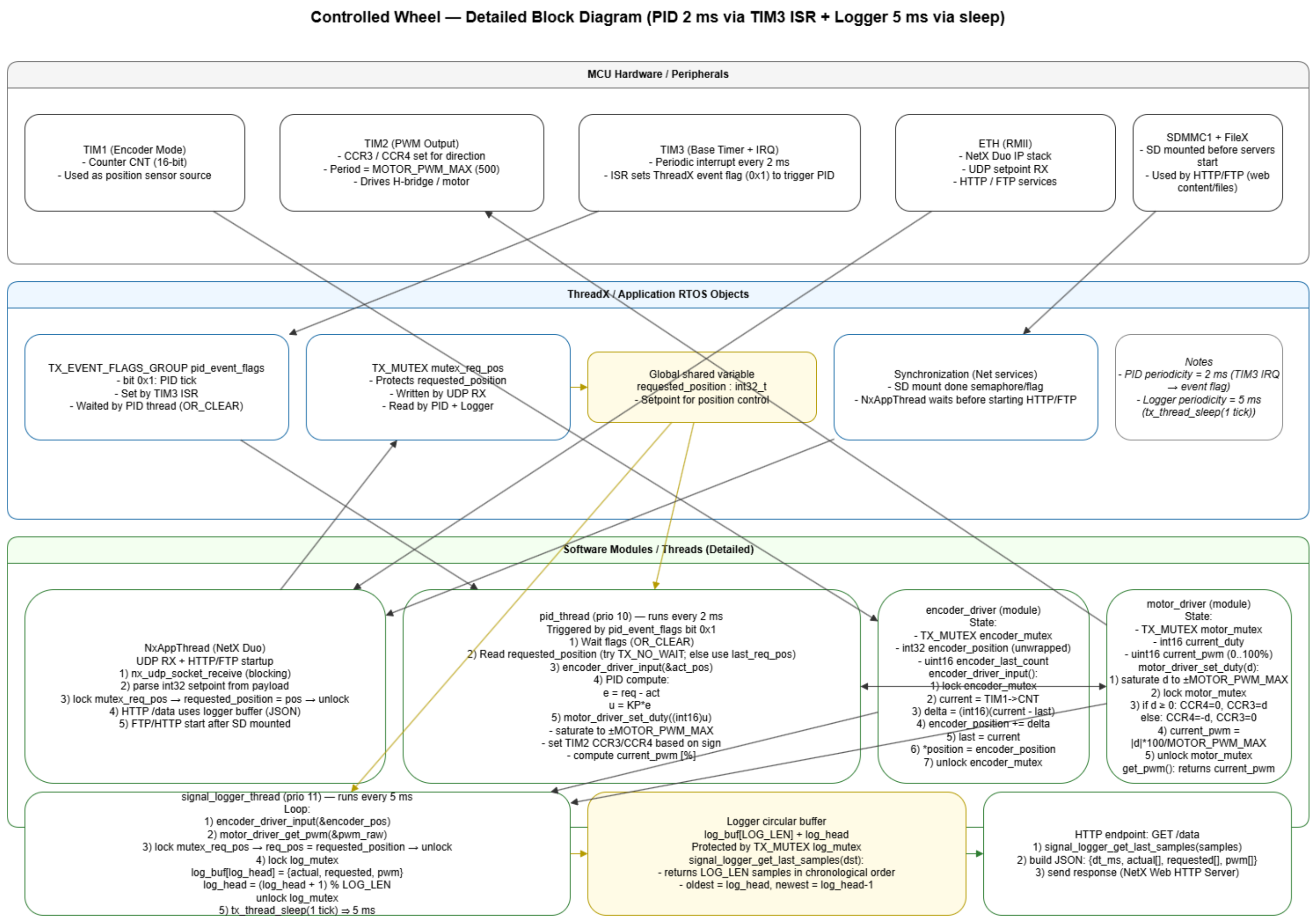
CONTROLLED WHEEL

Short textual description of the application

This project implements a real-time controlled wheel system based on an STM32 microcontroller and the ThreadX RTOS.
The application controls the angular position of a DC motor using a periodic PID controller running every 2 ms. The motor position is measured through a quadrature encoder, while the control output is applied using PWM signals.
The reference position is received asynchronously via UDP packets over Ethernet using the NetX Duo stack.
Additionally, the system includes a logging subsystem that periodically samples the motor position, reference position, and control output every 5 ms. These data are made available through an embedded HTTP server in JSON format for real-time monitoring and debugging.

