File: HMI_feedback_gen.c

```c
/**
 * @file feedback_generation.c
 * @brief Feedback Generation Subcomponent for HMI Interaction Module
 * @author Grok 3 (xAI)
 * @date March 09, 2025
 */

#include <stdint.h>
#include <stdbool.h>

// Operating modes (assumed from mode_selection.c)
typedef enum {
    MODE_EV,
    MODE_HYBRID,
    MODE_CHARGE_SUSTAINING
} drive_mode_t;

// Constants
#define HAPTIC_PULSE_DURATION_MS 200     // Duration of haptic feedback pulse
#define SOUND_FREQ_BASE_HZ       100     // Base frequency for synthetic sound
#define EFFICIENCY_THRESHOLD     0.8f    // Efficiency threshold for feedback

// State variables
static drive_mode_t last_mode = MODE_EV;
static bool haptic_active = false;

/**
 * @brief Generate haptic feedback for mode changes or efficiency tips
 * @param current_mode Current drive mode
 * @param efficiency Current system efficiency (0.0 to 1.0)
 */
static void generate_haptic_feedback(drive_mode_t current_mode, float efficiency) {
    if (current_mode != last_mode || efficiency < EFFICIENCY_THRESHOLD) {
        haptic_active = true;
        set_haptic_pulse(HAPTIC_PULSE_DURATION_MS);
        last_mode = current_mode;
    } else {
        haptic_active = false;
    }
}

/**
 * @brief Generate synthetic engine sounds based on powertrain operation
 * @param ice_rpm ICE speed (RPM)
 * @param motor_rpm Motor speed (RPM)
 * @param mode Current drive mode
 */
static void generate_auditory_cues(uint16_t ice_rpm, uint16_t motor_rpm, drive_mode_t mode) {
    float sound_freq_hz = SOUND_FREQ_BASE_HZ;

    if (mode == MODE_EV) {
```

```c
        sound_freq_hz += motor_rpm * 0.05f;  // Motor-dominated sound
    } else if (mode == MODE_HYBRID) {
        sound_freq_hz += (ice_rpm * 0.03f + motor_rpm * 0.02f);  // Blended sound
    } else {
        sound_freq_hz += ice_rpm * 0.04f;  // ICE-dominated sound
    }

    set_audio_frequency(sound_freq_hz);
}

/**
 * @brief Main feedback generation execution function
 * @param mode Current drive mode
 * @param ice_rpm ICE speed (RPM)
 * @param motor_rpm Motor speed (RPM)
 * @param efficiency Current system efficiency (0.0 to 1.0)
 */
void feedback_generation_execute(drive_mode_t mode, uint16_t ice_rpm, uint16_t motor_rpm, float efficiency) {
    generate_haptic_feedback(mode, efficiency);
    generate_auditory_cues(ice_rpm, motor_rpm, mode);
}

// External function prototypes (assumed implemented elsewhere)
extern void set_haptic_pulse(uint32_t duration_ms);
extern void set_audio_frequency(float frequency_hz);
```

File: battery_management_interface.c

```c
/**
 * @file battery_management_interface.c
 * @brief Battery Management Interface Module for Hybrid Vehicle ECU
 * @author Grok 3 (xAI)
 * @date March 09, 2025
 */

#include <stdint.h>
#include <stdbool.h>

// Constants
#define MAX_CHARGE_CURRENT_A    100.0f   // Maximum charge current
#define MAX_DISCHARGE_CURRENT_A 150.0f   // Maximum discharge current
#define SOH_THRESHOLD           80.0f    // State of health warning threshold (%)

// State variables
static float soc = 100.0f;  // State of charge (%)
static float soh = 100.0f;  // State of health (%)
static float current_limit_a = MAX_DISCHARGE_CURRENT_A;

/**
 * @brief Estimate SOC from BMS data
 * @param bms_voltage Battery voltage (V)
 * @param bms_current Battery current (A, positive for discharge)
 */
static void estimate_soc(float bms_voltage, float bms_current) {
    // Simple SOC estimation (assumes BMS provides raw data)
    soc -= (bms_current * 0.1f) / 3600.0f;  // Rough integration, assuming 1s update
    soc = (soc > 100.0f) ? 100.0f : (soc < 0.0f) ? 0.0f : soc;
}

/**
 * @brief Monitor battery state of health
 * @param cycle_count Number of charge cycles
 * @param internal_resistance Battery internal resistance (mOhm)
 */
static void monitor_soh(uint32_t cycle_count, float internal_resistance) {
    // Simple SOH degradation model
    soh = 100.0f - (cycle_count * 0.01f) - (internal_resistance * 0.05f);
    if (soh < SOH_THRESHOLD) {
        set_diagnostic_code(DTC_BATTERY_SOH);
    }
}

/**
 * @brief Control charge/discharge limits
 * @param soc Current state of charge (%)
 * @param temp_c Battery temperature (°C)
 */
static void control_charge_discharge(float soc, float temp_c) {
    if (soc > 95.0f) {
        current_limit_a = MAX_CHARGE_CURRENT_A * 0.5f;  // Reduce charging near full
```

```c
    } else if (soc < 10.0f) {
           current_limit_a = MAX_DISCHARGE_CURRENT_A * 0.5f;  // Reduce discharge near
empty
    } else {
        current_limit_a = (temp_c > 40.0f) ?
                         MAX_DISCHARGE_CURRENT_A * 0.8f : MAX_DISCHARGE_CURRENT_A;
    }
}

/**
 * @brief Main battery management execution function
 * @param bms_voltage Battery voltage (V)
 * @param bms_current Battery current (A)
 * @param cycle_count Number of charge cycles
 * @param internal_resistance Battery internal resistance (mOhm)
 * @param temp_c Battery temperature (°C)
 */
void battery_management_execute(float bms_voltage, float bms_current, uint32_t
cycle_count,
                               float internal_resistance, float temp_c) {
    estimate_soc(bms_voltage, bms_current);
    monitor_soh(cycle_count, internal_resistance);
    control_charge_discharge(soc, temp_c);

    set_current_limit(current_limit_a);
}

// External function prototypes (assumed implemented elsewhere)
extern void set_current_limit(float current_a);
extern void set_diagnostic_code(uint16_t dtc);
```

File: cloud_data_integration.c

```c
/**
 * @file cloud_data_integration.c
 * @brief Cloud Data Integration Subcomponent for OTA Update and Cloud Integration Module
 * @author Grok 3 (xAI)
 * @date March 09, 2025
 */

#include <stdint.h>
#include <stdbool.h>

// Constants
#define MAP_UPDATE_INTERVAL_S   300      // Map update interval (5 minutes)
#define LOG_BUFFER_SIZE         64       // Size of performance log buffer

// State variables
static uint32_t last_map_update_time_s = 0;
static uint8_t log_buffer[LOG_BUFFER_SIZE];
static uint16_t log_index = 0;

/**
 * @brief Update maps with real-time cloud data
 * @param current_time_s Current system time (seconds)
 */
static void update_maps(uint32_t current_time_s) {
    if (current_time_s - last_map_update_time_s >= MAP_UPDATE_INTERVAL_S) {
        cloud_map_data_t map_data;
        if (fetch_cloud_map_data(&map_data)) {
            set_traffic_data(map_data.traffic_level);
            set_weather_data(map_data.weather_condition);
            set_charging_stations(map_data.station_locations, map_data.station_count);
            last_map_update_time_s = current_time_s;
        }
    }
}

/**
 * @brief Log powertrain performance data to cloud
 * @param ice_torque ICE torque (Nm)
 * @param motor_torque Motor torque (Nm)
 * @param soc Battery state of charge (%)
 */
static void log_performance(float ice_torque, float motor_torque, float soc) {
    if (log_index < LOG_BUFFER_SIZE - 12) {  // Ensure space for 3 floats (12 bytes)
        *(float*)&log_buffer[log_index] = ice_torque;
        *(float*)&log_buffer[log_index + 4] = motor_torque;
        *(float*)&log_buffer[log_index + 8] = soc;
        log_index += 12;
    }

    if (log_index >= LOG_BUFFER_SIZE - 12 || is_cloud_connected()) {
        upload_performance_data(log_buffer, log_index);
```

```c
        log_index = 0;  // Reset buffer
    }
}

/**
 * @brief Main cloud data integration execution function
 * @param current_time_s Current system time (seconds)
 * @param ice_torque ICE torque (Nm)
 * @param motor_torque Motor torque (Nm)
 * @param soc Battery state of charge (%)
 */
void  cloud_data_integration_execute(uint32_t  current_time_s,  float  ice_torque,  float
motor_torque, float soc) {
    update_maps(current_time_s);
    log_performance(ice_torque, motor_torque, soc);
}

// External function prototypes (assumed implemented elsewhere)
typedef struct {
    uint8_t traffic_level;           // 0-100 congestion level
    uint8_t weather_condition;       // Enum-like value (e.g., 0=sunny, 1=rain)
    uint16_t* station_locations;     // Array of charging station coordinates
    uint8_t station_count;           // Number of stations
} cloud_map_data_t;

extern bool fetch_cloud_map_data(cloud_map_data_t* data);
extern void set_traffic_data(uint8_t traffic_level);
extern void set_weather_data(uint8_t weather_condition);
extern void set_charging_stations(uint16_t* locations, uint8_t count);
extern bool is_cloud_connected(void);
extern void upload_performance_data(uint8_t* data, uint16_t size);
```

