFix / NAV_RejectFix**: the data path
- **NAV_Tick**: time-driven state (stale detection)
- **NAV_ValidateCfgTbl**: the table validation callback; its rules are in a comment the ground tool mirrors

The rules the pattern enforces

- **nav_gate.c is pure:** no cFE calls, numbers come in as a struct argument, so it compiles with plain gcc and is tested in seconds
- **Constants live in the table** (nav_tbl.h): gains, thresholds, timers; a #define is only for structure (message ids, sizes, periods)
- **Every implementing line carries its requirement tag**, e.g. `[dRT-NAV-005]`; nav_app.c has 13; the trace generator reads them
- **Message layouts** in nav_msg.h are the contract with the ground; COSMOS definitions are audited against the compiled binary, not the header
- **Two test layers**: test_nav_gate.c (pure) and nav_app_tests.c (app shell under the cFS ut_assert framework with cFE stubs)

ACS and IMU follow the same file set: <app>_app.c/.h, the pure module (acs_mekf.c, imu_check.c), <app>_msg.h, <app>_tbl.h, fsw/tables/<app>_def_cfg.c, unit-test/ with run_coverage.sh and run_app_coverage.sh. A new app or a new tunable follows this or is rejected in review.

TWO WAYS TO RUN THE SAME SOURCE

The vehicle is the primary target; the native build is there when you need threads

	Vehicle (QEMU, PowerPC e500, RTEMS 6)	Native (Linux process, every app a thread)
Build	`run/draco.sh rtems fsw` then `sync_cf` (always: the dev /cf is NFS, not the baked image)	`make -C cFS SIMULATION=native BUILDTYPE=debug MISSIONCONFIG=draco O=build_native install`
Run	`tools/dev/vehicle-restart.sh`, about 3 minutes	`cFS/build_native/exe/cpu1/core-cpu1`, seconds
New code in	Rebuild and reboot only; apps are statically linked (RESTART_APP resets state, keeps the old code)	`tools/dev/es_app.py reload ACS`: apps are .so files, hot reload
Debugger	Cross GDB through QEMU's server on 127.0.0.1:1234; sees the CPU, one thread, no task list	gdb on the process: `info threads`, per-task breakpoints, thread apply all bt
A stop	Halts the whole vehicle: telemetry pauses, COSMOS goes stale, timers catch up on continue	Stops the process; nothing else notices
Fidelity	The real RTOS, byte order, boot chain and board; what the evidence is taken on	Linux, little-endian, POSIX OSAL; byte-order and board effects do not show
Use it for	Every verification; GDB snapshots, breakpoints and watchpoints on real behaviour	Long debugging sessions, thread-level work, a fast edit-run loop before the cross build

Both builds come from one source tree and one mission configuration (draco_rtems_defs and the tracked symlink draco_defs/tables). Anything proved natively is confirmed on the vehicle before it counts.

THE LOOP

Eight steps, in this order, for every change

1 · Understand
Read the app, its requirements and its tests. Name the requirement the change serves, or propose one

2 · Edit
Pure module stays pure; constants go in the table; tag the lines

3 · Unit tests
Both layers on the host, seconds. RESULT: PASS and the coverage floors hold

4 · Static analysis
cppcheck; any new finding is fixed or dispositioned in the deviation log

5 · Build and restart
vehicle-restart.sh: cross build, sync_cf, reboot, wait for OPERATIONAL, ground re-map, 42 back. 3 min

6 · Verify on the vehicle
Init event in the console, telemetry on COSMOS, --verify; behaviour changes get the mission test on a fresh vehicle

7 · Trace and gate
gen_trace.py: 0 untagged, 0 unknown; dvr_gate.py: PASS

8 · Record and review
What, why, how verified in claude-notes.md; a person reads the diff; commit only when asked

Debugging sits beside the loop, not in it: when step 6 shows something you cannot explain, slides 14 to 16 attach the debugger; the fix then re-enters at step 1. The three project skills (/fsw-change, /fsw-debug, /cosmos-screen) run these steps the same way every time.

