Embedded Robotic Car - MSP430FR2355, Wi-Fi Control, Line Following - Project Guide

C / MSP430FR2355 / 8 MHz / Code Composer Studio · ECE 306 Embedded Systems · Spring 2024

Topics: *MSP430FR2355 · background-foreground "while" OS · ISR-driven peripherals · ADC round-robin · H-bridge PWM motor control · 11-state line-following FSM · Wi-Fi IoT AT-command stack · TCP server on port 8130 · ring-buffer command parser*

> **The one-liner:** A Wi-Fi-controlled robotic car on an MSP430FR2355, with an 11-state line-following FSM, an H-bridge PWM motor driver, and a TCP command server on port 8130 that maps single-byte commands to motion and state-machine actions over a 32-byte UART ring buffer.
>
> **Why it's interesting:** My first end-to-end embedded project (Jan-May 2024, junior year). Bare-metal C across ISRs, ring buffers, FSMs, and a Wi-Fi AT-command stack, touching every layer between a 25 ms timer tick and a TCP socket. Worth noting: this was an intro-course project on a course-provided board, so the engineering is clean but small in scale.

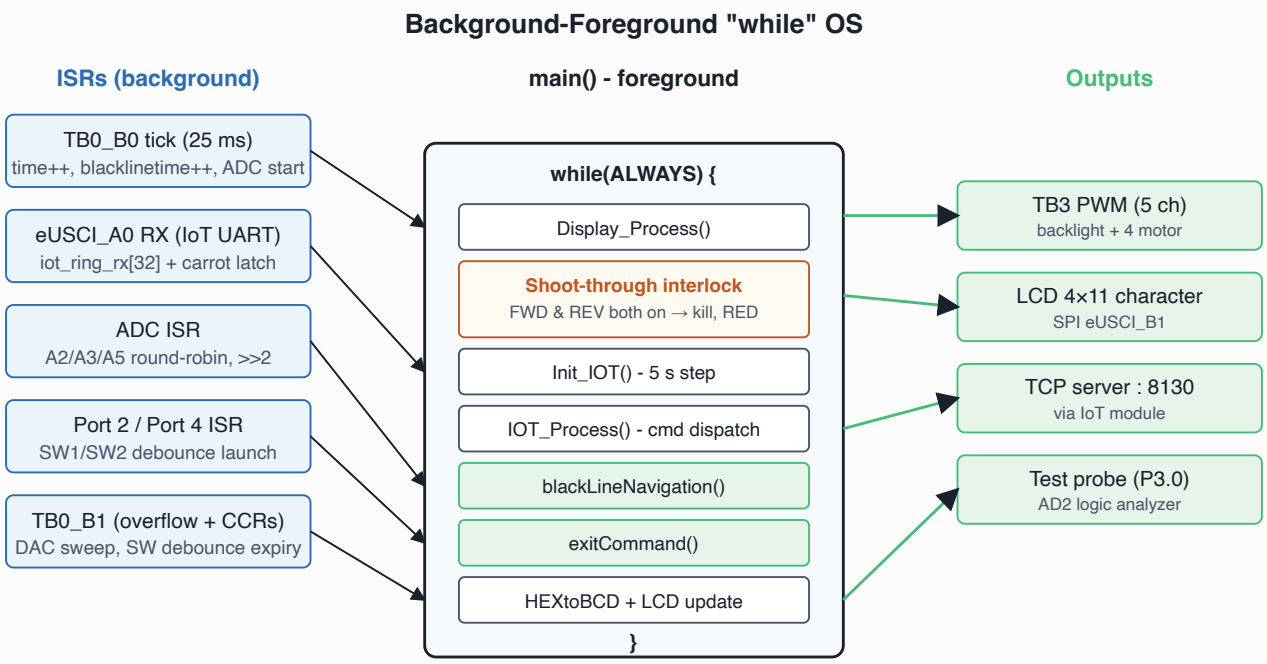
Scope - what was provided vs. what I built

ECE 306 builds the same physical platform across five revisions. The board stack and IR detector design were given by the course; everything in firmware and final integration was on me.