File: display_integration.c

```c
/**
 * @file display_integration.c
 * @brief Display Integration Subcomponent for HMI Interaction Module
 * @author Grok 3 (xAI)
 * @date March 09, 2025
 */

#include <stdint.h>
#include <stdbool.h>

// Operating modes (assumed from mode_selection.c)
typedef enum {
    MODE_EV,
    MODE_HYBRID,
    MODE_CHARGE_SUSTAINING
} drive_mode_t;

// Constants
#define EFFICIENCY_TIP_THRESHOLD 0.7f    // Threshold for efficiency coaching

/**
 * @brief Visualize power flow on dashboard display
 * @param ice_torque ICE torque (Nm)
 * @param motor_torque Motor torque (Nm)
 * @param battery_power Battery power (W, positive for discharge)
 * @param mode Current drive mode
 */
static void visualize_power_flow(float ice_torque, float motor_torque, float battery_power, drive_mode_t mode) {
    uint8_t flow_data[3];  // [ICE, Motor, Battery] as percentage of max

    flow_data[0] = (uint8_t)(ice_torque * 100.0f / MAX_ICE_TORQUE_NM);
    flow_data[1] = (uint8_t)(motor_torque * 100.0f / MAX_MOTOR_TORQUE_NM);
    flow_data[2] = (battery_power > 0.0f) ?
                   (uint8_t)(battery_power * 100.0f / MAX_BATTERY_POWER_W) :
                   (uint8_t)(-battery_power * 100.0f / MAX_BATTERY_POWER_W);

    set_display_power_flow(flow_data, mode);
}

/**
 * @brief Provide efficiency coaching tips
 * @param efficiency Current system efficiency (0.0 to 1.0)
 * @param driver_demand Driver torque demand (%)
 */
static void coach_efficiency(float efficiency, float driver_demand) {
    if (efficiency < EFFICIENCY_TIP_THRESHOLD && driver_demand > 70.0f) {
        set_display_message("Ease off throttle for better efficiency");
    } else if (efficiency > 0.9f) {
        set_display_message("Great driving! Keep it up!");
    } else {
        set_display_message("");  // Clear message
```

```
    }
}

/**
 * @brief Main display integration execution function
 * @param ice_torque ICE torque (Nm)
 * @param motor_torque Motor torque (Nm)
 * @param battery_power Battery power (W)
 * @param mode Current drive mode
 * @param efficiency Current system efficiency (0.0 to 1.0)
 * @param driver_demand Driver torque demand (%)
 */
void display_integration_execute(float ice_torque, float motor_torque, float battery_power,
                                             drive_mode_t mode, float efficiency, float driver_demand) {
    visualize_power_flow(ice_torque, motor_torque, battery_power, mode);
    coach_efficiency(efficiency, driver_demand);
}

// External function prototypes (assumed implemented elsewhere)
extern void set_display_power_flow(uint8_t flow_data[3], drive_mode_t mode);
extern void set_display_message(const char* message);

// Assumed constants from other modules
#define MAX_ICE_TORQUE_NM       300.0f
#define MAX_MOTOR_TORQUE_NM     200.0f
#define MAX_BATTERY_POWER_W     50000.0f  // Example max battery power (50 kW)
```

File: emission_control.c

```c
/**
 * @file emission_control.c
 * @brief Emission Control Module for Vehicle ECU
 * @author Grok 3 (xAI)
 * @date March 09, 2025
 */

#include <stdint.h>
#include <stdbool.h>
#include "emission_control.h"
#include "sensor_interface.h"
#include "actuator_interface.h"

// Constants
#define TARGET_LAMBDA           1.0f      // Stoichiometric AFR (14.7:1 for gasoline)
#define CATALYST_TEMP_MIN_C     300.0f    // Minimum catalyst operating temp
#define CATALYST_TEMP_MAX_C     900.0f    // Maximum safe catalyst temp
#define EGR_MAX_RATE            20.0f     // Maximum EGR percentage
#define EVAP_PURGE_DUTY_MAX     80.0f     // Maximum purge valve duty cycle

// Structure to hold emission control state
typedef struct {
    float lambda_actual;          // Current measured lambda
    float fuel_trim_short;        // Short-term fuel trim (%)
    float fuel_trim_long;         // Long-term fuel trim (%)
    float catalyst_temp_c;        // Catalyst temperature
    float o2_sensor_pre_v;        // Pre-catalyst O2 sensor voltage
    float o2_sensor_post_v;       // Post-catalyst O2 sensor voltage
    float egr_rate_pct;           // Current EGR rate
    float evap_purge_duty;        // EVAP purge valve duty cycle
    bool catalyst_efficient;      // Catalyst efficiency status
} emission_state_t;

// Private variables
static emission_state_t emission_state = {0};

/**
 * @brief Manage air-fuel ratio using O2 sensor feedback
 * @param engine_load Current engine load (%)
 * @param engine_rpm Current engine speed (RPM)
 */
static void manage_air_fuel_ratio(float engine_load, uint16_t engine_rpm) {
    float lambda_error;
    const float kp = 0.1f;  // Proportional gain
    const float ki = 0.01f; // Integral gain

    // Calculate lambda error
    lambda_error = TARGET_LAMBDA - emission_state.lambda_actual;

    // Update fuel trims (simple PI control)
    emission_state.fuel_trim_short = kp * lambda_error;
    emission_state.fuel_trim_long += ki * lambda_error;
```

```
    // Limit fuel trims
    emission_state.fuel_trim_short = (emission_state.fuel_trim_short > 25.0f) ?
                                    25.0f : ((emission_state.fuel_trim_short < -25.0f) ?
                                    -25.0f : emission_state.fuel_trim_short);
    emission_state.fuel_trim_long = (emission_state.fuel_trim_long > 25.0f) ?
                                   25.0f : ((emission_state.fuel_trim_long < -25.0f) ?
                                   -25.0f : emission_state.fuel_trim_long);

    // Apply fuel correction
              float    total_trim    =    1.0f    +    (emission_state.fuel_trim_short    +
emission_state.fuel_trim_long) / 100.0f;
             set_fuel_injection_pulse(total_trim    *    calculate_base_pulse(engine_load,
engine_rpm));
}

/**
 * @brief Monitor catalytic converter performance
 */
static void monitor_catalyst(void) {
    // Check catalyst temperature
    bool temp_ok = (emission_state.catalyst_temp_c >= CATALYST_TEMP_MIN_C &&
                   emission_state.catalyst_temp_c <= CATALYST_TEMP_MAX_C);

    // Simple catalyst efficiency check using O2 sensor oscillation
                    float     o2_amplitude     =     emission_state.o2_sensor_pre_v     -
emission_state.o2_sensor_post_v;
    emission_state.catalyst_efficient = temp_ok && (o2_amplitude < 0.1f);

    if (!emission_state.catalyst_efficient) {
        set_diagnostic_code(DTC_CATALYST_EFFICIENCY);
    }
}

/**
 * @brief Control EGR system based on operating conditions
 * @param engine_load Current engine load (%)
 * @param coolant_temp_c Engine coolant temperature (°C)
 */
static void control_egr(float engine_load, float coolant_temp_c) {
    // Only enable EGR under certain conditions
    if (coolant_temp_c > 70.0f && engine_load > 20.0f && engine_load < 80.0f) {
        // Simple EGR rate calculation based on load
        emission_state.egr_rate_pct = (engine_load - 20.0f) * (EGR_MAX_RATE / 60.0f);
        emission_state.egr_rate_pct = (emission_state.egr_rate_pct > EGR_MAX_RATE) ?
                                     EGR_MAX_RATE : emission_state.egr_rate_pct;
    } else {
        emission_state.egr_rate_pct = 0.0f;
    }

    set_egr_valve_position(emission_state.egr_rate_pct);
}

/**
```

```
 * @brief Control EVAP canister purge
 * @param engine_load Current engine load (%)
 * @param coolant_temp_c Engine coolant temperature (°C)
 */
static void control_evap(float engine_load, float coolant_temp_c) {
    // Only purge under certain conditions
     if (coolant_temp_c > 60.0f && engine_load > 15.0f && emission_state.lambda_actual <
1.05f) {
        emission_state.evap_purge_duty = engine_load * 2.0f;  // Simple linear mapping
                   emission_state.evap_purge_duty  =  (emission_state.evap_purge_duty  >
EVAP_PURGE_DUTY_MAX) ?
                                                                    EVAP_PURGE_DUTY_MAX  :
emission_state.evap_purge_duty;
    } else {
        emission_state.evap_purge_duty = 0.0f;
    }

    set_evap_purge_valve(emission_state.evap_purge_duty);
}

/**
 * @brief Main emission control execution function
 * @param inputs Pointer to emission control inputs structure
 */
void emission_control_execute(emission_inputs_t* inputs) {
    // Update state from sensors
    emission_state.lambda_actual = inputs->lambda;
    emission_state.catalyst_temp_c = inputs->catalyst_temp_c;
    emission_state.o2_sensor_pre_v = inputs->o2_sensor_pre_v;
    emission_state.o2_sensor_post_v = inputs->o2_sensor_post_v;

    // Execute control functions
    manage_air_fuel_ratio(inputs->engine_load, inputs->engine_rpm);
    monitor_catalyst();
    control_egr(inputs->engine_load, inputs->coolant_temp_c);
    control_evap(inputs->engine_load, inputs->coolant_temp_c);
}

/**
 * @brief Initialize emission control module
 */
void emission_control_init(void) {
    emission_state.lambda_actual = 1.0f;
    emission_state.fuel_trim_short = 0.0f;
    emission_state.fuel_trim_long = 0.0f;
    emission_state.catalyst_temp_c = 0.0f;
    emission_state.o2_sensor_pre_v = 0.0f;
    emission_state.o2_sensor_post_v = 0.0f;
    emission_state.egr_rate_pct = 0.0f;
    emission_state.evap_purge_duty = 0.0f;
    emission_state.catalyst_efficient = false;
}
```