WORK INSTRUCTION · 1 SET UP

VS Code over Remote-SSH onto the server; the repository brings its own tasks and debug configurations

Do

1. Install VS Code on your PC with the free **Remote-SSH** and **C/C++** extensions (Extensions view, Ctrl+Shift+X)
2. Press F1 (Command Palette), run **Remote-SSH: Connect to Host...**, enter your login on the server. Then **File › Open Folder...** and open **/home/efsi/draco7_0**. Not /var/www: the stack scripts and Docker need the home path
3. Menu bar **Terminal › Run Task...** (or F1, **Tasks: Run Task**) lists the nine **native:**, **vehicle:** and **tests:** tasks; Ctrl+Shift+B runs the default, **vehicle: build + restart vehicle** (next slide)
4. **Run and Debug** view (Ctrl+Shift+D, or its icon in the Activity Bar on the left): the menu at the top of the view lists the four launch configurations; F5 starts the one chosen
5. **Terminal › New Terminal** opens the integrated terminal at the bottom; **claude** starts Claude Code in the same tree, with the project skills

You should see

- The Explorer shows cFS, run, tools, web
- IntelliSense resolving cFE and app headers (include paths come with .vscode/)
- The Call Stack view listing every cFS task as a thread when the native launch stops at a breakpoint
- **Vehicle: attach to QEMU (PowerPC/RTEMS, GDB server)** connecting in a second and showing the source of whatever task is executing
- No copy or sync step anywhere: you edit the real tree the vehicle boots from

Without VS Code the same loop works from any SSH session: the task commands are plain scripts in tools/dev/ and run/, listed on the next slides. Everything runs from /home/efsi/draco7_0 (the bind mount), never from /var/www.

The VS Code tasks and launch configurations, by their exact names

Terminal › Run Task...	What it runs
native: build (Linux, debug)	make -C cFS SIMULATION=native BUILDTYPE=debug MISSIONCONFIG=draco O=build_native install
native: build one app	the same build, then tools/dev/es_app.py reload <app> (asks for the app)
native: restart app (state reset, same code)	tools/dev/es_app.py restart <app>
vehicle: build flight software + sync /cf	run/draco.sh rtems fsw && run/draco.sh rtems sync_cf; no reboot
vehicle: build + restart vehicle (Ctrl+Shift+B)	tools/dev/vehicle-restart.sh
vehicle: restart only (no build)	tools/dev/vehicle-restart.sh --no-build
vehicle: console log (follow)	tail -n 50 -f files/console.log
vehicle: gdb in a terminal	tools/dev/gdb-vehicle.sh
tests: app unit tests with coverage (acs, nav, imu)	both test layers for the three mission apps

Run and Debug (F5)	What it does
Native: launch core-cpu1 (all apps as threads)	runs **native: build (Linux, debug)** first, then starts the mission under /usr/bin/gdb
Native: attach to running core-cpu1	attaches gdb to a native core-cpu1 already running
Vehicle: attach to QEMU (PowerPC/RTEMS, GDB server)	tools/dev/ppc-gdb (the toolchain's cross gdb in a container) to 127.0.0.1:1234
Vehicle: attach to QEMU (gdb-multiarch fallback)	/usr/bin/gdb-multiarch to the same server

Source: .vscode/tasks.json and .vscode/launch.json; tools/dev/README.md explains them. Native tasks that ask for an app offer ACS, NAV, IMU, LC, SC, SCH, HS, HK, CI_LAB, TO_LAB.

WORK INSTRUCTION · 2 WRITE THE CHANGE

Start from the requirement, keep the pattern, tag what you add

Do

1. Find the requirement in **cFS/evidence/rtems_reqs/requirements.csv** (ids dRT-<APP>-nnn) and in **verify/requirements.yaml**. None fits? Write one there first; do not invent a tag in code
2. Read the app's source in **cFS/apps/<app>/fsw/src** and its tests before editing (VS Code: Ctrl+P and the file name opens it; Ctrl+G goes to a line)
3. Put logic that needs no cFE call in the pure module; give it its numbers through a struct
4. Any new number that could change goes in **<app>_tbl.h** and **fsw/tables/<app>_def_cfg.c**, with its rule in the validation callback and in the comment above the struct
5. Tag each implementing line `[dRT-APP-nnn]` as the surrounding code does
6. A new command or telemetry field: change <app>_msg.h, then the COSMOS definition, then the CCSDS length from the compiled struct (slide 19)

Checks before you move on

- The pure module still compiles with plain gcc and no cFE header
- No new #define that is really a tunable
- No line in a COSMOS .txt or plugin.txt ends in + or \ (the parser joins it to the next line)
- Event ids unique within the app; event text under 122 characters
- File names the vehicle will open are 19 characters or fewer; anything an app creates goes to /ram, never /cf

The table rule and the pattern are in CLAUDE.md (Constants live in tables) and in claude-notes/TABLE-CONSTANTS-AUDIT.md, which lists every constant, its file and line, and the table it moved to.

