

## Firmware Function Reference (TM4C1294 – integr\_v03)

This document is a function-by-function reference for the current firmware in this workspace. It focuses on the UART console/diagnostics path and the modules that were actively changed during the recent UART UX + ESP32-feature recovery work.

### UART Roles (High-Level)

- **UART3 (USER, 115200)**: interactive console. RX is ISR-driven (`USERUARTIntHandler`) with echo + in-ISR line editing.
- **UART0 (ICDI, 9600)**: diagnostics/status output. Runtime diagnostics are gated by `DEBUG ON/OFF`.

### Session Boundary (DTR on PQ1)

- DTR is read from **PQ1 (DTR\_PORT/DTR\_PIN)** and is **polled in the main loop**.
- Because DTR is polled (not interrupt-driven), the session loop uses periodic `SysCtlDelay(...)` to ensure disconnects are detected promptly (no “press ENTER to notice disconnect” behavior).

---

## main.c

### Global state

- `g_ui32SysClock`: system clock (Hz), set by `setup_system_clock()`.
- PWM state:
  - `g_pwmPeriod`: PWM period (ticks).
  - `g_pwmPulse`: PWM pulse width (ticks).
- UART3 RX/command state (ISR-owned, main loop consumes):
  - `user_rx_buf[]`: UART3 line accumulator.
  - `user_rx_len`: current bytes accumulated.
  - `user_cmd_ready`: set by ISR when a non-empty line is completed.
- GOTCHA hidden trigger state (UART3 ISR):
  - `g_uart3_p_run`: count of consecutive P keystrokes.
  - `g_uart3_gotcha_pending`: set when 5 consecutive P are typed.
- UART0 diagnostics gating:
  - `g_debug_enabled`: default false.

### `void debug_set_enabled(bool enabled)`

Enables/disables UART0 diagnostic output at runtime.

- Called from the `DEBUG ON/OFF` command via [commands.c](#).

### `bool debug_is_enabled(void)`

Returns the current diagnostic gating state.

**void pwm\_set\_percent(uint32\_t percent)**

Public wrapper used by the command layer.

- Delegates to set\_pwm\_percent(percent).

**void pwm\_set\_enabled(bool enabled)**

Enables/disables PWM output on PF2.

- enabled=true: restores PF2 mux to M0PWM2 and enables PWM output.
- enabled=false: disables PWM output and reconfigures PF2 as GPIO output low.

This exists primarily to support PSYN OFF for scope/debug, so the tach input can be observed without PWM coupling.

**bool pwm\_is\_enabled(void)**

Returns the current PWM output enabled state.

**static void set\_pwm\_percent(uint32\_t percent)**

Sets PWM duty cycle without disabling/re-enabling the generator.

- Bounds/clamps: percent is clamped to 0..100.
- Ensures pulse width remains in 1..(period-1).
- Calls:
  - PWMPulseWidthSet(PWM0\_BASE, PWM\_OUT\_2, pulse)

**static void setup\_system\_clock(void)**

Configures system clock to 120 MHz via PLL.

- Sets g\_ui32SysClock.

**static void setup\_pwm\_pf2(void)**

Configures PWM output on PF2 / M0PWM2.

- Enables PWM0 + GPIOF.
- Uses PWM\_SYSCLK\_DIV\_1.
- Computes and stores g\_pwmPeriod.

**static void setup\_uarts(void)**

Configures UART0 and UART3, plus supporting GPIO.

- UART0: PA0/PA1, 9600 8N1.
- UART3: PJ0/PJ1, 115200 8N1.
- PF4 configured as GPIO output (used as RX activity LED and GOTCHA blink).
- PQ1 configured as input with WPU (DTR detect).

- Enables interrupts:
  - INT\_UART0 → ICDIUARTIntHandler()
  - INT\_UART3 → USERUARTIntHandler()

### **void ICDIUARTIntHandler(void)**

UART0 ISR.

- Echoes RX bytes back to UART0.
- Briefly pulses PN0 for visibility.

### **void USERUARTIntHandler(void)**

UART3 ISR: echo + line accumulation + basic line editing.

Behavior summary:

- **Backspace/Delete** (\b or 0x7F):
  - Deletes one buffered character if user\_rx\_len > 0.
  - Emits "\b \b" erase sequence.
  - If buffer empty, emits bell (\a) to prevent erasing past the prompt boundary.
- **ENTER** (\r or \n):
  - If buffer non-empty: emits \r\n, NUL-terminates buffer, sets user\_cmd\_ready=true.
  - If buffer empty: does nothing (no extra newline/prompt spam).
- **Uppercase-as-you-type**: converts a..z to A..Z before echo and buffering.
- **Hidden GOTCHA**:
  - Counts consecutive P keystrokes.
  - On 5 consecutive P, sets g\_uart3\_gotcha\_pending=true and resets the counter.
  - Not a command; does not require ENTER; not listed in HELP.
- **Overflow**:
  - Resets buffer and prints ERROR: line too long + prompt.