File: fault_detection.c

```
#include "dsm_config.h"

typedef struct {
    float crank_pos_deg;       // Crankshaft position (degrees)
    float throttle_pos_pct;    // Throttle position (0-100%)
    float motor_current_a;     // Motor current (amps)
    uint8_t fault_detected;    // Fault flag
} FaultDetection_t;

// Initialize fault detection
void FaultDetection_Init(FaultDetection_t* fd) {
    fd->crank_pos_deg = 0.0f;
    fd->throttle_pos_pct = 0.0f;
    fd->motor_current_a = 0.0f;
    fd->fault_detected = FAULT_NONE;
}

// Sensor diagnostics (plausibility check)
FaultCode_t SensorDiagnostics(FaultDetection_t* fd, float crank_pos, float throttle_pos)
{
    // Check crankshaft position plausibility
    if (crank_pos < 0.0f || crank_pos > 360.0f) {
        fd->fault_detected = FAULT_SENSOR;
        return FAULT_SENSOR;
    }
    fd->crank_pos_deg = crank_pos;

    // Check throttle position plausibility
    if (throttle_pos < 0.0f || throttle_pos > 100.0f) {
        fd->fault_detected = FAULT_SENSOR;
        return FAULT_SENSOR;
    }
    fd->throttle_pos_pct = throttle_pos;

    // Cross-check throttle vs. crank (simplified correlation)
    if (throttle_pos > 50.0f && crank_pos < 10.0f) {
        fd->fault_detected = FAULT_SENSOR;
        return FAULT_SENSOR;
    }

    fd->fault_detected = FAULT_NONE;
    return FAULT_NONE;
}

// Actuator diagnostics
FaultCode_t ActuatorDiagnostics(FaultDetection_t* fd, float expected_current, float
actual_current) {
      float current_diff = (actual_current - expected_current) / expected_current *
100.0f;
    fd->motor_current_a = actual_current;

    if (current_diff > MAX_SENSOR_DIFF || current_diff < -MAX_SENSOR_DIFF) {
```

```c
        fd->fault_detected = FAULT_ACTUATOR;
        return FAULT_ACTUATOR;
    }

    fd->fault_detected = FAULT_NONE;
    return FAULT_NONE;
}
```

File: fuel_delivery.c

```
/**
 * @file fuel_delivery.c
 * @brief Fuel Delivery Control Module for Vehicle ECU
 * @author Grok 3 (xAI)
 * @date March 09, 2025
 */

#include <stdint.h>
#include <stdbool.h>

// Constants
#define BASE_INJ_PULSE_MS       2.0f     // Base injector pulse width (ms)
#define MAX_INJ_PULSE_MS        10.0f    // Maximum pulse width
#define TARGET_FUEL_PRESSURE_KPA 300.0f  // Target fuel pressure (kPa)
#define MIN_FUEL_PRESSURE_KPA   250.0f   // Minimum acceptable pressure

// State variables
static float injector_pulse_ms = BASE_INJ_PULSE_MS;
static float fuel_pump_duty = 50.0f;  // Initial duty cycle (%)

/**
 * @brief Control fuel injector pulse width
 * @param engine_load Engine load (%)
 * @param engine_rpm Engine speed (RPM)
 * @param lambda Current air-fuel ratio
 */
static void control_injection(float engine_load, uint16_t engine_rpm, float lambda) {
    // Base pulse width adjusted by load and RPM
      injector_pulse_ms = BASE_INJ_PULSE_MS * (engine_load / 100.0f) * (engine_rpm /
1000.0f);

    // Simple lambda correction
    float lambda_error = 1.0f - lambda;  // Target lambda = 1.0
    injector_pulse_ms *= (1.0f + lambda_error * 0.5f);

    // Apply limits
    injector_pulse_ms = (injector_pulse_ms > MAX_INJ_PULSE_MS) ?
                        MAX_INJ_PULSE_MS :
                        (injector_pulse_ms < 0.0f) ? 0.0f : injector_pulse_ms;
}

/**
 * @brief Regulate fuel pump pressure
 * @param fuel_pressure_kpa Current fuel pressure (kPa)
 * @param engine_load Engine load (%)
 */
static void regulate_fuel_pump(float fuel_pressure_kpa, float engine_load) {
    float pressure_error = TARGET_FUEL_PRESSURE_KPA - fuel_pressure_kpa;

    // Simple proportional control
    fuel_pump_duty += pressure_error * 0.1f;
```

```
    // Adjust duty based on load
    fuel_pump_duty += (engine_load - 50.0f) * 0.2f;

    // Limit duty cycle
    fuel_pump_duty = (fuel_pump_duty > 100.0f) ? 100.0f :
                    (fuel_pump_duty < 20.0f) ? 20.0f : fuel_pump_duty;
}

/**
 * @brief Main fuel delivery execution function
 * @param engine_load Engine load (%)
 * @param engine_rpm Engine speed (RPM)
 * @param lambda Current air-fuel ratio
 * @param fuel_pressure_kpa Current fuel pressure (kPa)
 */
void fuel_delivery_execute(float engine_load, uint16_t engine_rpm, float lambda, float fuel_pressure_kpa) {
    control_injection(engine_load, engine_rpm, lambda);
    regulate_fuel_pump(fuel_pressure_kpa, engine_load);

    // Apply commands (assumed external functions)
    set_injector_pulse(injector_pulse_ms);
    set_fuel_pump_duty(fuel_pump_duty);
}

// External function prototypes (assumed implemented elsewhere)
extern void set_injector_pulse(float pulse_ms);
extern void set_fuel_pump_duty(float duty_pct);
```

File: ignition_control.c

```c
/**
 * @file ignition_control.c
 * @brief Ignition Control Module for Vehicle ECU
 * @author Grok 3 (xAI)
 * @date March 09, 2025
 */

#include <stdint.h>
#include <stdbool.h>

// Constants
#define BASE_TIMING_ADVANCE_DEG     10.0f    // Base spark advance (degrees BTDC)
#define MAX_TIMING_ADVANCE_DEG      40.0f    // Maximum spark advance
#define MIN_TIMING_ADVANCE_DEG      0.0f     // Minimum spark advance
#define KNOCK_THRESHOLD             2.0f     // Knock sensor amplitude threshold
#define KNOCK_RETARD_STEP_DEG       2.0f     // Timing retard per knock event

// State variables
static float current_timing_advance_deg = BASE_TIMING_ADVANCE_DEG;
static bool knock_detected = false;

/**
 * @brief Optimize spark timing based on operating conditions
 * @param engine_load Engine load (%)
 * @param engine_rpm Engine speed (RPM)
 */
static void optimize_spark_timing(float engine_load, uint16_t engine_rpm) {
    // Simple timing map: advance increases with RPM, decreases with load
    float rpm_factor = (float)engine_rpm / 6000.0f;  // Normalized to max RPM
    float load_factor = engine_load / 100.0f;

    current_timing_advance_deg = BASE_TIMING_ADVANCE_DEG +
                                (20.0f * rpm_factor) -
                                (10.0f * load_factor);

    // Apply limits
    current_timing_advance_deg = (current_timing_advance_deg > MAX_TIMING_ADVANCE_DEG) ?
                                MAX_TIMING_ADVANCE_DEG :
                                (current_timing_advance_deg < MIN_TIMING_ADVANCE_DEG) ?
                                MIN_TIMING_ADVANCE_DEG : current_timing_advance_deg;
}

/**
 * @brief Detect and mitigate engine knock
 * @param knock_sensor_value Knock sensor amplitude
 */
static void detect_knock(float knock_sensor_value) {
    knock_detected = (knock_sensor_value > KNOCK_THRESHOLD);

    if (knock_detected) {
        // Retard timing when knock is detected
        current_timing_advance_deg -= KNOCK_RETARD_STEP_DEG;
```

```
            current_timing_advance_deg = (current_timing_advance_deg < MIN_TIMING_ADVANCE_DEG) ?
                                    MIN_TIMING_ADVANCE_DEG : current_timing_advance_deg;
    }
}

/**
 * @brief Main ignition control execution function
 * @param engine_load Engine load (%)
 * @param engine_rpm Engine speed (RPM)
 * @param knock_sensor_value Knock sensor amplitude
 */
void ignition_control_execute(float engine_load, uint16_t engine_rpm, float knock_sensor_value) {
    optimize_spark_timing(engine_load, engine_rpm);
    detect_knock(knock_sensor_value);

    // Apply timing command (assumed external function)
    set_spark_timing(current_timing_advance_deg);
}

// External function prototypes (assumed implemented elsewhere)
extern void set_spark_timing(float timing_deg);
```

File: inverter_management.c

```c
#include <stdint.h>
#include "motor_config.h"

#define PWM_FREQ_HZ 10000    // PWM frequency in Hz
#define MAX_TEMP_C 150       // Maximum IGBT temperature in Celsius

typedef struct {
    uint16_t pwm_duty_cycle; // PWM duty cycle (0-1000 for 0-100%)
    float igbt_temp_c;       // IGBT temperature
    uint8_t thermal_derate;  // Derating flag
} InverterManagement_t;

// Initialize inverter management
void InverterManagement_Init(InverterManagement_t* im) {
    im->pwm_duty_cycle = 0;
    im->igbt_temp_c = 25.0f;
    im->thermal_derate = 0;
}

// Generate PWM signals for inverter
void PWM_Generation(InverterManagement_t* im, float torque_cmd) {
    // Simplified: Map torque command to duty cycle
    im->pwm_duty_cycle = (uint16_t)((torque_cmd / MAX_TORQUE_NM) * 1000);
    if (im->pwm_duty_cycle > 1000) im->pwm_duty_cycle = 1000;
}

// Monitor and protect against thermal overload
void ThermalProtection(InverterManagement_t* im) {
    if (im->igbt_temp_c > MAX_TEMP_C) {
        im->thermal_derate = 1;
        im->pwm_duty_cycle = im->pwm_duty_cycle * 0.8f; // Derate by 20%
    } else if (im->igbt_temp_c < (MAX_TEMP_C - 20)) {
        im->thermal_derate = 0;
    }
}
```