WORK INSTRUCTION · 3 UNIT TESTS

Two layers on the host, seconds each; both must print RESULT: PASS and hold the floors

Do

1. **Terminal › New Terminal**, `cd /home/efsi/draco7_0/cFS`, then `bash apps/nav/unit-test/run_coverage.sh` (pure module, plain gcc with gcov)
2. Then `bash apps/nav/unit-test/run_app_coverage.sh` (app shell, cFS ut_assert with cFE stubs, native build in cFS/build_apptest_nav)
3. All three apps at once: **Terminal › Run Task... › tests: app unit tests with coverage (acs, nav, imu)**
4. Add or change a test for every behaviour you touched: a case that asserts the requirement, not one that merely executes the line
5. Read the floors in **verify/config.yaml** under coverage.modules; never lower one

You should see

- **RESULT: pass=n fail=0** and **COVERAGE: 100.00%** from the pure layer; **RESULT: PASS** with lines and branches from the app layer
- Logs in unit-test/_cov and _cov_app: build.log, run.log, gcov.log and the annotated .gcov
- An uncovered line marked ##### in the .gcov is a missing test, or dead code to remove

```
$ bash apps/nav/unit-test/run_coverage.sh
RESULT: pass=49 fail=0
nav_gate.c  Lines executed:100.00% of 57   (uncovered executable lines: 0)
COVERAGE: 100.00% of 57
COVERAGE-FULL: yes
real    0m9.501s

$ bash apps/nav/unit-test/run_app_coverage.sh
179: COMPLETE: 56 tests Segment(s) executed
1/1 Test #179: coverage-nav-nav_app ....   Passed    0.07 sec
100% tests passed, 0 tests failed out of 1
nav_app.c  lines 100.00% of 248  branches 100.00% of 94
RESULT: PASS
COVERAGE: 100.00%
BRANCHES: 100.00%
real    0m29.506s
```

NAV, measured 28 October 2026. ACS: 66 pure cases; IMU: 217. Floors: lines 100 % for every module; branches 100 % except nav_gate 90 % and acs_app 98 %, each open arm documented in config.yaml

The closed evidence run records these as V-ACS/NAV/IMU-APPCOV and the gate fails on any regression below a floor. The native app-test build needs the tracked symlink cFS/draco_defs/tables and fs.mqueue.msg_max raised (already set on this host).

WORK INSTRUCTION · 4 STATIC ANALYSIS

cppcheck on what you touched; every finding is fixed or written down

Do

1. From the repository root: `cppcheck --enable=warning,style,performance --inline-suppr cFS/apps/<app>/fsw/src`
2. A **warning** or **error** finding: fix the code
3. A **style** or **performance** finding you decide not to act on: add a row to **cFS/evidence/static/mission_apps_deviation_log.csv** with rule, file and line, severity, verdict DEVIATION and the rationale (why it has no functional or safety effect)
4. Never add a `cppcheck-suppress` comment: the log is the only place findings are dispositioned

```
$ cppcheck --enable=warning,style,performance --inline-
suppr cFS/apps/nav/fsw/src
Checking cFS/apps/nav/fsw/src/nav_app.c ...
1/2 files checked 82% done
Checking cFS/apps/nav/fsw/src/nav_gate.c ...
2/2 files checked 100% done
real    0m0.586s
```

The record so far

- Cppcheck 2.13 over acs, nav, imu, the mission tables, the CI_LAB override and the three test harnesses: **0 defects**, 6 style findings, 0 code changes
- The deviation log holds those 6 rows (variable scope, const pointer), each with its rationale
- Raw output kept beside it: mission_apps_cppcheck_raw.txt and the complexity report

A finding is only closed by a code change or a logged deviation; silence is not a disposition. The same applies to the gate's deviations: verify/config.yaml deviations: carries who, when and why, and the gate warns instead of failing.

