Point of Departure: UART UI + Repo Architecture (TM4C1294 PWM Controller)

Date: 2025-12-27
Repo folder: `integr_v03/`
Target: TI TM4C1294XL (Cortex-M4F)

This note is a consolidated snapshot of the current firmware architecture, how the UART "UI" works today, the repo's existing workflows/tools, and the most important technical constraints to keep in mind before the next merge/improvement cycle.

---

1) TL;DR snapshot

- Firmware provides **PWM output on PF2** (target ~21.5 kHz) with duty control.
- Uses **two UARTs**:
  - **UART0 (ICDI @ 9600)**: diagnostics/debug prints.
  - **UART3 (USER @ 115200)**: user command input ("UART UI").
- Uses **DTR session detection on PQ1** to gate whether a session is "active".
- There are **two CLI implementations** in the codebase:
  - A minimal ISR-driven one currently implemented directly in `main.c`.
  - A separate, richer CLI module in `cmdline.c`/`cmdline.h` (not currently wired into `main.c`).
- Memory is **custom** (Newlib syscall stubs + custom allocator). Important constraint: `malloc_simple.c` implements `free()` as a **no-op** (no memory reclamation).

---

2) Project structure: what each piece is for

Main application

- `main.c`
  - Clock config (`SysCtlClockFreqSet` → 120 MHz)
  - PWM init/update (PF2)
  - UART init (UART0 + UART3)
  - DTR gating (PQ1)
  - UART3 RX ISR (echo + line accumulation)
  - Command parsing/dispatch (currently: `PSYN n`)

UART UI / command-line

- `cmdline.h` / `cmdline.c`
  - Provides a more full-featured line editor UX (prompt, ANSI colors, backspace, Ctrl-U, etc.)
  - Assumes it owns the UART3 "read loop" (polled loop style)
  - Mentions that `set_pwm_percent()` must be externally visible (not `static`) if used from this module

Diagnostics

- `diag_uart.h` / `diag_uart.c`

  - UART0 diagnostic output helpers: `diag_puts`, memory layout prints, variable dumps
  - Contains global replacements for `printf/sprintf/snprintf` (high impact: affects any file that links those symbols)
  - Includes a tiny formatter that supports only a limited subset of format specifiers (important when using `%ld`, etc.)

Memory / syscalls

- `syscalls.c`
  - Implements `_sbrk` for heap growth and minimal newlib syscall stubs
- `malloc_simple.c`
  - Overrides `malloc/free/realloc`
  - **Important**: `free()` is a no-op
- `malloc_lock_stubs.c`
  - Provides `__malloc_lock/__malloc_unlock` stubs for newlib thread-safety builds

Drivers / networking (not currently "UI path", but available)

- `drivers/` includes TI-provided ethernet/LwIP pieces (e.g. `eth_client_lwip.c`, `http.c`)

---

3) Build, flash, and UART workflows (existing repo tools)

Build + flash

From repo root (`integr_v03/`):

```
make clean
make
make flash
```

Notes: - `Makefile` builds all `*.c` in the folder and links against TivaWare driverlib. - `STELLARISWARE_PATH` must point at a local TivaWare checkout.

UART capture (two ports)

The repo includes `tools/uart_session.py` plus Makefile targets that wrap it.

Capture both UARTs (writes logs into `./logs/`):

```
make capture
```

Send a command to UART3:

```
make send-uart3 CMD='PSYN 44\r'
```

Autonomous flow (flash + capture + optional command):

```
make auto
make auto CMD='PSYN 44\r' DURATION=8
```

Override devices/bauds if needed:

```
make capture UART0_DEV=/dev/ttyACM0 UART3_DEV=/dev/ttyUSB1
```

Post-build symbol sanity checks

The repo includes a helper script:

```
bash postMake_checks.sh
```

It checks that heap/stack symbols exist and writes: `-undefined_malloc_symbols.txt` `-defined_malloc_symbols.txt`

---

4) Current UART UI behavior (what's actually active today)

Session gating

- Session "inactive" while PQ1 reads high.
- When DTR asserts (PQ1 low), session is active and UART3 UI prompt appears.