File: limp_home_mode.c

```c
#include "dsm_config.h"

typedef struct {
    OperationMode_t mode;     // Current operating mode
    float max_power_pct;      // Max allowed power in limp mode (0-1)
    float backup_rpm;         // Backup RPM from model
} LimpHomeMode_t;

// Initialize limp-home mode
void LimpHomeMode_Init(LimpHomeMode_t* lhm) {
    lhm->mode = MODE_NORMAL;
    lhm->max_power_pct = 1.0f;
    lhm->backup_rpm = 0.0f;
}

// Switch to degraded operation based on fault
void DegradedOperationLogic(LimpHomeMode_t* lhm, FaultCode_t fault) {
    switch (fault) {
        case FAULT_SENSOR:
            lhm->mode = MODE_LIMP;
            lhm->max_power_pct = 0.5f; // 50% power limit
            break;
        case FAULT_ACTUATOR:
            lhm->mode = MODE_LIMP;
            lhm->max_power_pct = 0.3f; // 30% power limit
            break;
        case FAULT_TORQUE:
        case FAULT_HV:
            lhm->mode = MODE_SHUTDOWN;
            lhm->max_power_pct = 0.0f; // Full shutdown
            break;
        default:
            lhm->mode = MODE_NORMAL;
            lhm->max_power_pct = 1.0f;
            break;
    }
}

// Backup control using redundant signals/models
float BackupControl(LimpHomeMode_t* lhm, float primary_rpm, uint8_t sensor_ok) {
    if (!sensor_ok) {
        // Simplified model: Estimate RPM based on throttle and time
        lhm->backup_rpm = lhm->backup_rpm + 10.0f; // Dummy increment
        if (lhm->backup_rpm > 3000) lhm->backup_rpm = 3000; // Cap backup RPM
        return lhm->backup_rpm;
    }
    lhm->backup_rpm = primary_rpm;
    return primary_rpm;
}
```

File: mode_selection.c

```c
/**
 * @file mode_selection.c
 * @brief Mode Selection Control Module for Hybrid Vehicle ECU
 * @author Grok 3 (xAI)
 * @date March 09, 2025
 */

#include <stdint.h>
#include <stdbool.h>

// Constants
#define SOC_EV_THRESHOLD        30.0f    // Minimum SOC for EV mode (%)
#define SOC_HYBRID_THRESHOLD    20.0f    // Minimum SOC for hybrid mode (%)
#define DEMAND_THRESHOLD_HIGH   70.0f    // High demand threshold (%)
#define CLUTCH_ENGAGE_TIME_MS   500      // Clutch engagement time (ms)

// Operating modes
typedef enum {
    MODE_EV,
    MODE_HYBRID,
    MODE_CHARGE_SUSTAINING
} drive_mode_t;

// State variables
static drive_mode_t current_mode = MODE_EV;
static bool clutch_engaged = false;

/**
 * @brief Determine optimal drive mode
 * @param soc Battery state of charge (%)
 * @param driver_demand Driver torque demand (%)
 * @param route_urban Upcoming urban driving flag
 */
static void select_drive_mode(float soc, float driver_demand, bool route_urban) {
     if (soc > SOC_EV_THRESHOLD && driver_demand < DEMAND_THRESHOLD_HIGH && route_urban)
{
        current_mode = MODE_EV;
    } else if (soc > SOC_HYBRID_THRESHOLD) {
        current_mode = MODE_HYBRID;
    } else {
        current_mode = MODE_CHARGE_SUSTAINING;
    }
}

/**
 * @brief Manage smooth transition between modes
 * @param new_mode Target drive mode
 */
static void control_transition(drive_mode_t new_mode) {
    if (new_mode != current_mode) {
        if (new_mode == MODE_EV) {
            clutch_engaged = false;
```

```c
            ramp_down_ice_torque(CLUTCH_ENGAGE_TIME_MS);
        } else {
            clutch_engaged = true;
            ramp_up_ice_torque(CLUTCH_ENGAGE_TIME_MS);
        }
        current_mode = new_mode;
    }
}

/**
 * @brief Main mode selection execution function
 * @param soc Battery state of charge (%)
 * @param driver_demand Driver torque demand (%)
 * @param route_urban Upcoming urban driving flag
 */
void mode_selection_execute(float soc, float driver_demand, bool route_urban) {
    drive_mode_t target_mode;

    select_drive_mode(soc, driver_demand, route_urban);
    control_transition(target_mode);

    set_drive_mode(current_mode);
    set_clutch_state(clutch_engaged);
}

// External function prototypes (assumed implemented elsewhere)
extern void set_drive_mode(drive_mode_t mode);
extern void set_clutch_state(bool state);
extern void ramp_up_ice_torque(uint32_t time_ms);
extern void ramp_down_ice_torque(uint32_t time_ms);
```

File: motor_torque_control.c

```c
#include <stdint.h>
#include "motor_config.h"

#define MAX_TORQUE_NM 300    // Maximum torque in Newton-meters
#define MIN_REGEN_TORQUE -150 // Minimum regenerative torque

typedef struct {
    float battery_soc;       // State of Charge (0-1)
    uint8_t hybrid_mode;     // 0: Electric, 1: Hybrid, 2: Regen
    float current_torque;    // Current torque in Nm
    float stator_current_d;  // Direct-axis current (FOC)
    float stator_current_q;  // Quadrature-axis current (FOC)
} TorqueControl_t;

// Initialize torque control parameters
void TorqueControl_Init(TorqueControl_t* tc) {
    tc->battery_soc = 1.0f;
    tc->hybrid_mode = 0;
    tc->current_torque = 0.0f;
    tc->stator_current_d = 0.0f;
    tc->stator_current_q = 0.0f;
}

// Calculate required motor torque
float MotorTorqueDemand(TorqueControl_t* tc, float driver_demand) {
    float torque_cmd = 0.0f;

    switch(tc->hybrid_mode) {
        case 0: // Electric mode
            torque_cmd = driver_demand * MAX_TORQUE_NM * tc->battery_soc;
            break;
        case 1: // Hybrid mode
            torque_cmd = driver_demand * (MAX_TORQUE_NM * 0.7f);
            break;
        case 2: // Regenerative braking
            torque_cmd = MIN_REGEN_TORQUE * driver_demand;
            break;
    }
    tc->current_torque = torque_cmd;
    return torque_cmd;
}

// Field-Oriented Control implementation
void FOC_Control(TorqueControl_t* tc, float torque_ref) {
    // Simplified FOC: Calculate d-q currents based on torque reference
    tc->stator_current_q = torque_ref / MOTOR_KT; // Torque constant
    tc->stator_current_d = 0.0f; // Assuming no flux weakening here
}

// Regenerative braking torque management
float RegenerativeBraking(TorqueControl_t* tc, float brake_pedal) {
    if (tc->hybrid_mode == 2 && tc->battery_soc < 0.9f) {
```

```
        float regen_torque = MIN_REGEN_TORQUE * brake_pedal;
        tc->current_torque = regen_torque;
        return regen_torque;
    }
    return 0.0f;
}
```

File: obd_compliance.c

```
#include "dsm_config.h"

#define MAX_DTC_COUNT 10 // Maximum stored DTCs

typedef struct {
    uint8_t mil_status;      // Malfunction Indicator Lamp (0: Off, 1: On)
    DTC_t dtc_list[MAX_DTC_COUNT]; // Stored DTCs
    uint8_t dtc_count;       // Number of active DTCs
} OBDCompliance_t;

// Initialize OBD compliance
void OBDCompliance_Init(OBDCompliance_t* obd) {
    obd->mil_status = 0;
    obd->dtc_count = 0;
    for (int i = 0; i < MAX_DTC_COUNT; i++) {
        obd->dtc_list[i].code = 0;
        obd->dtc_list[i].status = 0;
        obd->dtc_list[i].timestamp = 0;
    }
}

// Activate MIL for emissions-related faults
void MIL_Activation(OBDCompliance_t* obd, FaultCode_t fault) {
    if (fault == FAULT_SENSOR || fault == FAULT_ACTUATOR) { // Emissions-related
        obd->mil_status = 1;
    } else if (fault == FAULT_NONE) {
        obd->mil_status = 0; // Reset if no fault
    }
}

// Store Diagnostic Trouble Code
void DTC_Storage(OBDCompliance_t* obd, FaultCode_t fault, uint32_t timestamp) {
    if (obd->dtc_count < MAX_DTC_COUNT && fault != FAULT_NONE) {
          obd->dtc_list[obd->dtc_count].code = (uint16_t)fault + 0xP1000; // Example DTC
format (P1000+)
        obd->dtc_list[obd->dtc_count].status = 1;
        obd->dtc_list[obd->dtc_count].timestamp = timestamp;
        obd->dtc_count++;
    }
}

// Retrieve DTCs (for service)
void DTC_Retrieve(OBDCompliance_t* obd, DTC_t* output, uint8_t* count) {
    *count = obd->dtc_count;
    for (int i = 0; i < obd->dtc_count; i++) {
        output[i] = obd->dtc_list[i];
    }
}
```

File: predictive_energy_management.c

```c
/**
 * @file predictive_energy_management.c
 * @brief Predictive Energy Management Module for Hybrid Vehicle ECU
 * @author Grok 3 (xAI)
 * @date March 09, 2025
 */

#include <stdint.h>
#include <stdbool.h>

// Constants
#define URBAN_SOC_TARGET        50.0f    // Target SOC for urban areas (%)
#define BATTERY_PRECON_TEMP_C   25.0f    // Optimal battery temperature (°C)
#define ICE_PRECON_TEMP_C       70.0f    // Optimal ICE temperature (°C)

// State variables
static bool precon_active = false;

/**
 * @brief Optimize energy usage based on route data
 * @param soc Current state of charge (%)
 * @param route_urban Upcoming urban driving flag
 * @param route_distance Remaining route distance (km)
 */
static void optimize_route_energy(float soc, bool route_urban, float route_distance) {
    if (route_urban && soc < URBAN_SOC_TARGET && route_distance > 5.0f) {
        // Charge battery for urban zone
        request_charge_mode();
    } else if (!route_urban && soc > 20.0f) {
        // Use more battery on highway
        request_discharge_mode();
    }
}

/**
 * @brief Pre-condition battery and ICE for upcoming demand
 * @param battery_temp_c Current battery temperature (°C)
 * @param coolant_temp_c Current coolant temperature (°C)
 * @param high_demand_ahead Upcoming high demand flag
 */
static void precondition_thermal(float battery_temp_c, float coolant_temp_c, bool
high_demand_ahead) {
    if (high_demand_ahead) {
        precon_active = true;
        if (battery_temp_c < BATTERY_PRECON_TEMP_C) {
            activate_battery_heater();
        }
        if (coolant_temp_c < ICE_PRECON_TEMP_C) {
            request_ice_warmup();
        }
    } else {
        precon_active = false;
```

```
        deactivate_battery_heater();
    }
}

/**
 * @brief Main predictive energy management execution function
 * @param soc Current state of charge (%)
 * @param route_urban Upcoming urban driving flag
 * @param route_distance Remaining route distance (km)
 * @param battery_temp_c Current battery temperature (°C)
 * @param coolant_temp_c Current coolant temperature (°C)
 * @param high_demand_ahead Upcoming high demand flag
 */
void predictive_energy_execute(float soc, bool route_urban, float route_distance,
                                        float battery_temp_c, float coolant_temp_c, bool
high_demand_ahead) {
    optimize_route_energy(soc, route_urban, route_distance);
    precondition_thermal(battery_temp_c, coolant_temp_c, high_demand_ahead);
}

// External function prototypes (assumed implemented elsewhere)
extern void request_charge_mode(void);
extern void request_discharge_mode(void);
extern void activate_battery_heater(void);
extern void deactivate_battery_heater(void);
extern void request_ice_warmup(void);
```