Layer	What it covered
Hardware (course-provided)	MSP430FR2355 FRAM board (24 MHz capable, 32 KB FRAM, run at 8 MHz); Power/Display board with thumbwheel pot, on/off switch, LCD; H-bridge FET board; IR detector board with two photodiode-based line sensors; ESP8266-class IoT module; battery pack and chassis kit.
Firmware (mine)	13 C source files + 3 headers. Port-by-port pin configuration; DCO 8 MHz + XT1-32.768 kHz software-trim clock init; ADC + DAC + REF; Timer B0 (25 ms tick + switch-debounce timers + DAC sweep); Timer B3 PWM at 50005-count period across 5 channels; eUSCI_A0/A1 UART with RX/TX ISRs; 6-step Wi-Fi AT-command init; 32-byte ring buffer + 4×16 parsed-command log; 11-state line-follow FSM + 4-state exit FSM; H-bridge shoot-through interlock; LCD hex-to-BCD display path.
Physical build (mine)	Chassis assembly; solder integration of IoT module, IR detector, FET board, LCD/power board onto the FRAM carrier; AD2-driven (Analog Discovery 2) board-level testing through revisions R10 → R22 with section-by-section bring-up (power → LCD → PWM → IR → IoT integration).

Architecture - background-foreground "while" OS

The whole system is a single `while(ALWAYS)` loop in `main.c` with five interrupt sources feeding shared globals. There's no RTOS - the "while OS" is the design pattern the course teaches and the one I built to. ISRs do the minimum work needed to capture data or set a flag; main does dispatch.



ISRs stay thin (capture/flag only). Orange = shoot-through interlock. Green = the two FSMs.

System architecture: five ISRs feed shared globals; main runs a single while loop dispatching to motion, IoT, FSMs, and LCD. The shoot-through interlock (orange) is the one hard guard; the two state machines (green) are the autonomous-mode workhorses.

Clock setup matters because it sets every other timing constant. `Init_Clocks()` runs a DCO software-trim loop (`Software_Trim`, ported from the TI BSP) against the 32.768 kHz XT1 reference to lock the DCO at 8 MHz. MCLK = SMCLK = 8 MHz, ACLK = 32.768 kHz. Every timer derives off SMCLK divided by 8 (ID__8) and 5 (TBIDEX__5), giving a 200 kHz timer clock. The 25 ms tick is `TB0CCR0_INTERVAL = 5000` counts at 200 kHz; everything else (the `ONE_SECOND = 40` tick count, the state-machine wait constants) descends from that.

Motion control - H-bridge PWM

Four brushed DC motor lines (left/right × forward/reverse) drive a FET-board H-bridge. Five PWM channels on Timer B3, in up-mode, period 50005 counts (~160 Hz at 8 MHz SMCLK): one for LCD backlight dim, four for motor direction-side pairs.

Function	Right FWD	Left FWD	Right REV	Left REV
`forwardWheelsSlow`	60%	40%	0	0
`forwardWheelsFast`	100%	90%	0	0
`reverseWheelsSlow`	0	0	40%	50%
`turnRight` (point turn)	0	40%	0	0
`turnLeft` (point turn)	40%	0	0	0
`turnClockwise` (pivot)	0	40%	40%	0
`turnCounterClockwise`	40%	0	0	40%

The left/right asymmetry on the "forward" primitives (right always ~10-20% hotter than left) is empirical compensation for physical wheel/motor mismatch on the chassis - calibrated by driving the car in a straight line and adjusting until it tracked. Not characterized, just tuned.

> **Shoot-through interlock.** The H-bridge has separate FWD and REV PWM lines per side. If both go high simultaneously, the bridge shorts supply to ground through the FETs. Main loop guards this every iteration:
>
> `if((LEFT_FORWARD_SPEED != WHEEL_OFF) && (LEFT_REVERSE_SPEED != WHEEL_OFF)) { allWheelsOff(); P1OUT |= RED_LED; }` (and the symmetric check for the right side).
>
> The motion primitives are written so they never trigger this, but the guard catches buggy command sequences from the IoT side and lights the red LED as a visible diagnostic.

Black-line FSM - 11 entry states + 4 exit states

When the operator sends `^B`, `blackLine = 1` and the main loop starts running `blackLineNavigation()` every iteration. The function is a `switch(state)` that dispatches to one of 11 per-state handlers. State transitions are driven by `blacklinetime` (the 25 ms tick counter) or by ADC threshold crossings on the two line sensors.

State	Action	Transition trigger
`RESET`	Zero blacklinetime	Immediate → WAIT1
`WAIT1`	Display "BL Start", motors off	10 s → STRAIGHT1
`STRAIGHT1`	forwardWheelsFast	1.5 s → TURN1
`TURN1`	turnRight	1 s → STRAIGHT2
`STRAIGHT2`	forwardWheelsFast	3 s → TURN2
`TURN2`	turnRight	1 s → DETECT
`DETECT`	forwardWheelsSlow, watch L/R ADCs	ADC_L ≥ 400 || ADC_R ≥ 400 → WAIT2 + show "Intercept"
`WAIT2`	allWheelsOff	10 s → ALIGN
`ALIGN`	Left-reverse 60% only (rotate onto line)	0.5 s → WAIT3
`WAIT3`	allWheelsOff	10 s → TRAVEL
`TRAVEL`	Closed-loop line follow (4 quadrant cases on L/R thresholds)	Stays here until operator sends `^E`

The opening sequence (WAIT1 → TURN2) is an open-loop dead-reckoning approach: drive forward, turn, drive forward, turn, looking for the painted line. DETECT then crawls slowly until either sensor crosses the 400-count threshold (12-bit ADC, then divided by 4 in the ISR, so 400 ≈ 4× the noise floor I measured during bring-up).