WORK INSTRUCTION · 5 BUILD AND RESTART

One script: cross build, sync, reboot, wait, re-map the ground, 42 back

Do

1. Tell anyone on COSMOS or Yamcs first: the vehicle is gone for about 90 s of boot and comes back fresh (unflown)
2. VS Code: **Ctrl+Shift+B**, the default build task **vehicle: build + restart vehicle**; it runs in a terminal panel at the bottom. Shell: `tools/dev/vehicle-restart.sh`. **--no-build** (task **vehicle: restart only (no build)**) reboots only
3. By hand it is: `run/draco.sh rtems fsw`, then `run/draco.sh rtems sync_cf`, then stop the 42 view, recreate the qemu container, wait for OPERATIONAL, `run/bootstrap-draco.sh --per-boot`, start the 42 view
4. Watch the console: https://draco.efsi.com/ › Demo control › job pane › **Serial console** tab, or the task **vehicle: console log (follow)**, or `tail -f files/console.log`

You should see

- Section headers in the terminal: == build flight software (PowerPC/RTEMS) == · == sync dev /cf == · == stop the 42 view (it holds the bridge) == · == recreate the vehicle container (picks up compose changes) == · == waiting for cFE OPERATIONAL … `ok (n s)` · == per-boot ground setup (COSMOS targets, downlink) == · == 42 view back up ==
- The build ends by packaging **files/sp0**; sync_cf copies the apps' table files into root/cf_rtems
- Console: U-Boot TFTP, RTEMS, then **CFE_ES_Main entering OPERATIONAL state** within about 60 s, then each app's init event
- per-boot maps ACS, NAV, IMU to CFS_INT and enables the downlink; the front door's status strip badge reads **FRESH · launch-ready**
- Last line: **vehicle restarted. GDB server: 127.0.0.1:1234**

About 3 minutes (28 October); the training walkthrough measured about 4 on 3 November. Why each step exists, and what to do when it stops: next slide.

Why each step exists, and what stops the script

Why each step exists

- **sync_cf after every build**: the running vehicle reads /cf over NFS from root/cf_rtems; skip it and the new tables never arrive
- **The 42 view is stopped before the boot**: frames from the bridge reaching the emulated NIC before U-Boot has a receive ring wedge the boot (ARP retry count exceeded); only a new QEMU process clears it
- **The console log is deleted first**, so the OPERATIONAL marker is this boot's, not last week's
- **Never build QEMU with draco.sh qemu** on this host; use run/build-qemu.sh (22.04 container, the glibc of the vehicle container)
- **files/sp0 is a gzip uImage**: check symbols with powerpc-rtems6-nm on cFS/build_rtems/exe/cpu1/core-cpu1.exe, not strings
- Never copy a build tree between hosts; rebuild

When it stops

- **BUILD FAILED** or **SYNC FAILED**: it stops before touching the vehicle. Read the first compiler error, not the last; the cross build runs in a container where the repo is /cfs
- **docker: not found** at once: the terminal lacks /snap/bin on its PATH; `export PATH=/snap/bin:$PATH` and press Ctrl+Shift+B again
- **TIMEOUT** waiting for OPERATIONAL (240 s): read the Serial console; front door › Demo control › HEALTH › **Verify**, then STACK › **Fresh vehicle**
- A table layout change also needs the ground catalog rebuilt: `python3 tools/tables/catalog.py --build --roundtrip`

WORK INSTRUCTION · 6 VERIFY ON THE VEHICLE

A pass needs positive evidence that the path ran, never the absence of an error

Do, in this order

1. **Console**: the app's init event with the new version or table name; for a table change the activation event (e.g. *NAV: CFG_TBL activated, version 2*)
2. **Telemetry**: the item through the COSMOS API (get_tlm) with a fresh RECEIVED time, or on the app's page (Viewer › GNC › **ACS**, **IMU**, **NAV (GPS gate)**); counters that should move, moving
3. **Health**: `run/bootstrap-draco.sh --verify` (or front door › Demo control › HEALTH › **Verify**), read-only, checks the stack, the stations and telemetry
4. **Behaviour**: the mission test on a fresh vehicle (front door › Demo control › MISSION › **Mission test (COSMOS)**, or Viewer › Tools › **Script Runner**, FOGI/procedures/fogi_mission_test.rb): 9 phases, about 3 minutes, asserts ACS, LC, IMU and NAV
5. **Events**: the **Events** dock at the bottom of every viewer page, or Yamcs › **Events** (left sidebar): the app's own events, including the ones you did not expect

What counts as evidence

- An event line, a counter that moved, a packet timestamp, a dump that compares byte for byte
- Not: "no error appeared", "the build passed", "it worked natively"
- **Never disable a gate** (NAV or IMU validation, LC actionpoints, HS) to make a check green; if a gate blocks you, say so and stop
- After a 42 restart NAV and IMU reject on purpose (replayed time); use the reset recipe in CLAUDE.md, do not fix the gate
- A *received with actual packet length* message in COSMOS means a ground definition no longer matches the struct: fix the definition, never the struct to suit the ground

Keep what you saw: the console lines, the API output, the mission-test log name (COSMOS tool_logs/sr/<date>/..._sr_fogi_mission_test.txt). They go in the notes and, for a requirement, in the evidence folder.