File: predictive_maintenance.c

```
#include "dsm_config.h"

#define TEMP_THRESHOLD_C 120.0f    // Warning temperature threshold
#define CURRENT_THRESHOLD_A 200.0f // Warning current threshold
#define MAX_TREND_POINTS 100       // Number of data points for trend analysis

typedef struct {
    float motor_temp_c[MAX_TREND_POINTS];  // Motor temperature history
    float motor_current_a[MAX_TREND_POINTS]; // Motor current history
    uint16_t trend_idx;                    // Current index in trend array
    uint8_t maintenance_flag;              // 0: No action, 1: Schedule maintenance
} PredictiveMaint_t;

// Initialize predictive maintenance
void PredictiveMaint_Init(PredictiveMaint_t* pm) {
    pm->trend_idx = 0;
    pm->maintenance_flag = 0;
    for (int i = 0; i < MAX_TREND_POINTS; i++) {
        pm->motor_temp_c[i] = 0.0f;
        pm->motor_current_a[i] = 0.0f;
    }
}

// Record sensor data for trend analysis
void PredictiveMaint_Record(PredictiveMaint_t* pm, float temp, float current) {
    if (pm->trend_idx < MAX_TREND_POINTS) {
        pm->motor_temp_c[pm->trend_idx] = temp;
        pm->motor_current_a[pm->trend_idx] = current;
        pm->trend_idx++;
    } else {
        // Shift data left and add new point (circular buffer)
        for (int i = 0; i < MAX_TREND_POINTS - 1; i++) {
            pm->motor_temp_c[i] = pm->motor_temp_c[i + 1];
            pm->motor_current_a[i] = pm->motor_current_a[i + 1];
        }
        pm->motor_temp_c[MAX_TREND_POINTS - 1] = temp;
        pm->motor_current_a[MAX_TREND_POINTS - 1] = current;
    }
}

// Analyze trends and predict maintenance needs
void PredictiveMaint_Analyze(PredictiveMaint_t* pm) {
    float temp_avg = 0.0f, current_avg = 0.0f;
    float temp_slope = 0.0f; // Simplified trend slope

    // Calculate averages
    for (int i = 0; i < pm->trend_idx; i++) {
        temp_avg += pm->motor_temp_c[i];
        current_avg += pm->motor_current_a[i];
    }
    temp_avg /= pm->trend_idx;
    current_avg /= pm->trend_idx;
```

```
    // Simplified slope calculation (last 10 points)
    if (pm->trend_idx >= 10) {
        float temp_start = pm->motor_temp_c[pm->trend_idx - 10];
        float temp_end = pm->motor_temp_c[pm->trend_idx - 1];
        temp_slope = (temp_end - temp_start) / 10.0f; // °C per sample
    }

    // Maintenance prediction logic
    if (temp_avg > TEMP_THRESHOLD_C || current_avg > CURRENT_THRESHOLD_A || temp_slope >
0.5f) {
        pm->maintenance_flag = 1; // Schedule maintenance
    } else {
        pm->maintenance_flag = 0;
    }
}

// Get maintenance status
uint8_t PredictiveMaint_GetStatus(PredictiveMaint_t* pm) {
    return pm->maintenance_flag;
}
```

File: predictive_torque.c

```
#include <stdint.h>
#include "motor_config.h"

#define MAX_INCLINE_DEG 15.0f  // Max road incline considered
#define ECO_FACTOR 0.7f        // Torque reduction factor for efficiency

typedef struct {
    float vehicle_speed_kph;  // Vehicle speed (km/h)
    float road_incline_deg;   // Road incline (degrees)
    float driver_accel_pct;   // Accelerator pedal position (0-100%)
    float predicted_torque;   // Predicted torque demand (Nm)
    uint8_t hybrid_mode;      // 0: Electric, 1: Hybrid
} PredictiveTorque_t;

// Initialize predictive torque allocation
void PredictiveTorque_Init(PredictiveTorque_t* pt) {
    pt->vehicle_speed_kph = 0.0f;
    pt->road_incline_deg = 0.0f;
    pt->driver_accel_pct = 0.0f;
    pt->predicted_torque = 0.0f;
    pt->hybrid_mode = 0;
}

// Predict torque based on driving conditions
float PredictiveTorqueAllocation(PredictiveTorque_t* pt, float speed, float incline,
float accel) {
    pt->vehicle_speed_kph = speed;
    pt->road_incline_deg = incline;
    pt->driver_accel_pct = accel;

    // Base torque from driver input
    float base_torque = accel / 100.0f * MAX_TORQUE_NM;

    // Adjust for incline (more torque uphill, less downhill)
    float incline_factor = 1.0f + (incline / MAX_INCLINE_DEG) * 0.3f;
    if (incline_factor < 0.7f) incline_factor = 0.7f; // Minimum limit

    // Adjust for speed (reduce torque at high speed for efficiency)
    float speed_factor = (speed > 80.0f) ? ECO_FACTOR : 1.0f;

    // Final predicted torque
    pt->predicted_torque = base_torque * incline_factor * speed_factor;

    // Mode switch logic
    if (pt->predicted_torque > MAX_TORQUE_NM * 0.8f || speed > 120.0f) {
        pt->hybrid_mode = 1; // Switch to hybrid mode
    } else {
        pt->hybrid_mode = 0; // Stay in electric mode
    }

    if (pt->predicted_torque > MAX_TORQUE_NM) pt->predicted_torque = MAX_TORQUE_NM;
    return pt->predicted_torque;
```

```c
}

// Update torque allocation periodically
void PredictiveTorque_Update(PredictiveTorque_t* pt) {
    // Placeholder for real-time sensor updates
    float new_speed = pt->vehicle_speed_kph; // From CAN bus
    float new_incline = pt->road_incline_deg; // From IMU
    float new_accel = pt->driver_accel_pct;   // From pedal sensor
    PredictiveTorqueAllocation(pt, new_speed, new_incline, new_accel);
}
```

File: safety_functions.c

```c
#include "dsm_config.h"

typedef struct {
    float requested_torque_nm; // Requested torque
    float delivered_torque_nm; // Actual delivered torque
    float hv_voltage_v;        // High-voltage system voltage
    uint8_t interlock_status;  // 0: Open, 1: Closed
} SafetyFunctions_t;

// Initialize safety functions
void SafetyFunctions_Init(SafetyFunctions_t* sf) {
    sf->requested_torque_nm = 0.0f;
    sf->delivered_torque_nm = 0.0f;
    sf->hv_voltage_v = MAX_VOLTAGE_V;
    sf->interlock_status = 1; // Closed by default
}

// Torque monitoring (ASIL-D requirement)
FaultCode_t TorqueMonitoring(SafetyFunctions_t* sf, float requested, float delivered) {
    sf->requested_torque_nm = requested;
    sf->delivered_torque_nm = delivered;

    float torque_diff = (delivered - requested) / requested * 100.0f;
    if (torque_diff > 5.0f || torque_diff < -5.0f) { // 5% tolerance
        return FAULT_TORQUE;
    }
    return FAULT_NONE;
}

// High-voltage interlock management
FaultCode_t HighVoltageInterlock(SafetyFunctions_t* sf, float voltage) {
    sf->hv_voltage_v = voltage;

    if (voltage < MIN_VOLTAGE_V || voltage > MAX_VOLTAGE_V) {
        sf->interlock_status = 0; // Open interlock
        return FAULT_HV;
    }

    sf->interlock_status = 1; // Closed interlock
    return FAULT_NONE;
}
```

File: speed_control.c

```c
#include <stdint.h>
#include "motor_config.h"

#define MAX_RPM 12000    // Maximum allowable RPM
#define MIN_RPM 0        // Minimum RPM

typedef struct {
    uint16_t current_rpm;    // Current motor speed
    uint16_t target_rpm;     // Desired motor speed
    uint8_t over_speed_flag; // Over-speed condition
} SpeedControl_t;

// Initialize speed control
void SpeedControl_Init(SpeedControl_t* sc) {
    sc->current_rpm = 0;
    sc->target_rpm = 0;
    sc->over_speed_flag = 0;
}

// Regulate motor RPM
void RPM_Regulation(SpeedControl_t* sc, uint16_t drivetrain_rpm) {
    sc->target_rpm = drivetrain_rpm;
    if (sc->current_rpm < sc->target_rpm) {
        // Increase torque demand (interface with torque control)
    } else if (sc->current_rpm > sc->target_rpm) {
        // Decrease torque demand
    }
}

// Over-speed protection
void OverSpeedProtection(SpeedControl_t* sc) {
    if (sc->current_rpm > MAX_RPM) {
        sc->over_speed_flag = 1;
        // Trigger torque reduction or shutdown
    } else {
        sc->over_speed_flag = 0;
    }
}
```