TRAVEL is the actual closed-loop follower. Four quadrant cases:

ADC_Left vs 200	ADC_Right vs 200	Interpretation	Response
Below	Below	Off the line entirely	Both reverse 30%
Below	Above	Drifted right, line to the right	Left forward 30% only (turn left)
Above	Below	Drifted left, line to the left	Right forward 30% only (turn right)
Above	Above	Centered on the line	Forward 30%/20%

Bang-bang quadrant logic, no hysteresis, no integral term. It works because the line is wide enough and the loop runs at the 25 ms tick - a real-world car would jitter at the edges; this one tolerates it because the threshold (200) sits well below the on-line value (≥400).

Exit is the `^E` mirror sequence: RESET2 → WAIT4 (10 s stop, display "BL Exit") → STRAIGHT3 (forwardWheelsSlow for 2 s, latch `finalclock` from the running stopwatch) → STOP ("BL Stop / Course Done! / Time: XXs", motors off, timer disabled).

Wi-Fi command path - AT-command stack + ring buffer parse

Connectivity is an ESP8266-class IoT module talking AT commands over UCA0 UART at 115200 baud (8 MHz BRCLK, UCBRW=4, UCBRF=5, UCBRS=0x55, oversampling on). Init walks a 6-step state machine on a 5-second cadence from main (one AT command per cadence step, giving the module time to respond):

Step	Command	Effect
`SET_IOT_EN`	Drive IOT_EN_CPU (P3.7) high	Boot the module
`STOP_SAVE`	`AT+SYSSTORE=0`	Don't persist settings to module flash
`UPDATE_CONNECT`	`AT+CIPMUX=1`	Enable multiple TCP connections (required for server mode)
`CONFIG_SERV_PORT`	`AT+CIPSERVER=1,8130`	Start TCP server on port 8130
`GET_SSID`	`AT+CWJAP?`	Query current SSID (diagnostic)
`GET_IPADD`	`AT+CIFSR`	Query IP; display "10.155.16.242"; set iot_start = TRUE

Each command is `strcpy`'d into `iot_ring_rx` (32-byte buffer doing double duty as TX queue + RX history), then UCA0 TX interrupt is enabled. The TX ISR drains the buffer one byte at a time per interrupt, stopping when it hits the `\n` terminator, then disables TX and zeros the buffer for the next command.

Inbound parse is the symmetric path. UCA0 RX ISR pushes each received byte into the ring, increments `iot_rx_count`, and wraps when `iot_rx_count >= 32`. Each byte, the ISR checks whether the byte two positions back is a carrot (`^`, 0x5E). When it matches, the 3-byte window (`^`, command, terminator) gets latched into the `IOT_Data[line][character]` 4×16 2D array. Main's `IOT_Process()` then walks that array and dispatches on the command byte.

Command map (single-byte after the carrot):

Cmd	Action	LCD echo (line 4)
`^F`	forwardWheelsFast	F0001 s
`^f`	forwardWheelsSlow	f0002 s
`^I`	Reset clock, display "Vincent Buff", start stopwatch	I0003 s
`^R`	reverseWheelsSlow	R0004 s
`^T` / `^t`	turnRight / turnLeft	T0005 s / t0006 s
`^S`	allWheelsOff	S0007 s
`^P` / `^p`	platform++/--, toggle LCD backlight by parity	P0008 s
`^B`	Set blackLine flag (start 11-state FSM)	B0009 s
`^E`	Clear blackLine, set exit flag (start 4-state exit FSM)	E0010 s

The 4×16 IOT_Data array is overkill for single-byte commands - only 3 bytes per row are ever populated. The design hedges against the future case of multi-byte commands (e.g., `^P 5` to set a specific platform) which the project never needed. Easy thing to trim if I rebuilt it; not load-bearing if I leave it.

Design decisions and trade-offs

Three limitations of the project, and the reasoning behind each.

1. **The drive actuators are brushed DC motors, not servos.** Servos take a 1-2 ms pulse-width-encoded angle and have an internal feedback loop. These are open-loop brushed DC motors driven by four PWM

duty-cycle channels into a FET-board H-bridge. Same Timer B3 in software, different actuator class. The terms blur in casual usage, but the precise description is brushed DC with PWM.