GDB through QEMU's server: a snapshot in seconds, then resume

Do (batch, the reliable form)

1. Attach for a snapshot: the command on the right, from a terminal in /home/efsi/draco7_0. The three **-iex** settings must come before the ELF loads or GDB says *Architecture of file not recognized*. Since 3 November also add `-iex "set auto-load python-scripts off"`: with the RTEMS scripts loaded (the *RTEMS GDB Support* line) a breakpoint never stops the vehicle
2. **State**: print a global (ACS_AppData, NAV_AppData, IMU_AppData); globals are right at any time, locals only inside a frame
3. **Catch a path**: `-ex "break IMU_Tick" -ex continue -ex "info args" -ex "bt 4" -ex finish -ex delete -ex detach`, then trigger it from the ground; wrap in `timeout 120`
4. **Watch**: `watch ACS_AppData.Mode`; QEMU gives unlimited hardware watchpoints. **Conditional**: `break NAV_ProcessCmd if FcnCode == 3`
5. Interactive: `tools/dev/gdb-vehicle.sh` (task **vehicle: gdb in a terminal**) or VS Code (next slide)
6. Always **detach** or **continue** before you finish, and say so

```
$ PATH=/snap/bin:$PATH tools/dev/ppc-gdb -q -batch \
  -iex "set osabi none" -iex "set gnutarget elf32-powerpc" \
  -iex "set architecture powerpc:common" -iex "set auto-load safe-path /"
\
  cFS/build_rtems/exe/cpu1/core-cpu1.exe \
  -ex "set substitute-path /cfs /home/efsi/draco7_0/cFS" \
  -ex "target remote 127.0.0.1:1234" -ex "bt 4" \
  -ex "print NAV_AppData.State" -ex "print NAV_AppData.FixCount" \
  -ex "print NAV_AppData.ValidationOn" -ex "detach"
RTEMS GDB Support
#0  bsp_exc_vector_base () at
../../../bsps/powerpc/qoriq/start/start.S:396
#1  0x002ecb64 in _API_Mutex_Unlock (the_mutex=<optimized out>)
#2  0x02dfe4e0 in bsp_section_work_begin ()
$1 = 1 '\001'          NAV_STATE_FIX
$2 = 103584             fixes accepted this boot
$3 = true               validation gate on
[Inferior 1 (process 1) detached]
```

Live, 28 October 2026, vehicle stopped for under a second. The stack is the idle path (the interrupt vector base and an unlock): the CPU was idle when it stopped, which is normal for a 1 Hz app set.

A stop halts the whole vehicle; on continue the timers catch up and HS, LC or the cFE tone may react (slide 15). Report: target, exact expression, what GDB printed, your conclusion, and that the vehicle was resumed.

A breakpoint on a line of flight code, hit from the ground, in the editor

Do (example: ACS NOOP, acs_app.c:681)

1. Tell the others: a stop freezes the shared vehicle
2. Ctrl+P, **acs_app.c**; Ctrl+G, **681** (inside **ACS_ProcessCmd**, case ACS_NOOP_CC). Click in the margin left of the line number: a red dot
3. **Run and Debug** (Ctrl+Shift+D), menu at the top: **Vehicle: attach to QEMU (PowerPC/RTEMS, GDB server)**; press F5. The debug toolbar appears at the top; the vehicle pauses a moment while it attaches, then runs on
4. Send **ACS NOOP** from the ground: Viewer › Tools › **Command Sender** (target ACS, command NOOP, **Send**)
5. Read: VS Code stops on line 681 (yellow). **Call Stack** (left): ACS_ProcessCmd ← ACS_AppMain ← CFE_ES_TaskEntryPoint. Hover **A->CmdCount**: it already counts this NOOP (line 680 ran)
6. F5 (Continue); click the red dot to remove it; Shift+F5 (Disconnect). The event appears in the viewer's **Events** dock

While it is stopped

- The **whole vehicle** is halted: telemetry pauses, COSMOS goes stale, and on continue the timers catch up; HS and LC may react, and the cFE tone can be rejected for a cycle
- Keep it under a minute or two; a long stop can trip IMU STALE (recovery: IMU CLEAR_FAULT, LC SET_AP_STATE AP_ID 2 NEW_STATE 1, ACS SET_RATE_SOURCE SOURCE AUTO, or Fresh vehicle)
- For long sessions use the native build (next slide)
- Never stops? The launch configuration needs **set auto-load python-scripts off**; a configuration of your own needs it too
- Assembly instead of source: paths in the binary start with /cfs; the configs map them with set substitute-path

The same from a terminal: tools/dev/gdb-vehicle.sh, then break acs_app.c:681, continue, send ACS NOOP, bt 3, print A->CmdCount, delete, detach. Measured 3 November (training walkthrough F, web/training/walkthroughs.js).