File: thermal_management.c

```c
/**
 * @file thermal_management.c
 * @brief Thermal Management Control Module for Vehicle ECU
 * @author Grok 3 (xAI)
 * @date March 09, 2025
 */

#include <stdint.h>
#include <stdbool.h>

// Constants
#define FAN_ON_TEMP_C           95.0f    // Temperature to turn on fan (°C)
#define FAN_OFF_TEMP_C          85.0f    // Temperature to turn off fan (°C)
#define TARGET_WARMUP_TEMP_C    70.0f    // Target warm-up temperature (°C)
#define MAX_COOLANT_TEMP_C      110.0f   // Maximum allowable temperature (°C)

// State variables
static bool fan_active = false;
static bool thermostat_open = false;

/**
 * @brief Control coolant temperature via fan and thermostat
 * @param coolant_temp_c Current coolant temperature (°C)
 */
static void control_coolant_temp(float coolant_temp_c) {
    // Fan control with hysteresis
    if (coolant_temp_c > FAN_ON_TEMP_C) {
        fan_active = true;
    } else if (coolant_temp_c < FAN_OFF_TEMP_C) {
        fan_active = false;
    }

    // Thermostat control
    thermostat_open = (coolant_temp_c > TARGET_WARMUP_TEMP_C);

    // Overheat protection
    if (coolant_temp_c > MAX_COOLANT_TEMP_C) {
        set_diagnostic_code(DTC_OVERHEAT);
    }
}

/**
 * @brief Manage engine warm-up strategy
 * @param coolant_temp_c Current coolant temperature (°C)
 * @param engine_load Engine load (%)
 */
static void manage_warmup(float coolant_temp_c, float engine_load) {
    if (coolant_temp_c < TARGET_WARMUP_TEMP_C) {
        // Request increased idle speed during warm-up (assumed external function)
        request_idle_speed(1000);  // Higher idle for faster warm-up

        // Limit load during warm-up
```

```c
        if (engine_load > 50.0f) {
            limit_engine_load(50.0f);
        }
    } else {
        request_idle_speed(700);  // Normal idle
    }
}

/**
 * @brief Main thermal management execution function
 * @param coolant_temp_c Current coolant temperature (°C)
 * @param engine_load Engine load (%)
 */
void thermal_management_execute(float coolant_temp_c, float engine_load) {
    control_coolant_temp(coolant_temp_c);
    manage_warmup(coolant_temp_c, engine_load);

    // Apply commands (assumed external functions)
    set_fan_state(fan_active);
    set_thermostat_state(thermostat_open);
}

// External function prototypes (assumed implemented elsewhere)
extern void set_fan_state(bool state);
extern void set_thermostat_state(bool state);
extern void request_idle_speed(uint16_t rpm);
extern void limit_engine_load(float max_load_pct);
extern void set_diagnostic_code(uint16_t dtc);
```

File: torque_control.c

```c
/**
 * @file torque_control.c
 * @brief Torque Control Module for Hybrid Vehicle Powertrain
 * @author Grok 3 (xAI)
 * @date March 09, 2025
 */

#include <stdint.h>
#include <stdbool.h>
#include "torque_control.h"
#include "vehicle_config.h"
#include "sensor_interface.h"

// Constants
#define MAX_ICE_TORQUE_NM       300.0f    // Maximum ICE torque capability
#define MAX_MOTOR_TORQUE_NM     200.0f    // Maximum electric motor torque
#define MIN_IDLE_RPM            700       // Target idle speed
#define MAX_THERMAL_LIMIT_C     120.0f    // Maximum engine temperature

// Structure to hold torque control state
typedef struct {
    float ice_torque_nm;          // Current ICE torque command
    float motor_torque_nm;        // Current motor torque command
    float driver_demand_pct;      // Accelerator pedal position (0-100%)
    uint16_t engine_rpm;          // Current engine speed
    float engine_temp_c;          // Engine temperature
    operating_mode_t mode;        // Current operating mode
} torque_control_t;

// Private variables
static torque_control_t torque_state = {0};

/**
 * @brief Calculate demanded torque based on driver input and hybrid strategy
 * @param pedal_position Accelerator pedal position (0-100%)
 * @param vehicle_speed Current vehicle speed (km/h)
 * @return float Total demanded torque (Nm)
 */
static float calculate_torque_demand(float pedal_position, float vehicle_speed) {
    float demand_torque = 0.0f;

    // Simple linear mapping of pedal position to torque
    torque_state.driver_demand_pct = pedal_position;
        demand_torque = (pedal_position / 100.0f) * (MAX_ICE_TORQUE_NM +
MAX_MOTOR_TORQUE_NM);

    // Adjust based on hybrid strategy and vehicle speed
    switch (torque_state.mode) {
        case MODE_ELECTRIC_ONLY:
            demand_torque = (demand_torque > MAX_MOTOR_TORQUE_NM) ?
                           MAX_MOTOR_TORQUE_NM : demand_torque;
            break;
```

```
        case MODE_HYBRID:
            // Blend between ICE and motor based on speed
            demand_torque *= (vehicle_speed < 50.0f) ? 0.7f : 1.0f;
            break;
        case MODE_ICE_ONLY:
            demand_torque = (demand_torque > MAX_ICE_TORQUE_NM) ?
                           MAX_ICE_TORQUE_NM : demand_torque;
            break;
    }

    return demand_torque;
}

/**
 * @brief Arbitrate torque between ICE and electric motor
 * @param demand_torque Total requested torque (Nm)
 */
static void arbitrate_torque(float demand_torque) {
    switch (torque_state.mode) {
        case MODE_ELECTRIC_ONLY:
            torque_state.motor_torque_nm = demand_torque;
            torque_state.ice_torque_nm = 0.0f;
            break;

        case MODE_HYBRID:
            // Simple 60/40 split between motor and ICE in hybrid mode
            torque_state.motor_torque_nm = demand_torque * 0.6f;
            torque_state.ice_torque_nm = demand_torque * 0.4f;

            // Cap at individual max limits
                        torque_state.motor_torque_nm  =  (torque_state.motor_torque_nm  >
MAX_MOTOR_TORQUE_NM) ?
                                                                   MAX_MOTOR_TORQUE_NM  :
torque_state.motor_torque_nm;
                           torque_state.ice_torque_nm  =  (torque_state.ice_torque_nm  >
MAX_ICE_TORQUE_NM) ?
                                       MAX_ICE_TORQUE_NM : torque_state.ice_torque_nm;
            break;

        case MODE_ICE_ONLY:
            torque_state.motor_torque_nm = 0.0f;
            torque_state.ice_torque_nm = demand_torque;
            break;
    }
}

/**
 * @brief Maintain stable idle speed when ICE is running
 */
static void control_idle_speed(void) {
    if (torque_state.mode != MODE_ELECTRIC_ONLY &&
        torque_state.engine_rpm < MIN_IDLE_RPM) {
        // Increase ICE torque to maintain idle
        torque_state.ice_torque_nm += ((float)(MIN_IDLE_RPM - torque_state.engine_rpm) *
```

```
0.1f);
        torque_state.ice_torque_nm = (torque_state.ice_torque_nm > 50.0f) ?
                                    50.0f : torque_state.ice_torque_nm;
    }
}

/**
 * @brief Apply torque limits based on mechanical and thermal constraints
 */
static void limit_torque(void) {
    // Thermal limit
    if (torque_state.engine_temp_c > MAX_THERMAL_LIMIT_C) {
        torque_state.ice_torque_nm *= 0.8f;  // Derate by 20%
    }

    // Mechanical limits
    torque_state.ice_torque_nm = (torque_state.ice_torque_nm > MAX_ICE_TORQUE_NM) ?
                                MAX_ICE_TORQUE_NM : torque_state.ice_torque_nm;
     torque_state.motor_torque_nm = (torque_state.motor_torque_nm > MAX_MOTOR_TORQUE_NM)
?
                                 MAX_MOTOR_TORQUE_NM : torque_state.motor_torque_nm;

    // Ensure non-negative torque
    torque_state.ice_torque_nm = (torque_state.ice_torque_nm < 0.0f) ?
                                0.0f : torque_state.ice_torque_nm;
    torque_state.motor_torque_nm = (torque_state.motor_torque_nm < 0.0f) ?
                                  0.0f : torque_state.motor_torque_nm;
}

/**
 * @brief Main torque control execution function
 * @param inputs Pointer to torque control inputs structure
 */
void torque_control_execute(torque_inputs_t* inputs) {
    float demand_torque;

    // Update state from sensors
    torque_state.engine_rpm = inputs->engine_rpm;
    torque_state.engine_temp_c = inputs->engine_temp_c;
    torque_state.mode = inputs->operating_mode;

    // Calculate torque demand
    demand_torque = calculate_torque_demand(inputs->pedal_position,
                                          inputs->vehicle_speed);

    // Arbitrate between power sources
    arbitrate_torque(demand_torque);

    // Maintain idle speed if needed
    control_idle_speed();

    // Apply safety limits
    limit_torque();
```

```
    // Send torque commands to actuators
    set_ice_torque_command(torque_state.ice_torque_nm);
    set_motor_torque_command(torque_state.motor_torque_nm);
}

/**
 * @brief Initialize torque control module
 */
void torque_control_init(void) {
    torque_state.ice_torque_nm = 0.0f;
    torque_state.motor_torque_nm = 0.0f;
    torque_state.driver_demand_pct = 0.0f;
    torque_state.engine_rpm = 0;
    torque_state.engine_temp_c = 0.0f;
    torque_state.mode = MODE_ELECTRIC_ONLY;
}
```