Application data flow


Controlled Wheel — Detailed Block Diagram (PID 2 ms via TIM3 ISR + Logger 5 ms via sleep)
MCU Hardware / Peripherals
TIM1 (Encoder Mode) - Counter CNT (16-bit) - Used as position sensor source
TIM2 (PWM Output) - CCR3 / CCR4 set for direction - Period = MOTOR_PWM_MAX (500) - Drives H-bridge / motor
TIM3 (Base Timer + IRQ) - Periodic interrupt every 2 ms - ISR sets ThreadX event flag (0x1) to trigger PID
ETH (RMII) - NetX Duo IP stack - UDP setpoint RX - HTTP / FTP services
SDMMC1 + FileX - SD mounted before servers start - Used by HTTP/FTP (web content/files)
ThreadX / Application RTOS Objects
TX_EVENT_FLAGS_GROUP pid_event_flags - bit 0x1: PID tick - Set by TIM3 ISR - Waited by PID thread (OR_CLEAR)
TX_MUTEX mutex_req_pos - Protects requested_position - Written by UDP RX - Read by PID + Logger
Global shared variable requested_position : int32_t - Setpoint for position control
Synchronization (Net services) - SD mount done semaphore/flag - NxAppThread waits before starting HTTP/FTP
Notes - PID periodicity = 2 ms (TIM3 IRQ → event flag) - Logger periodicity = 5 ms (tx_thread_sleep(1 tick))
Software Modules / Threads (Detailed)
NxAppThread (NetX Duo) UDP RX + HTTP/FTP startup 1) nx_udp_socket_receive (blocking) 2) parse int32 setpoint from payload 3) lock mutex_req_pos → requested_position = pos → unlock 4) HTTP /data uses logger buffer (JSON) 5) FTP/HTTP start after SD mounted
pid_thread (prio 10) — runs every 2 ms Triggered by pid_event_flags bit 0x1 1) Wait flags (OR_CLEAR) 2) Read requested_position (try TX_NO_WAIT; else use last_req_pos) 3) encoder_driver_input(&act_pos) 4) PID compute: e = req - act u = KP*e 5) motor_driver_set_duty((int16)u) - saturate to ±MOTOR_PWM_MAX - set TIM2 CCR3/CCR4 based on sign - compute current_pwm [%]
encoder_driver (module) State: - TX_MUTEX encoder_mutex - int32 encoder_position (unwrapped) - uint16 encoder_last_count encoder_driver_input(): 1) lock encoder_mutex 2) current = TIM1->CNT 3) delta = (int16)(current - last) 4) encoder_position += delta 5) last = current 6) *position = encoder_position 7) unlock encoder_mutex
motor_driver (module) State: - TX_MUTEX motor_mutex - int16 current_duty - uint16 current_pwm (0..100%) motor_driver_set_duty(d): 1) saturate d to ±MOTOR_PWM_MAX 2) lock motor_mutex 3) if d ≥ 0: CCR4=0, CCR3=d else: CCR4=-d, CCR3=0 4) current_pwm = |d|*100/MOTOR_PWM_MAX 5) unlock motor_mutex get_pwm(): returns current_pwm
signal_logger_thread (prio 11) — runs every 5 ms Loop: 1) encoder_driver_input(&encoder_pos) 2) motor_driver_get_pwm(&pwm_raw) 3) lock mutex_req_pos → req_pos = requested_position → unlock 4) lock log_mutex log_buf[log_head] = {actual, requested, pwm} log_head = (log_head + 1) % LOG_LEN unlock log_mutex 5) tx_thread_sleep(1 tick) ⇒ 5 ms
Logger circular buffer log_buf[LOG_LEN] + log_head Protected by TX_MUTEX log_mutex signal_logger_get_last_samples(dst): - returns LOG_LEN samples in chronological order - oldest = log_head, newest = log_head-1
HTTP endpoint: GET /data 1) signal_logger_get_last_samples(samples) 2) build JSON: {dt_ms, actual[], requested[], pwm[]} 3) send response (NetX Web HTTP Server)

The data flow of the application is organized around a periodic real-time control loop and several asynchronous components.
A hardware timer (TIM3) generates a periodic interrupt every 2 ms. Inside the interrupt service routine, a ThreadX event flag is set to release the PID control thread, which therefore executes with a strictly timer-driven periodicity.

The desired motor position (setpoint) is received asynchronously through the Ethernet interface. A NetX Duo thread listens for incoming UDP packets and extracts the requested position from the received payload. The setpoint is stored in a global shared variable, and access to this variable is protected by a mutex to ensure data consistency between the network thread and the real-time control tasks.

When the PID thread is activated, it reads the most recent requested position from the shared variable and acquires the actual motor position through the encoder driver. The encoder driver reads the hardware encoder counter and performs wrap-around handling to reconstruct a continuous position value.