### **static void user\_uart3\_consume\_pending\_input(void)**

Consumes pending UART3 RX bytes while UART3 interrupts are disabled.

Why it exists:

- When hosts send CRLF, the ISR may complete the line on \r and then later receive/echo the trailing \n.
- If the main loop prints the next prompt while UART3 interrupts are disabled, the delayed \n can arrive after the prompt and move the cursor, creating the “extra ENTER required” UX symptom.

What it does:

- While UART3 has bytes available:
  - Swallows extra \r/\n tails.
  - Echoes other bytes and appends them to the current buffer (if a command isn’t already pending).

### **static void flash\_pf4\_gotcha(uint32\_t flashes)**

Flashes PF4 LED flashes times.

- Used when GOTCHA triggers.
- Delay is derived from g\_ui32SysClock (currently ~75ms on/off).

```
void example_dynamic_cmd_copy_and_process(const volatile char *user_rx_buf, uint32_t len)
```

UART0-only diagnostics helper.

- Runs only when debug\_is\_enabled().
- Uses diag\_uart helpers to dump internal state and stress some malloc/formatting paths.

```
int main(void)
```

Main firmware entry.

Execution overview:

1. Clock + PWM + UART setup.
  2. Outer loop waits for DTR session.
  3. On session begin:
    - UART0 prints “SESSION WAS INITIATED”.
    - UART3 prints rainbow banner + welcome + prompt via ui\_uart3\_session\_begin().
  4. Session loop:
    - Polls DTR.
    - If g\_uart3\_gotcha\_pending is set:
      - Prints UART0 message immediately.
      - Flashes PF4.
    - If user\_cmd\_ready:
      - Disables UART3 IRQ.
      - Copies buffered line to local storage.
      - Clears ISR-owned state.
      - Calls user\_uart3\_consume\_pending\_input().
      - Re-enables UART3 IRQ.
      - Dispatches the command via commands\_process\_line(cmd\_local).
      - If DEBUG enabled: prints additional UART0 diagnostics.
    - Uses a short SysCtlDelay(...) to ensure DTR polling stays responsive.
  5. On disconnect:
    - UART0 prints “SESSION WAS DISCONNECTED” immediately (no user keystrokes required).
- 

## commands.c / commands.h

commands\_process\_line() implements the UART3 command dispatcher.

Design notes:

- Avoids sprintf/newlib printf-family in the command response path.
- Uses simple parsing (strtok\_r) and a small decimal formatter.

```
void commands_process_line(const char *line)
```

Parses and executes one complete command line.

- Trims leading whitespace.
- Uppercases command token.
- Supported commands:
  - PSYN n — sets PWM duty (5..96).
  - PSYN ON — enables PWM on PF2.
  - PSYN OFF — disables PWM and forces PF2 low.
  - TACHIN ON — start printing tach/RPM lines on UART0 every 0.5s.
  - TACHIN OFF — stop printing tach/RPM lines on UART0.
  - HELP — prints help.
  - DEBUG ON|OFF — gates UART0 diagnostics.
  - EXIT — closes the current UART3 session (no arguments).
  - TSYN ON — enable TACH synthesizer on PM3 (drives burst waveform).
  - TSYN OFF — disable TACH synthesizer (restores PM3 to tach input).

```
void pwm_set_percent(uint32_t percent) (declared in commands.h)
```

Platform-provided PWM setter (implemented in [main.c](#)).

```
void debug_set_enabled(bool enabled) / bool debug_is_enabled(void) (declared in commands.h)
```

Platform-provided debug gating API (implemented in [main.c](#)).

---

## timebase.c / timebase.h

Minimal SysTick-based timebase used by the tach sensing path.

---

```
void timebase_init(uint32_t sysClockHz)
```

Initializes SysTick to generate a 1ms interrupt and establishes the reference clock for cycle-based delta timing.

- Configures a 1ms tick using `SysTickPeriodSet(sysClockHz / 1000)`.
- Enables SysTick interrupt and SysTick counter.
- Stores:
  - `g_sysclk_hz` (for later conversion and debug)
  - `g_systick_reload` (cycles per millisecond)

Dependency note:

- The SysTick vector in [TM4C1294XL\\_startup.c](#) must point to `SysTickIntHandler()`.