WORK INSTRUCTION · 8 DEBUG NATIVELY

The same mission as a Linux process: every app a thread, new code without a reboot

Do

1. Build: `make -C cFS SIMULATION=native BUILDTYPE=debug MISSIONCONFIG=draco O=build_native install` (VS Code task **native: build (Linux, debug)**)
2. Run: `cd cFS/build_native/exe/cpu1 && ./core-cpu1`; CI_LAB listens on UDP 1333 on this host
3. Debug: Run and Debug (Ctrl+Shift+D) › **Native: launch core-cpu1 (all apps as threads)** (builds, starts under gdb) or **Native: attach to running core-cpu1**, then F5; by hand `gdb -p $(pgrep -f exe/cpu1/core-cpu1$)`
4. `info threads`, `thread apply all bt`; a breakpoint in one app stops only when that task runs it
5. Changed one app? Rebuild it and `tools/dev/es_app.py reload ACS`: CFE_ES stops, unloads and reloads the .so with the new code (task **native: build one app**, which asks for the app; **native: restart app (state reset, same code)** only resets it)
6. Send commands with the same tool: `es_app.py noop` builds a CCSDS packet with the cFE checksum; **--host 10.5.5.38** aims it at the vehicle instead

Keep in mind

- Little-endian, POSIX OSAL, Linux timing: byte-order bugs, board behaviour and the boot chain do not show here
- Table files and the start-up script come from the same mission config through the tracked symlink **cFS/draco_defs/tables**
- No ground station is attached by default: point COSMOS at it only if you need screens, otherwise read events on the console
- What you prove here is a hypothesis for the vehicle: confirm there before it counts as evidence
- A long investigation (hours, many stops) belongs here, not on the vehicle, where every stop freezes the mission

The native build is also what the app unit tests build on (cFS/build_apptest_<app>), so a native build problem usually shows first as a test build failure. Message queue depth needs fs.mqueue.msg_max 10000 (set in /etc/sysctl.d/90-draco-cfs-native.conf).

WORK INSTRUCTION · 9 TRACE, GATE, RECORD

The change is finished when the trace is clean, the gate passes and the notes say how it was verified

Do

1. `python3 cFS/evidence/rtems_reqs/gen_trace.py`: rewrites TRACE.md and TRACE_MODEL.md from the tags in the code and the tests
2. `python3 verify/dvr_gate.py`: the rules over requirements, coverage floors, closed-run verdicts and deviations
3. A new requirement: add it to **verify/requirements.yaml** and **requirements.csv**; its verdict stays PENDING until a closed evidence run turns it VERIFIED-RUN
4. Closed run, when the vehicle is free (about 40 min): `docker compose -p draco7 down` first, then `python3 cFS/evidence/rtems_reqs/run_evidence.py --keep-up`, then **--per-boot**
5. Write the NOW entry in **claude-notes.md**: what changed, why, how it was verified, what is open. CLAUDE.md only if a convention changed
6. Show the diff; commit only when asked

```
$ python3 cFS/evidence/rtems_reqs/gen_trace.py
wrote cFS/evidence/rtems_reqs/TRACE.md + TRACE_MODEL.md:
  83 reqs, 83 traced, 0 untagged, 0 unknown tags

$ python3 verify/dvr_gate.py
  [WARN] R4-deviation: dRT-RMT-006: V-RMT-SERSHELL FAILED,
  dispositioned 2026-09-25 by project (...): The UART1 serial
  shell is suppressed by the tracked flag file root/no_serial_shell
GATE: PASS  (1 warnings)
```

You should see

- **0 untagged, 0 unknown tags**: every requirement has code or a test behind it, and no tag in the code points nowhere
- **GATE: PASS**; a warning is an accepted deviation with who, when and why in verify/config.yaml, never a silenced check
- After a closed run: cFS/evidence/runs/run_<stamp>/ with run.json, and the verdict column updated by apply_verdicts.py

The three table requirements written on 27 October (dRT-ACS-025, dRT-NAV-010, dRT-IMU-011) are traced and tested but still PENDING: the next closed run is the open item that closes them.