2. **The DAC is initialized but doesn't carry the line-tracking design.** `Init_DAC()` configures SAC3 as a 12-bit reference DAC (DACSREF_1 = internal Vref). The Timer B0 overflow ISR then sweeps the DAC value from 4000 down to 2200 in 200-step decrements. That sweep is a course-mandated peripheral demonstration - the line tracker would run identically without it. The DAC's actual role: it satisfies the course rubric; it does not drive line tracking.
3. **This was a junior-year intro-embedded course, not a senior-level design.** The hardware platform was course-provided across 5 board revisions. The 11-state FSM and command parser are clean but small in scale. ADC thresholds (`DET_MAX = 400` , `DET_TRAVEL = 200`) were empirical, not characterized against ambient lighting or surface variation. It shows comfort across the full bare-metal stack early in my embedded work, but it is not research-grade.

Common questions about this project

Question	Short answer
What does the boot sequence look like?	`Init_Ports` (P1-P6 config), `Init_Clocks` (DCO 8 MHz against XT1 32.768 kHz with software trim), `Init_Conditions` (clear display + enable interrupts), `Init_LEDs / Timer_B0 / Timer_B3 / REF / ADC / DAC / LCD / Serial_UCA0 / Serial_UCA1`, `allWheelsOff`. Then the while loop. Every 5 s, main steps the 6-state `Init_IOT` machine over UCA0 until the Wi-Fi module is serving on port 8130 and `iot_start` is true.
What is the timer architecture?	Two timers. `TB0` in continuous mode, SMCLK / 8 / 5 = 200 kHz, with CCR0 producing a 25 ms tick (`TB0CCR0 += 5000` each ISR). CCR1 and CCR2 handle switch-debounce timeouts (250 ms after a button press). Overflow runs the DAC sweep. `TB3` in up-mode at period 50005 drives 5 PWM channels (LCD backlight + 4 motor direction-side pairs).
How does the command parser work?	UCA0 RX ISR writes each received byte into `iot_ring_rx[32]`. After each write, it checks whether the byte two positions back is `^` (0x5E). On match, it latches the 3-byte window into `IOT_Data[line][character]`. Main's `IOT_Process` walks the IOT_Data array and dispatches on the command byte. Decoupling the raw-bytes view (ring) from the parsed-commands view (2D array) lets the ISR stay short.
How does the closed-loop line follow work?	The ADC ISR rotates through A2 (V_DETECT_L), A3 (V_DETECT_R), A5 (V_THUMB) every 25 ms - one channel per tick. Each reading is right-shifted by 2 (divide by 4) and stored globally. The TRAVEL state then picks one of four PWM responses based on whether L and R are above or below the 200 threshold: both off → reverse both; one off → turn toward the line; both on → forward 30%/20%. Bang-bang, no hysteresis, but stable because the on-line value (~≥400) sits well above the threshold.

Question	Short answer
What is the shoot-through interlock for?	Each side of the H-bridge has separate FWD and REV PWM lines. Both high at once shorts supply to ground through the FETs - fast, irreversible bridge damage. The main loop checks every iteration: if either side has both FWD and REV nonzero, kill all wheels and light the red LED. Belt and suspenders - the motion primitives don't trigger it, but the guard catches programmer error in command sequences.
How was it debugged?	TEST_PROBE GPIO (P3.0) toggled every main loop iteration so I could measure the loop period on an Analog Discovery 2 logic analyzer. UART pass-through (UCA1 RX → UCA0 TX) gave me terminal-side observability of the AT-command dialog. Five report revisions tracked bring-up by board section: power, LCD, PWM, IR detector, IoT integration - each one validated on the bench before stacking the next on top.
Why the MSP430FR2355?	Course-provided - not my pick. But the chip earns its keep: 32 KB FRAM (no separate flash sector for non-volatile storage, no erase cycles to plan around), low-power DCO + XT1 mix, and a generous peripheral set (eUSCI_A0/A1 UART + eUSCI_B1 SPI, 2× 12-bit ADC capable, 4-channel timers with capture/compare). I'd reach for it again on an embedded project that needs UART + SPI + ADC + a few PWM channels under a 100 mA budget.
What would change in a rebuild?	(a) Decouple ADC sampling from the display tick - sample faster, update LCD slower; line follow would respond quicker. (b) Add hysteresis on the line-detection threshold so the TRAVEL quadrant logic doesn't chatter at edges. (c) Swap the bang-bang quadrant for a small PI loop on (ADC_R – ADC_L) - actual proportional steering. (d) Either give the DAC a real job or remove it from the build entirely. (e) Trim the 4×16 IOT_Data to single-byte commands; the buffer overhead isn't paying for itself.