Zephyr Shell on STM32F4 Discovery

Project Overview

- **Purpose-Built UART Shell:** Implements a custom command-line shell using Zephyr RTOS on STM32F4, providing direct access to GPIOs and system state for debugging and control.
- **Target Hardware: STM32F4 Discovery:** Utilizes USART2 at 115200 baud over ST-LINK VCP or USB-UART for shell interface, compatible with most STM32F407 development boards.
- **Key Shell Commands:** Includes LED control, uptime tracking, thread monitoring, and raw memory peek using 32-bit word reads. Registered under module `td`.
- **Debugging and QoL Features:** Supports Zephyr's logging, stack checks, thread naming, and assert mechanisms to facilitate in-field diagnostics and development.
- **Extendable Command Set:** Designed to be modular: new shell commands can be added easily using Zephyr's SHELL_CMD registration pattern.

Hardware + Zephyr Setup

Board, Pinout, and Build Tools

- **Target: STM32F4 Discovery Board:** Utilizes STM32F407VG with a Cortex-M4 core. Peripheral access via USART2 (PA2/PA3) mapped to ST-LINK VCP or USB-UART dongle.
- **Overlay for Console Routing:** Device tree overlay redirects shell and console to &usart2. Ensures compatibility with serial terminal tools like minicom and screen.
- **Pinmux and Baud Rate Setup:** Pin control explicitly defined for TX (PA2) and RX (PA3). Baud rate set to 115200 to match ST-LINK VCP defaults.
- **Build & Flash Commands:** Uses west toolchain: `west build -b stm32f4_disco` and `west flash`. Automatically flashes via ST-LINK if connected.
- **Zephyr Configuration via prj.conf:** Enables UART, shell backend, and thread monitoring via Kconfig options. Also provisions heap and stack for shell operation.

Zephyr Shell Configuration

UART Integration and Serial Shell

- **Shell Over UART via USART2:** Shell is routed over USART2 with pinmux set to PA2 (TX) and PA3 (RX), allowing connection via ST-LINK VCP or external USB-UART adapter.
- **Shell Prompt and Built-ins:** Custom shell prompt set to 'f407>'. Includes built-in commands: kernel, device, log, uptime, etc., alongside user-defined ones.
- **Shell Configurations (Kconfig):** Enabled via `CONFIG_SHELL`, `CONFIG_SHELL_BACKEND_SERIAL`, and `CONFIG_SHELL_CMDS_BUILTIN`. Ensures core command access without custom code.
- **Serial Terminal Setup:** Use tools like minicom or screen at 115200 baud, 8N1 config to access shell. Immediate feedback and full command-line interaction.
- **Extensibility and Modular Shell:** Zephyr shell design allows registering command modules via `SHELL_CMD_REGISTER`. Commands are grouped under a namespace (e.g., `td`).

LED Control via Shell

GPIO Management Through Commands

- **GPIO Binding via devicetree alias:** Uses `DT_ALIAS(led0)` and `GPIO_DT_SPEC_GET()` to access the LED pin through structured Zephyr GPIO APIs.
- **Shell Command: td led:** Accepts `on`, `off`, or `toggle` arguments. Routes to `gpio_pin_set_dt()` or `gpio_pin_toggle_dt()` accordingly.
- **Error Checking and Feedback:** Provides usage guidance and error messages for invalid input. Shell output confirms action (e.g., LED: on).
- **GPIO Initialization on Boot:** In `main()`, LED pin is configured as inactive output and checked with `device_is_ready()`. Ensures hardware readiness.
- **Application Example:** Run `td led toggle` or `td led off` to control the onboard LED in real-time from a serial shell interface.

Uptime & Thread Monitoring

Zephyr Shell Tools for Runtime Introspection

- **Uptime Tracking (ms & ns):** Shell command `td uptime` prints milliseconds via `k_uptime_get()` and nanoseconds using `k_cyc_to_ns_floor64()`.
- **Thread Overview Command:** `td threads` lists thread names, priorities, states, and unused stack memory via `k_thread_foreach()` callback.
- **Thread Stack Usage Insight:** If `CONFIG_THREAD_STACK_INFO` is enabled, unused stack space is calculated per thread and reported to the shell.
- **Thread Naming & Monitoring:** Enabled via `CONFIG_THREAD_NAME` and `CONFIG_THREAD_MONITOR`. Threads without names are labeled '(anon)'.
- **Diagnostic Shell Feedback:** Thread state and priority are shown in a structured format to simplify performance debugging and RTOS introspection.