WORKED EXAMPLE · DEBUGGING

Three bugs inside cFE Table Services, found on the running vehicle with GDB, fixed to match upstream

What was seen

- Every ground **LOAD_TBL** answered *No working buffers available* while housekeeping showed all four shared buffers free
- After a sequence load, SC reported *Table manage command packet error: table ID = 1218789452*
- **DUMP_TBL** of any table never loaded from the ground was rejected with no event; after four, *Too many table dumps* until reboot

How the debugger settled each one

- Breakpoint in **CFE_TBL_LoadCmd**, step: it never called CFE_TBL_ValidateLoadRequest, the step that reserves the load buffer; and never closed the file (one descriptor leaked per command)
- Breakpoint in **CFE_TBL_SendNotificationMsg**, print the message: a 7-byte packet, because the task-init CFE_MSG_Init with an invalid id fails before the size is set; SC read its parameter from stale bus bytes
- Print the registry entry after an app's start-up load: **ActiveBufferId** pointed at a buffer whose LoadBufferId was UNDEFINED; **CFE_TBL_DiscardWorkingBuffer** had freed the table's own buffer right after it became active

The fixes

- cfe_tbl_task_cmds.c: validate the load request, close the file (as nasa/cFE main)
- cfe_tbl_internal.c: initialise the notification with the real id and size on every send
- cfe_tbl_internal.c: free only shared buffers (as main); cfe_tbl_dump.c: free the dump block on failure and raise an event
- cfe_tbl_api.c: the validation-failed event prints the app callback's result, not the API status

Proof afterwards (27 October)

- Six tables never ground-loaded dumped with 0 Table Services errors
- RTS 8 built on the ground, loaded, validated by SC, started; three apps answered 2 s apart
- NAV table v2 activated with its event, v3 refused by name, v1 restored; dumps byte-identical
- Mission test PASS, 189 s, three times that day

Missed by reading, tests and the native build: start-up loads were correct; only the ground command path broke, only on a vehicle (cFE unit tests are not built here). Details: CLAUDE.md, claude-notes/RTS-TOOL.md.

WORKED EXAMPLE · WRITING TO THE CONTRACT

The ground dictionary is audited against the compiled binary: sizes and offsets come from DWARF, not from reading headers

The problem

- Apps that check command length exactly rejected ground commands with an event (*CF: invalid ground command length for command 0x06, expected 20 got 14*); apps that check nothing (TO_LAB) silently misread the fields
- The headers looked right. The compiler's view was not what a reader assumed: tail padding, alignment of doubles, two header styles in the plugin

The method

1. Ask the binary: `gdb-multiarch -batch -ex 'set architecture powerpc:common' -ex 'file core-cpu1.exe' -ex 'ptype /o NAV_GpsFixCmd_t'`
2. **CCSDS_LEN = sizeof(struct) - 7**, and sizeof includes the tail padding (CF_TransactionCmd_t: 17 bytes of fields, 20 bytes long)
3. Field offsets and enumeration values from the same output; compare every COSMOS definition; fix the definition, never the struct to suit the ground
4. Retest live: send each command, read the app's events

```
(gdb) ptype /o NAV_GpsFixCmd_t
type = struct {
/*    0  |   8 */    CFE_MSG_CommandHeader_t CmdHeader;
/*    8  |   1 */    uint8 Valid;
/*    9  |   3 */    uint8 Spare[3];
/*   12  |   4 */    uint32 Rollover;
/*   16  |   4 */    uint32 Week;
/*   20  |   4 */    uint32 Spare2;
/*   24  |   8 */    double Sec;
/*   32  |  24 */    double PosN[3];
/*   56  |  24 */    double VelN[3];
/*   80  |  24 */    double PosW[3];
/*  104  |  24 */    double VelW[3];
/*  128  |   8 */    double Lat;
/*  136  |   8 */    double Lon;
/*  144  |   8 */    double Alt;
                      /* total size (bytes):  152 */   so CCSDS_LEN 145
```

The result (22 October)

- **257 of 257** commands match flight on message id, function code and length; 28 definitions fixed, one command that does not exist in this cFE removed
- Record: claude-notes/CMD-AUDIT-2026-10-22.md; the Yamcs dictionary is generated from the COSMOS one, so it inherited the fixes

The same DWARF is what the Table Workbench reads for table layouts (tools/tables/dwarf.py), so a struct change in flight reaches the ground tools by rebuilding the catalog, not by hand-editing offsets. Big-endian on the wire: the vehicle's byte order, whatever the host.