Based on the position error, the PID controller computes the control action and sends the resulting duty cycle to the motor driver. The motor driver applies the command to the PWM hardware channels, selecting the rotation direction and duty cycle used to drive the motor.

In parallel with the control loop, a logger thread runs periodically every 5 ms using a ThreadX sleep mechanism. This thread samples the actual position, the requested position, and the applied PWM duty cycle and stores the collected data in a circular buffer.

An embedded HTTP server, running within the NetX Duo stack, accesses the logger buffer upon client requests and returns the most recent samples in JSON format. This mechanism enables real-time monitoring and debugging of the controlled wheel system through a web interface.

3. Description of Global Functions and Variables

This section describes the main global variables and application-specific functions implemented in the project, including their purpose, parameters, and return values.

Global Variables and RTOS Objects

requested_position (int32_t)
Global variable representing the desired motor position (setpoint).
It is updated asynchronously by the UDP receiver thread and read by the PID control thread and the logger thread. Access to this variable is protected by a mutex to prevent concurrent access issues.

mutex_req_pos (TX_MUTEX)
Mutex used to protect access to requested_position.
The PID thread attempts to acquire this mutex in non-blocking mode to preserve real-time behavior; if the mutex is unavailable, the previously used setpoint is reused.

pid_event_flags (TX_EVENT_FLAGS_GROUP)
Event flag group used to synchronize the PID control thread with the periodic timer interrupt. The flag is set every 2 ms inside the TIM3 interrupt service routine and waited upon by the PID thread, ensuring a timer-driven execution of the control loop.

PID Control Thread

pid_thread_entry(ULONG init)
Thread entry function implementing the real-time PID controller.

- **Parameter:** init (unused ThreadX initialization parameter).
- **Description:**
  The thread waits for a periodic event flag generated by the timer interrupt, reads the requested and actual motor positions, computes the PID control law, and applies the resulting duty cycle to the motor driver.
- **Return value:** None (infinite thread loop).

Encoder Driver

encoder_driver_initialize(void)
Initializes the encoder driver by creating a mutex and resetting internal state variables.
Returns TX_SUCCESS on success or a ThreadX error code otherwise.

encoder_driver_input(int32_t *position)
Reads the current encoder position.

- **Parameter:** pointer to an int32_t variable where the absolute position is stored.
- **Description:**
  The function reads the hardware encoder counter, handles wrap-around using 16-bit arithmetic, updates an internal unwrapped position, and returns a continuous position value.
- **Return value:** TX_SUCCESS on success, ThreadX error code on failure.

Motor Driver

motor_driver_initialize(void)
Initializes the motor driver and creates a mutex protecting motor control variables.
Returns TX_SUCCESS or a ThreadX error code.

motor_driver_set_duty(int16_t duty)
Applies a signed duty cycle command to the motor.

- **Parameter:** duty, whose sign determines rotation direction and magnitude determines PWM duty cycle.
- **Description:**
  The duty value is saturated, translated into PWM signals on the appropriate timer channels, and stored internally for monitoring purposes.
- **Return value:** TX_SUCCESS on success, ThreadX error code otherwise.

motor_driver_get_duty(int16_t *duty_out)
Returns the last applied duty cycle.
Returns TX_SUCCESS or a ThreadX error code.

motor_driver_get_pwm(uint16_t *pwm_out)
Returns the applied PWM duty cycle expressed as a percentage.
Returns TX_SUCCESS or a ThreadX error code.

Signal Logger

signal_logger_init(void)
Initializes the logging module by resetting the circular buffer and creating the associated mutex.
Returns TX_SUCCESS or a ThreadX error code.

signal_logger_thread_entry(ULONG thread_input)
Thread entry function implementing periodic signal logging.

- **Parameter:** thread_input (unused).
- **Description:**
  Every 5 ms, the thread samples the actual motor position, the requested position, and the applied PWM duty cycle, storing the values in a circular buffer.
- **Return value:** None.

signal_logger_get_last_samples(sample_t *dst)
Copies the most recent logged samples into the destination array in chronological order.

- **Parameter:** pointer to an array of samples.
- **Return value:** TX_SUCCESS on success, TX_PTR_ERROR if the pointer is invalid.

Network-Related Functions

nx_app_thread_entry(ULONG thread_input)
Main NetX Duo application thread.

- **Description:**
  Initializes FTP and HTTP services, creates and binds a UDP socket, receives setpoint commands via UDP packets, and updates the global requested_position variable using a mutex.
- **Return value:** None.

http_request_notify(...)
Callback function handling HTTP requests.

- **Description:**
  Implements the /data endpoint, which retrieves the most recent logged samples and returns them in JSON format for real-time monitoring.
- **Return value:** NetX status code.