Memory Inspection with peek32

Reading Raw 32-bit Words from Address

- **Command: td peek32 [count]:** Allows inspection of memory-mapped peripherals or RAM using base address and optional count of 32-bit reads.
- **Direct Memory Access:** Converts string to `uintptr_t`, casts to `uint32_t*`, and reads volatile word at computed offset.
- **Shell-Safe Output:** Prints address and value in hex format: `[0xADDR] = 0xVALUE`, aiding debugging of registers or buffers.
- **Fault Risk Warning:** Invalid or restricted addresses can lead to bus faults; caution required when peeking at low-level hardware addresses.
- **Use Case: Debugging Peripherals:** Useful for reading timers, GPIOs, or peripheral registers directly from shell without needing IDE or debugger.

Built-in Shell Commands

Leverage Zephyr’s Core Shell Toolkit

- **Preloaded Command Set:** Shell comes with built-in commands like `kernel`, `device`, `uptime`, `log`, and `history`, providing essential diagnostics.
- **System Introspection:** Commands such as `kernel version`, `device list`, and `log enable` help examine the RTOS state and driver registration.
- **Command Discovery via help:** Use the `help` command to explore available built-ins and their syntax. Organizes commands by module.
- **History and Repeat:** Shell tracks previously entered commands and allows repetition or editing, improving developer productivity.
- **No Code Needed:** These utilities work out of the box via `CONFIG_SHELL_CMDS_BUILTIN=y` — no manual coding or registration required.

RTT & UART Shell Backends

Choosing Your Debug Interface

- **UART: Default Shell Backend:** Uses USART2 mapped to PA2/PA3 for shell interface. Compatible with minicom, screen, or Tera Term at 115200 baud.
- **RTT: Memory-Mapped Debug Channel:** SEGGER RTT offers high-speed debugging over J-Link interface using memory-mapped buffers and no serial port.
- **Enabling RTT in Zephyr:** Activate `CONFIG_USE_SEGGER_RTT` and `CONFIG_SHELL_BACKEND_RTT`. No need for UART wiring if using J-Link.
- **Viewing RTT Output:** Compatible with SEGGER RTT Viewer or OpenOCD. Useful in setups where UART is unavailable or busy.
- **Hybrid Debug Options:** Both UART and RTT can be compiled in with multiple shell backends if needed for flexible debugging scenarios.

Security & Deployment Considerations

Locking Down the Shell for Production

- **Shell is Powerful:** Exposes system internals, thread state, memory access, and GPIO control — valuable during development, risky in production.
- **Disable via Kconfig:** Use `CONFIG_SHELL=n` in production builds to entirely remove shell backend and associated attack surfaces.
- **Authentication Options:** Consider wrapping shell access with custom authentication modules or conditionally compiling based on build type.
- **Segmentation by Build Profile:** Use different `.conf` overlays for debug vs. production profiles to manage shell availability and system exposure.
- **Physical Interface Risk:** Shell over UART or RTT can be accessed with physical tools — secure hardware, use tamper-proof casings in critical apps.

Conclusion & Extension Ideas

Next Steps with Zephyr Shell

- **Add More Commands:** Extend the shell using `SHELL_CMD()` macros to add I2C scanning, SPI transfers, watchdog control, reboot, and heap stats.
- **Structure Commands into Modules:** Group related commands under namespaces like `sensor`, `power`, or `net` to scale shell complexity cleanly.
- **Use Shell for Automation:** Automate tests, board validation, and stress tests by scripting shell interaction through UART or RTT logging tools.
- **Integrate with Logging System:** Combine shell and log output using Zephyr’s logging backend to trace events and correlate with command output.
- **Document for Team Use:** Create internal docs or cheat sheets to help engineers quickly use and extend shell utilities during development.