File: traction_stability_control.c

```c
/**
 * @file traction_stability_control.c
 * @brief Traction and Stability Control Module for Hybrid Vehicle ECU
 * @author Grok 3 (xAI)
 * @date March 09, 2025
 */

#include <stdint.h>
#include <bool.h>

// Constants
#define SLIP_THRESHOLD_PCT      10.0f    // Wheel slip threshold for traction loss (%)
#define MAX_ICE_TORQUE_NM       300.0f   // Maximum ICE torque
#define MAX_MOTOR_TORQUE_NM     200.0f   // Maximum motor torque
#define ECO_MODE_TORQUE_SCALE   0.7f     // Torque scaling factor in Eco mode
#define REGEN_BRAKE_MAX_NM      150.0f   // Maximum regenerative braking torque

// Operating modes (assumed from mode_selection.c)
typedef enum {
    MODE_EV,
    MODE_HYBRID,
    MODE_CHARGE_SUSTAINING
} drive_mode_t;

// State variables
static float ice_torque_cmd_nm = 0.0f;
static float motor_torque_cmd_nm = 0.0f;
static float regen_brake_torque_nm = 0.0f;
static bool traction_active = false;
static bool esp_intervention = false;

/**
 * @brief Detect wheel slip using wheel speed sensors
 * @param wheel_speeds Array of wheel speeds (km/h) [FL, FR, RL, RR]
 * @param vehicle_speed Vehicle speed (km/h)
 * @return bool True if slip detected
 */
static bool detect_slip(float wheel_speeds[4], float vehicle_speed) {
    float max_slip_pct = 0.0f;

    for (int i = 0; i < 4; i++) {
        float slip_pct = (wheel_speeds[i] - vehicle_speed) * 100.0f /
                        (vehicle_speed > 0.1f ? vehicle_speed : 0.1f);
        if (slip_pct > max_slip_pct) {
            max_slip_pct = slip_pct;
        }
    }

    return (max_slip_pct > SLIP_THRESHOLD_PCT);
}

/**
```

```c
 * @brief Reduce torque to regain traction
 * @param mode Current drive mode
 * @param slip_detected Slip detection status
 */
static void reduce_torque(drive_mode_t mode, bool slip_detected) {
    if (slip_detected) {
        traction_active = true;
        if (mode == MODE_EV) {
            motor_torque_cmd_nm *= 0.8f;  // Reduce motor torque by 20%
        } else {
            ice_torque_cmd_nm *= 0.8f;    // Reduce ICE torque by 20%
            motor_torque_cmd_nm *= 0.8f;
        }
    } else {
        traction_active = false;
    }

    // Apply limits
    ice_torque_cmd_nm = (ice_torque_cmd_nm > MAX_ICE_TORQUE_NM) ? MAX_ICE_TORQUE_NM :
                       (ice_torque_cmd_nm < 0.0f) ? 0.0f : ice_torque_cmd_nm;
            motor_torque_cmd_nm   =   (motor_torque_cmd_nm   >   MAX_MOTOR_TORQUE_NM)   ?
MAX_MOTOR_TORQUE_NM :
                         (motor_torque_cmd_nm < 0.0f) ? 0.0f : motor_torque_cmd_nm;
}

/**
 * @brief Integrate with Electronic Stability Program (ESP)
 * @param esp_torque_reduction Requested torque reduction from ESP (Nm)
 */
static void integrate_esp(float esp_torque_reduction) {
    if (esp_torque_reduction > 0.0f) {
        esp_intervention = true;
        ice_torque_cmd_nm -= esp_torque_reduction * 0.5f;
        motor_torque_cmd_nm -= esp_torque_reduction * 0.5f;

        ice_torque_cmd_nm = (ice_torque_cmd_nm < 0.0f) ? 0.0f : ice_torque_cmd_nm;
        motor_torque_cmd_nm = (motor_torque_cmd_nm < 0.0f) ? 0.0f : motor_torque_cmd_nm;
    } else {
        esp_intervention = false;
    }
}

/**
 * @brief Blend regenerative and friction braking
 * @param brake_demand Total brake demand (Nm)
 * @param soc Battery state of charge (%)
 */
static void blend_brakes(float brake_demand, float soc) {
    if (soc < 90.0f) {  // Allow regen if battery not full
        regen_brake_torque_nm = brake_demand * 0.6f;  // 60% regen
        regen_brake_torque_nm = (regen_brake_torque_nm > REGEN_BRAKE_MAX_NM) ?
                               REGEN_BRAKE_MAX_NM : regen_brake_torque_nm;
    } else {
        regen_brake_torque_nm = 0.0f;  // No regen if battery full
```

```c
    }

    float friction_brake_torque_nm = brake_demand - regen_brake_torque_nm;
    friction_brake_torque_nm = (friction_brake_torque_nm < 0.0f) ? 0.0f : friction_brake_torque_nm;

    set_regen_brake(regen_brake_torque_nm);
    set_friction_brake(friction_brake_torque_nm);
}

/**
 * @brief Support Adaptive Cruise Control (ACC)
 * @param acc_torque_request Torque request from ACC (Nm)
 */
static void support_acc(float acc_torque_request) {
    if (acc_torque_request > 0.0f) {
        motor_torque_cmd_nm = acc_torque_request;  // Prefer motor for ACC
        ice_torque_cmd_nm = (acc_torque_request > MAX_MOTOR_TORQUE_NM) ?
                            (acc_torque_request - MAX_MOTOR_TORQUE_NM) : 0.0f;
    } else if (acc_torque_request < 0.0f) {
        blend_brakes(-acc_torque_request, soc);  // Negative torque = braking
    }
}

/**
 * @brief Adjust power delivery for Eco mode
 * @param driver_demand Driver torque demand (%)
 * @param eco_mode Eco mode active flag
 * @param mode Current drive mode
 */
static void eco_mode_logic(float driver_demand, bool eco_mode, drive_mode_t mode) {
    float total_torque = driver_demand * (MAX_ICE_TORQUE_NM + MAX_MOTOR_TORQUE_NM) / 100.0f;

    if (eco_mode) {
        total_torque *= ECO_MODE_TORQUE_SCALE;  // Softer throttle response

        if (mode == MODE_HYBRID) {
            motor_torque_cmd_nm = total_torque * 0.7f;  // Favor motor for efficiency
            ice_torque_cmd_nm = total_torque * 0.3f;
        } else if (mode == MODE_EV) {
            motor_torque_cmd_nm = total_torque;
            ice_torque_cmd_nm = 0.0f;
        } else {
            ice_torque_cmd_nm = total_torque;
            motor_torque_cmd_nm = 0.0f;
        }
    } else {
        motor_torque_cmd_nm = total_torque * 0.5f;  // Default split
        ice_torque_cmd_nm = total_torque * 0.5f;
    }
}

/**
```

```
 * @brief Main traction and stability control execution function
 * @param wheel_speeds Array of wheel speeds (km/h) [FL, FR, RL, RR]
 * @param vehicle_speed Vehicle speed (km/h)
 * @param driver_demand Driver torque demand (%)
 * @param mode Current drive mode
 * @param brake_demand Total brake demand (Nm)
 * @param soc Battery state of charge (%)
 * @param esp_torque_reduction ESP torque reduction request (Nm)
 * @param acc_torque_request ACC torque request (Nm)
 * @param eco_mode Eco mode active flag
 */
void traction_stability_execute(float wheel_speeds[4], float vehicle_speed, float driver_demand,
                               drive_mode_t mode, float brake_demand, float soc,
                                   float esp_torque_reduction, float acc_torque_request,
bool eco_mode) {
    // Traction Control
    bool slip_detected = detect_slip(wheel_speeds, vehicle_speed);
    eco_mode_logic(driver_demand, eco_mode, mode);  // Set initial torque commands
    reduce_torque(mode, slip_detected);

    // Stability Control Interface
    integrate_esp(esp_torque_reduction);
    blend_brakes(brake_demand, soc);

    // Driver Assistance Integration
    support_acc(acc_torque_request);

    // Apply torque commands
    set_ice_torque(ice_torque_cmd_nm);
    set_motor_torque(motor_torque_cmd_nm);
}

// External function prototypes (assumed implemented elsewhere)
extern void set_ice_torque(float torque_nm);
extern void set_motor_torque(float torque_nm);
extern void set_regen_brake(float torque_nm);
extern void set_friction_brake(float torque_nm);
```