**`uint32_t timebase_millis(void)`**

Returns a monotonically increasing millisecond tick counter.

- Implemented as an ISR-incremented counter (`g_ms_ticks`).
- Read uses a short global interrupt disable/enable to snapshot consistently.

**`uint32_t timebase_cycles32(void)`**

Returns a 32-bit “cycle-ish” counter derived from SysTick.

- Computes:  
– `cycles = ms * reload + (reload - SysTickValueGet())`
- Samples `g_ms_ticks` twice to avoid race at the millisecond boundary.
- Intended for **short delta measurements**; wraps naturally at 32 bits.

**`uint32_t timebase_sysclk_hz(void)`**

Returns the system clock rate passed into `timebase_init()`.

---

**tach.c / tach.h**

Interrupt-driven TACH (fan tachometer) input sensing with a simple glitch reject filter, plus optional periodic UART0 reporting.

This implementation is intentionally “debug-first”: it is good enough to diagnose coupling/noise patterns and compare strategies against the ESP32 reference (`fan_master_s2_final.ino`), but it is not yet presented as a final/production tach algorithm.

**Wiring and default pinning**

Default configuration (compile-time override via macros in [tach.h](#)):

- TACH input: **PM3 / GPIO M3**
- Electrical assumption: open-collector/open-drain tach output.
- Uses internal weak pull-up (`GPIO_PIN_TYPE_STD_WPU`, to 3.3V).

Safety note:

- Do **not** pull the tach line up to +5V directly when connected to the TM4C.

**`void tach_init(void)`**

Initializes the GPIO and interrupt configuration for tach capture.

- Configures pad: - input, 2mA drive (irrelevant for input), weak pull-up - Configures interrupt: - falling-edge trigger (GPIO\_FALLING\_EDGE)

- Clears and enables pin interrupt - enables the NVIC interrupt (IntEnable(TACH\_GPIO\_INT))

- Resets internal counters and state:

- g\_tach\_pulses,  
g\_tach\_rejects,  
g\_last\_edge\_cycles  
    ↳

---

```
###  
void  
GPIOIntHandler(void)  
GPIO  
inter-  
rupt  
han-  
dler  
that  
counts  
tach  
pulses.
```

---

-  
Inter-  
rupt  
status  
is  
read  
and  
cleared  
first. -  
For  
each  
falling  
edge  
on  
TACH\_GPIO\_PIN:  
- snap-  
shots  
a  
times-  
tamp  
via  
timebase\_cycles32()  
- com-  
putes  
delta  
= now  
-  
g\_last\_edge\_cycles  
- ap-  
plies  
**minimum-**  
**edge-**  
**spacing**  
**reject:**  
- con-  
verts  
TACH\_MIN\_EDGE\_US  
to  
cycles  
using  
timebase\_sysclk\_hz()  
- if  
delta  
<  
min\_cycles:  
incre-  
ments  
g\_tach\_rejects  
and  
ig-  
nores  
the

Glitch  
reject  
ratio-  
nale:

- The project PWM is ~21.5kHz (period ~46.5μs). A TACH\_MIN\_EDGE\_US defines fault of 200μs rejects most PWM-coupled “fake edges” on the tach line. - This is a diagnostic filter; it may need to change when we move to a period-based tach strategy like the ESP32 implementation.

---

```
###  
void  
tach_set_reporting(bool  
enabled)  
En-  
ables/dis-  
ables  
peri-  
odic  
re-  
port-  
ing to  
UART0.
```

-  
When  
en-  
abling:  
-  
sched-  
ules  
first  
report  
at now  
+  
500ms  
-  
prints  
a one-  
time  
ban-  
ner  
on  
UART0  
with  
the  
active  
GPIO  
base/pin  
and  
con-  
figu-  
ra-  
tion: -  
TACHIN  
ON:  
gpio\_base=0x...  
pin\_mask=0x...  
edge=FALL  
pullup=WPU  
-  
When  
dis-  
abling:  
- stops  
re-  
port-  
ing -  
resets  
coun-  
ters  
(pulses,  
rejects,  
last\_edge\_cycles)  
under

---

Im-  
por-  
tant  
inter-  
action  
note:  
- Re-  
port-  
ing  
writes  
to  
**UART0**  
**di-**  
**rectly**  
(ROM  
UARTCharPut)  
and is  
**not**  
gated  
by  
DEBUG  
ON/OFF.  
###  
bool  
tach\_is\_reporting(void)  
Re-  
turns  
whether  
peri-  
odic  
UART0  
re-  
port-  
ing is  
en-  
abled.  
###  
void  
tach\_task(void)

---

Periodic task (called from the main loop) that emits RPM diagnostics every 0.5s when enabled.