TROUBLESHOOTING

What you see, what it means, what to do

You see	It means	Do
"mkdir /var/www: read-only file system" from Docker	A stack command ran from /var/www; snap Docker only sees $HOME	Run it from /home/efsi/draco7_0 (findmnt /home/efsi/draco7_0 to check the bind mount)
No OPERATIONAL marker after the restart; console shows "ARP Retry count exceeded"	Frames reached the emulated NIC before U-Boot had a receive ring (the 42 bridge was sending)	Stop the 42 view, recreate the qemu container (vehicle-restart.sh --no-build does both)
New table values never arrive; app still reports the old version	sync_cf was skipped; the vehicle reads /cf from root/cf_rtems over NFS	run/draco.sh rtems sync_cf, then restart the app or reboot
GDB: "Architecture of file not recognized"	The ELF was loaded before osabi, gnutarget and architecture were set	Use the -iex form (slide 14) or the VS Code configuration; both set them first
Breakpoint set, never hit; the app seems idle	The path did not run (wrong function, app not scheduled, command refused)	Print the app's counters; trigger from the ground with es_app.py --host 10.5.5.38; break on ProcessCmd first
After continuing: IMU STALE, LC events, "tone rejected"	The vehicle was stopped long enough for timers to lapse; the catch-up fires the gates	Expected. Keep stops short; use the native build for long sessions; if a gate latched: IMU CLEAR_FAULT, LC SET_AP_STATE AP_ID 2 NEW_STATE 1, ACS SET_RATE_SOURCE SOURCE AUTO (slide 15)
App event "invalid ground command length, expected n got m"	The COSMOS definition's CCSDS_LEN is not sizeof(struct) - 7	ptype /o the struct (slide 19); fix the definition; reinstall the plugin
Unit test build fails in build_apptest_<app>	The native mission build broke: missing symlink draco_defs/tables, or a new source not in CMakeLists	Check the symlink; add the file to the app's CMakeLists and unit-test/CMakeLists
open() returns -104 or FM hangs	File name over 19 characters, or a create on /cf (RTEMS NFS cannot create)	Shorten the name; write to /ram; never FM CREATE_DIR or COPY to /cf

Not here? The console log (front door › Demo control › Serial console), the app's events (viewer Events dock, Yamcs › Events) and a GDB snapshot settle most questions; /fsw-debug does exactly that.

THE RULES, AND CLAUDE CODE IN THE LOOP

Rules that came from this project's own mistakes, and how the assistant works under them

Rules

- **A pass needs positive evidence** the path executed: an event, a counter, a timestamp; never the absence of an error
- **Never disable a gate** to make a demo or a check green; if it blocks you, say so and stop
- **Workarounds and feature drops are written down** as a visible decision in the notes
- **Constants live in tables**; a new tunable as a #define is rejected in review
- **sync_cf after every build; everything from /home/efsi/draco7_0**
- **Ask the binary** for layouts; fix definitions, not structs
- **Never lower a coverage floor; never silence a gate check** except through a logged deviation
- **A person reviews every diff before it is committed**; commit only when asked

Claude Code in the loop

- Runs on this server in the VS Code terminal or any SSH session, in the same tree, under the same gates
- **/fsw-change <what>**: steps 1 to 8 of the loop; shows the diff and the verification output, commits only when asked
- **/fsw-debug <question>**: the batch GDB recipes on the vehicle or the native build; resumes the vehicle; separates what it saw from what it infers
- **/cosmos-screen <page>**: screens and definitions from the flight structs, reinstall, live check
- Under NASA handbook topic 7.25 and SWE-146 the developer remains responsible for AI-written code: same repository, same static analysis, unit tests, closed runs and review (CLASSIFICATION.md section 5). It does not set requirements, approve a bypass or sign anything
- The instructions given to it are configuration items (cFS/evidence/logs/campaign-prompts.md); the notes record what it did per session

Oversight mode (handbook 8.25): a person in the loop directing each task, the tool's permissions and the flight-software gates enforcing the boundary. Reference pages: tools/dev/README.md, .claude/skills/, CLAUDE.md Notes, claude-notes/ONBOARDING.md.