UART3 RX path

- UART3 interrupt handler echoes each received byte.
- It accumulates bytes into `user_rx_buf` until CR/LF.
- On CR/LF with non-empty buffer, it sets `user_cmd_ready=true`.

Command parsing

- The active parser is in `main.c` (`process_user_line()`):
  - trims leading whitespace
  - tokenizes by spaces/tabs
  - case-insensitive command keyword
  - supports: `PSYN n` where `n` is 5..96

Notable UX limitations

- No reliable line editing (backspace/delete handling is not implemented in the active ISR path).
- Minimal discoverability (no `HELP`/`STATUS`).
- Acknowledgements are inconsistent (the "OK: duty set..." was intentionally commented out due to formatting/newlib issues).

---

5) Important architecture/technical constraints (watch-outs)

A) There are two CLI implementations

- `main.c` contains a minimal ISR-driven CLI.
- `cmdline.c` contains a richer line editor and command handler.

This is a merge hazard: - If both are enabled/used, you can get duplicate concepts (prompt/echo/dispatch) and potentially conflicting UART usage.

B) `malloc_simple.c`: `free()` is a no-op

This is a major behavioral constraint: - Anything that repeatedly allocates memory will eventually exhaust the heap. - Even code that "allocates then frees" will still consume heap because `free()` doesn't reclaim.

Consequence: - Avoid adding heap allocations in the UART UI path (command parsing/printing) unless allocator strategy changes.

C) `diag_uart.c` overrides `printf/snprintf/sprintf`

- High impact: linking these replacements means *any* code using those symbols will use the custom formatter.
- The formatter supports only a subset of format specifiers.

Rule of thumb: - Prefer TI `usprintf/usnprintf` (from `utils/ustdlib.h`) or fixed-format printing if you need richer formatting without libc surprises.

D) ISR vs blocking output

- UART output inside ISRs can be risky (latency, potential FIFO stalls).
- Current ISR uses non-blocking puts for echo, which is generally safer.

---

6) Recommended "next immediate merge" approach (minimal-risk)

Goal: improve UART UI usability without destabilizing PWM timing or heap behavior.

Step 1 — choose one UI path

Pick either: - **Option A (recommended):** Keep ISR RX accumulation in `main.c`, but refactor parsing/dispatch into a small command table and add minimal line editing (backspace, line-kill) without heap usage. - **Option B:** Switch to `cmdline.c`'s polled loop and remove the UART3 ISR accumulation. (Simpler conceptually, but ensure it doesn't fight your low-power `SysCtlSleep()` approach.)

Step 2 — add small, high-value commands

Without changing the UI beyond a basic CLI: - `HELP` — list commands + examples - `STATUS` — print current duty + period/pulse, maybe session pin state - `PSYN ?` or `GET PSYN` — read back current value

Step 3 — keep formatting deterministic

- Avoid heap formatting helpers (`diag_vasprintf_heap`, etc.) until allocator is improved.
- Avoid `%ld`/advanced formats in the custom formatter unless extended.

---

7) Generating a print-friendly PDF of this note

This repo includes a helper:

`tools/md2pdf.sh docs/Point-of-Departure_UART-UI_2025-12-27.md docs/Point-of-Departure_UART-UI_2025-12-27.pdf`

Requirements: - `pandoc` - LaTeX engine: `lualatex` (fallback: `xelatex`, then `pdflatex`)

Tip: - The script disables syntax highlighting by default to avoid LaTeX failures with embedded `\r` and `\n` sequences.

---

8) Appendix: quick command examples

- Set duty to 50%:
  - UART3: `PSYN 50` + Enter
  - or: `make send-uart3 CMD='PSYN 50\r'`
- Capture UART logs while testing:
  - `make capture`
- Flash + capture + send command:
  - `make auto CMD='PSYN 44\r' DURATION=8`