- Every 500ms:

- atomically snapshots and clears `g_tach_pulses` and `g_tach_rejects` - computes an implied RPM using the current simplified model:

RPM =  
60 ·  
 $pulses_{0.5s}$   
This  
comes  
from:  
- Win-  
dow  
= 0.5s  
-  
pulses/sec  
= 2 \*  
 $pulses\_in\_window$   
- For a  
2-  
pulses-  
per-  
rev  
fan:  
RPM =  
(pulses/sec)  
\* 30  
= 60  
\*  
 $pulses\_in\_window$   
-  
Prints  
one  
line  
on  
UART0:  
- TACH  
 $pulses=<n>$   
 $rejects=<n>$   
 $rpm=<n>$   
###  
Compile-  
time  
con-  
figu-  
ration  
knobs

---

These  
can  
be  
over-  
rid-  
den at  
build  
time  
(e.g. via  
-  
D...):  
-  
TACH\_GPIO\_PERIPH,  
TACH\_GPIO\_BASE,  
TACH\_GPIO\_PIN,  
TACH\_GPIO\_INT  
-  
TACH\_MIN\_EDGE\_US  
(de-  
fault  
200)  
###  
Known  
limita-  
tions  
(cur-  
rent  
diag-  
nostic  
imple-  
men-  
ta-  
tion)

-  
Counter-based  
win-dow-ing is  
sensi-tive to  
noise  
bursts;  
the  
rejects  
counter  
helps  
quan-tify  
that  
noise.

-  
Using  
a  
weak  
inter-nal  
pull-up  
may  
be too  
sus-cep-tible on  
long  
wires  
/  
noisy  
grounds;  
exter-nal  
condi-tion-ing  
may  
be re-quired.

- The

cur-

rent

RPM

con-

ver-

sion

as

---

- The  
cur-  
rent  
RPM  
con-  
ver-  
sion  
as-  
sumes  
2  
pulses/rev  
and a  
stable  
0.5s  
win-  
dow.  
Practi-  
cal  
debug  
tip:  
- Use  
PSYN  
OFF to  
force  
PF2  
low  
and  
re-  
duce  
PWM  
cou-  
pling  
while  
ob-  
serv-  
ing  
the  
tach  
signal  
and  
rejects  
be-  
hav-  
ior.

---

###  
ESP32  
refer-  
ence  
guid-  
ance  
(for  
next  
tach  
algo-  
rithm  
itera-  
tions)  
The  
ESP32  
refer-  
ence  
sketch  
(fan\_master\_s2\_final.ino)  
uses a  
differ-  
ent  
strat-  
egy  
than  
the  
cur-  
rent  
TM4C  
imple-  
men-  
ta-  
tion:

---

- ISR  
captures  
**pe-**  
**riod**  
**be-**  
**tween**  
**edges**  
using  
`micros()`  
(stores  
`lastPeriod_us`  
and a  
`newTachMeasurement`  
flag).  
- Main  
loop  
con-  
sumes  
that  
snap-  
shot  
and  
com-  
putes:  
- freq  
= `1e6`  
/  
`period_us`  
- RPM  
=  
`(freq`  
`* 60)`  
/  
`PULSES_PER_REV`

---

This  
style  
is  
often  
more  
ro-  
bust  
than  
“count  
pulses  
in a  
fixed  
win-  
dow”  
when  
noise  
bursts  
are  
present,  
be-  
cause  
you  
can  
qual-  
ify  
each  
edge-  
to-  
edge  
mea-  
sure-  
ment  
and  
dis-  
card  
out-  
liers  
with-  
out  
cor-  
rupt-  
ing  
the  
whole  
win-  
dow.

Con-  
crete  
candi-  
dates  
to  
port  
into  
the  
TM4C  
path  
(still  
diag-  
nos-  
tic/ex-  
peri-  
men-  
tal  
until  
vali-  
dated  
on the  
bench):

---

- **Hy-**  
**brid**  
**mea-**  
**sure-**  
**ment:**  
keep  
pulses/rejects  
win-  
dow  
coun-  
ters,  
but  
also  
track  
`last_period_us`  
(or  
`last_period_cycles`)  
from  
ac-  
cepted  
edges.  
- **Edge**  
**quali-**  
**fica-**  
**tion:**  
re-  
quire  
both  
a min-  
imum  
edge  
spac-  
ing  
*and* a  
plau-  
sible  
pe-  
riod  
range  
(min/max  
RPM  
bounds)  
be-  
fore  
ac-  
cept-  
ing a  
new  
pe-  
riod. -  
**Timeout-**  
**to**

Notes  
ob-  
served  
in the  
ESP32  
sketch  
worth  
mir-  
roring  
dur-  
ing  
diag-  
nos-  
tics:

---

- It uses a controlled “fake tach generator” with ~120µs low pulses to validate the algorithm path.