File: transmission_control.c

```
/**
 * @file transmission_control.c
 * @brief Transmission Control Module (TCM) for Hybrid Vehicle ECU
 * @author Grok 3 (xAI)
 * @date March 09, 2025
 */

#include <stdint.h>
#include <stdbool.h>

// Constants
#define MIN_GEAR_RATIO          0.5f     // Minimum effective gear ratio (e-CVT)
#define MAX_GEAR_RATIO          4.0f     // Maximum effective gear ratio
#define CLUTCH_ENGAGE_TIME_MS   300      // Clutch engagement time (ms)
#define CREEP_TORQUE_NM         10.0f    // Creep torque for EV mode
#define OPTIMAL_ICE_RPM         2000     // Peak efficiency RPM for ICE
#define OPTIMAL_MOTOR_RPM       3000     // Peak efficiency RPM for motor

// Operating modes (assumed from mode_selection.c)
typedef enum {
    MODE_EV,
    MODE_HYBRID,
    MODE_CHARGE_SUSTAINING
} drive_mode_t;

// State variables
static float current_gear_ratio = 1.0f;
static bool clutch_engaged = false;
static float launch_torque_nm = 0.0f;

/**
 * @brief Select optimal gear ratio (e-CVT style)
 * @param vehicle_speed Vehicle speed (km/h)
 * @param driver_demand Driver torque demand (%)
 * @param mode Current drive mode
 */
static void select_gear_ratio(float vehicle_speed, float driver_demand, drive_mode_t
mode) {
    if (mode == MODE_EV) {
        // Favor higher ratio (lower RPM) for motor efficiency
        current_gear_ratio = (vehicle_speed > 0.0f) ?
                                          (OPTIMAL_MOTOR_RPM * 0.06f / vehicle_speed) :
MAX_GEAR_RATIO;
    } else {
        // Adjust ratio based on demand and ICE efficiency
        current_gear_ratio = (driver_demand > 50.0f) ?
                             MIN_GEAR_RATIO + (MAX_GEAR_RATIO - MIN_GEAR_RATIO) * (1.0f -
driver_demand / 100.0f) :
                            MAX_GEAR_RATIO;
    }

    // Apply limits
```

```c
    current_gear_ratio = (current_gear_ratio > MAX_GEAR_RATIO) ? MAX_GEAR_RATIO :
                           (current_gear_ratio < MIN_GEAR_RATIO) ? MIN_GEAR_RATIO :
current_gear_ratio;
}

/**
 * @brief Control clutch engagement/disengagement
 * @param mode Current drive mode
 */
static void control_clutch(drive_mode_t mode) {
    if (mode == MODE_EV && clutch_engaged) {
        clutch_engaged = false;
        ramp_clutch_disengage(CLUTCH_ENGAGE_TIME_MS);
    } else if (mode != MODE_EV && !clutch_engaged) {
        clutch_engaged = true;
        ramp_clutch_engage(CLUTCH_ENGAGE_TIME_MS);
    }
}

/**
 * @brief Simulate creep behavior in EV mode
 * @param vehicle_speed Vehicle speed (km/h)
 * @param brake_pressed Brake pedal state
 */
static void manage_creep(float vehicle_speed, bool brake_pressed) {
    if (vehicle_speed < 5.0f && !brake_pressed) {
        launch_torque_nm = CREEP_TORQUE_NM;
    } else {
        launch_torque_nm = 0.0f;
    }
}

/**
 * @brief Coordinate smooth vehicle launch
 * @param driver_demand Driver torque demand (%)
 * @param mode Current drive mode
 */
static void smooth_start(float driver_demand, drive_mode_t mode) {
    float target_torque = driver_demand * (MAX_ICE_TORQUE_NM + MAX_MOTOR_TORQUE_NM) /
100.0f;

    if (mode == MODE_EV) {
        launch_torque_nm = target_torque;  // Motor handles launch
    } else if (mode == MODE_HYBRID) {
        // Blend ICE and motor torque
        launch_torque_nm = target_torque * 0.6f;  // Motor provides initial boost
        set_ice_torque(target_torque * 0.4f);
    } else {
        launch_torque_nm = 0.0f;  // ICE handles launch
        set_ice_torque(target_torque);
    }
}

/**
```

```c
 * @brief Shift load point for efficiency
 * @param ice_rpm Current ICE RPM
 * @param motor_rpm Current motor RPM
 * @param mode Current drive mode
 */
static void shift_load_point(uint16_t ice_rpm, uint16_t motor_rpm, drive_mode_t mode) {
    if (mode == MODE_HYBRID || mode == MODE_CHARGE_SUSTAINING) {
        if (ice_rpm < OPTIMAL_ICE_RPM - 200) {
            current_gear_ratio += 0.1f;  // Lower RPM, increase ratio
        } else if (ice_rpm > OPTIMAL_ICE_RPM + 200) {
            current_gear_ratio -= 0.1f;  // Higher RPM, decrease ratio
        }
    } else if (mode == MODE_EV) {
        if (motor_rpm < OPTIMAL_MOTOR_RPM - 300) {
            current_gear_ratio += 0.1f;
        } else if (motor_rpm > OPTIMAL_MOTOR_RPM + 300) {
            current_gear_ratio -= 0.1f;
        }
    }

    // Apply limits
    current_gear_ratio = (current_gear_ratio > MAX_GEAR_RATIO) ? MAX_GEAR_RATIO :
                                (current_gear_ratio < MIN_GEAR_RATIO) ? MIN_GEAR_RATIO :
current_gear_ratio;
}

/**
 * @brief Main transmission control execution function
 * @param vehicle_speed Vehicle speed (km/h)
 * @param driver_demand Driver torque demand (%)
 * @param mode Current drive mode
 * @param brake_pressed Brake pedal state
 * @param ice_rpm Current ICE RPM
 * @param motor_rpm Current motor RPM
 */
void transmission_control_execute(float vehicle_speed, float driver_demand, drive_mode_t
mode,
                                          bool brake_pressed, uint16_t ice_rpm, uint16_t
motor_rpm) {
    // Shift Control
    select_gear_ratio(vehicle_speed, driver_demand, mode);
    control_clutch(mode);

    // Launch Control
    manage_creep(vehicle_speed, brake_pressed);
    smooth_start(driver_demand, mode);

    // Efficiency Optimization
    shift_load_point(ice_rpm, motor_rpm, mode);

    // Apply commands
    set_gear_ratio(current_gear_ratio);
    set_clutch_state(clutch_engaged);
    set_motor_torque(launch_torque_nm);
```

```
}

// External function prototypes (assumed implemented elsewhere)
extern void set_gear_ratio(float ratio);
extern void set_clutch_state(bool state);
extern void ramp_clutch_engage(uint32_t time_ms);
extern void ramp_clutch_disengage(uint32_t time_ms);
extern void set_motor_torque(float torque_nm);
extern void set_ice_torque(float torque_nm);

// Assumed constants from other modules
#define MAX_ICE_TORQUE_NM       300.0f
#define MAX_MOTOR_TORQUE_NM     200.0f
```

File: update_management.c

```c
/**
 * @file update_management.c
 * @brief Update Management Subcomponent for OTA Update and Cloud Integration Module
 * @author Grok 3 (xAI)
 * @date March 09, 2025
 */

#include <stdint.h>
#include <stdbool.h>

// Constants
#define FIRMWARE_VERSION_CURRENT 0x0100  // Example current version (1.0)
#define CHECKSUM_SIZE            4       // Size of checksum in bytes
#define ROLLBACK_TIMEOUT_MS      30000   // Timeout for rollback check (30s)

// State variables
static uint16_t current_version = FIRMWARE_VERSION_CURRENT;
static bool update_in_progress = false;

/**
 * @brief Validate firmware update integrity and compatibility
 * @param firmware_data Pointer to firmware data
 * @param size Firmware size in bytes
 * @param version Firmware version
 * @param checksum Expected checksum
 * @return bool True if valid
 */
static bool validate_firmware(const uint8_t* firmware_data, uint32_t size, uint16_t
version, uint32_t checksum) {
    // Simple checksum calculation (example: sum of bytes)
    uint32_t calculated_checksum = 0;
    for (uint32_t i = 0; i < size - CHECKSUM_SIZE; i++) {
        calculated_checksum += firmware_data[i];
    }

    // Check compatibility (example: major version must match)
    bool version_compatible = (version & 0xFF00) == (FIRMWARE_VERSION_CURRENT & 0xFF00);

    return (calculated_checksum == checksum) && version_compatible && (version >
current_version);
}

/**
 * @brief Handle rollback if update fails
 * @param update_successful Flag indicating update success
 */
static void manage_rollback(bool update_successful) {
    if (update_in_progress && !update_successful) {
        // Wait for system stability check
        if (get_system_uptime_ms() > ROLLBACK_TIMEOUT_MS) {
            revert_to_previous_firmware();
            set_diagnostic_code(DTC_OTA_UPDATE_FAILED);
```

```
            update_in_progress = false;
        }
    } else if (update_successful) {
        update_in_progress = false;
    }
}

/**
 * @brief Main update management execution function
 * @param firmware_data Pointer to firmware data
 * @param size Firmware size in bytes
 * @param version Firmware version
 * @param checksum Expected checksum
 */
void update_management_execute(const uint8_t* firmware_data, uint32_t size, uint16_t
version, uint32_t checksum) {
         if (!update_in_progress && validate_firmware(firmware_data, size, version,
checksum)) {
        update_in_progress = true;
        apply_firmware_update(firmware_data, size);
        current_version = version;
    }

    // Check if update was successful (assumed external confirmation)
    bool update_successful = check_update_status();
    manage_rollback(update_successful);
}

// External function prototypes (assumed implemented elsewhere)
extern void apply_firmware_update(const uint8_t* data, uint32_t size);
extern bool check_update_status(void);
extern void revert_to_previous_firmware(void);
extern uint32_t get_system_uptime_ms(void);
extern void set_diagnostic_code(uint16_t dtc);
```

File: user_customization.c

```c
/**
 * @file user_customization.c
 * @brief User Customization Subcomponent for OTA Update and Cloud Integration Module
 * @author Grok 3 (xAI)
 * @date March 09, 2025
 */

#include <stdint.h>
#include <stdbool.h>

// Operating modes (assumed from mode_selection.c)
typedef enum {
    MODE_EV,
    MODE_HYBRID,
    MODE_CHARGE_SUSTAINING
} drive_mode_t;

// Drive profile types
typedef enum {
    PROFILE_ECO,
    PROFILE_SPORT,
    PROFILE_DEFAULT
} drive_profile_t;

// State variables
static drive_profile_t current_profile = PROFILE_DEFAULT;

/**
 * @brief Sync drive mode profiles from cloud
 * @param cloud_profile Profile data from cloud
 */
static void sync_drive_profiles(drive_profile_t cloud_profile) {
    if (cloud_profile != current_profile) {
        current_profile = cloud_profile;
        switch (cloud_profile) {
            case PROFILE_ECO:
                set_throttle_response(0.7f);  // Softer response
                set_default_mode(MODE_HYBRID);
                break;
            case PROFILE_SPORT:
                set_throttle_response(1.2f);  // Sharper response
                set_default_mode(MODE_HYBRID);
                break;
            case PROFILE_DEFAULT:
                set_throttle_response(1.0f);  // Normal response
                set_default_mode(MODE_HYBRID);
                break;
        }
    }
}

/**
```

```c
 * @brief Process remote commands from smartphone app
 * @param command_id Command identifier from app
 * @param soc Battery state of charge (%)
 */
static void process_remote_commands(uint8_t command_id, float soc) {
    switch (command_id) {
        case REMOTE_CMD_PRECONDITION:
            activate_preconditioning();
            break;
        case REMOTE_CMD_MODE_EV:
            if (soc > 30.0f) set_requested_mode(MODE_EV);
            break;
        case REMOTE_CMD_MODE_HYBRID:
            set_requested_mode(MODE_HYBRID);
            break;
        case REMOTE_CMD_MODE_CHARGE:
            set_requested_mode(MODE_CHARGE_SUSTAINING);
            break;
        default:
            break;
    }
}

/**
 * @brief Main user customization execution function
 * @param cloud_profile Profile data from cloud
 * @param command_id Command identifier from app
 * @param soc Battery state of charge (%)
 */
void user_customization_execute(drive_profile_t cloud_profile, uint8_t command_id, float
soc) {
    sync_drive_profiles(cloud_profile);
    process_remote_commands(command_id, soc);
}

// External function prototypes (assumed implemented elsewhere)
extern void set_throttle_response(float scale_factor);
extern void set_default_mode(drive_mode_t mode);
extern void set_requested_mode(drive_mode_t mode);
extern void activate_preconditioning(void);

// Remote command IDs (example values)
#define REMOTE_CMD_PRECONDITION  1
#define REMOTE_CMD_MODE_EV       2
#define REMOTE_CMD_MODE_HYBRID   3
#define REMOTE_CMD_MODE_CHARGE   4
```