- It clamps commanded/target RPM to a safe min/max range (useful for sanity bounds during testing).

---

## **ui\_uart3.c / ui\_uart3.h**

UI helpers for UART3 output discipline (banner/welcome/prompt).

**void ui\_uart3\_session\_begin(void)**

Prints session-start UI:

- Deterministic ANSI “rainbow banner”.

- A short welcome line.
- A single prompt.

Implementation constraint:

- Session-begin output must be deterministic and not rely on libc-heavy string searching/formatting to avoid stalls.

**void ui\_uart3\_puts(const char \*s)**

Outputs a C string to UART3 via `UARTSend(..., UARTDEV_USER)`.

**void ui\_uart3\_prompt\_once(void)**

Prints the prompt once (`ANSI_PROMPT + PROMPT_SYMBOL + ANSI_RESET`) and avoids duplicate prompt spam.

**void ui\_uart3\_prompt\_force\_next(void)**

Clears the “prompt already printed” latch so the next `ui_uart3_prompt_once()` will print.

---

## diag\_uart.c / diag\_uart.h

Diagnostics helpers that write to UART0 (ICDI).

Important notes:

- These functions are intended for **non-ISR** contexts.
- The file contains both:
  - heap-based formatting helpers (`diag_vasprintf_heap`, `diag_snprintf_heap_send`) and
  - a minimal printf-like formatter (`diag_simple_sprintf`) plus global `sprintf/snprintf/printf` over-rides.

## UART0 output primitives

- `diag_putc(char c)`
- `diag_puts(const char *s)`
- `diag_put_hex32(uint32_t v)`
- `diag_put_u32_dec(uint32_t v)`
- `diag_put_ptr(const void *p)`

## Heap formatting helpers

- `char *diag_vasprintf_heap(const char *fmt, va_list ap)`
- `char *diag_asprintf_heap(const char *fmt, ...)`
- `int diag_snprintf_heap_send(const char *fmt, ...)`

## Memory/allocator diagnostics

- void diag\_sbrk\_probe(void)
- void diag\_test\_malloc\_with\_gpio(void)
- void diag\_test\_malloc\_sequence(void)
- void diag\_print\_memory\_layout(void)
- void diag\_print\_sbrk\_info(void)
- void diag\_print\_variable(const char \*name, const void \*addr, size\_t size, size\_t preview\_limit)
- void diag\_print\_variables\_summary(void)

## Memory protection helpers

- void diag\_check\_memory\_integrity(const char \*context)
  - void diag\_check\_stack\_usage(const char \*function\_name)
  - int diag\_stack\_bytes\_used(void)
  - int diag\_heap\_bytes\_used(void)
- 

## cmdline.c / cmdline.h (legacy)

This module provides an older UART3 command-line loop (`cmdline_run_until_disconnect`) and its own PSYN parsing.

Current status:

- The active firmware path in `main.c` uses `USERUARTIntHandler + commands_process_line()`.
- The build includes all `*.c` via the Makefile wildcard; however, link-time garbage collection (`--gc-sections`) typically discards this module unless referenced.

If you decide to use `cmdline_run_until_disconnect()` again:

- It expects a platform-visible `set_pwm_percent(uint32_t)` symbol (currently `set_pwm_percent` is static in `main.c`).
- 

## tools/uart\_session.py (host-side)

Primary host automation tool for UART0/UART3 capture and scripted testing.

Key behaviors:

- Defaults: `--send-delay 0.6, --type-delay 0.02`.
  - Preflight/postflight cleanup is enabled by default; can be disabled via `--no-preflight / --no-postflight`.
  - For testing GOTCHA (real-time keystrokes), prefer `TYPE PPPPP` rather than a line-based `SEND` that appends `ENTER`.
-

## Hidden GOTCHA Feature (current spec)

- Trigger: **5 consecutive P keystrokes typed on UART3.**
- Immediate effects:
  - UART0 prints: GOTCHA: PPPPP detected on UART3.
  - PF4 LED flashes 5 times.
- Not a command; not listed in HELP.