Porting Kalman Filter Simulation to nRF52832 for Maxon Motor Driver

September 16, 2025

Contents

1 Feasibility

Porting the `kalmanPWM.m` MATLAB/Octave simulation to the nRF52832 for the Maxon Motor Driver hardware, using the provided C files and the nRF5 SDK v15.2.0, is feasible but requires significant effort due to differences in environment, hardware constraints, and real-time requirements. This document outlines the feasibility, challenges, and steps to achieve this port, focusing on the provided code and hardware context from the Maxon Motor Driver repository.

The `kalmanPWM.m` script implements a Kalman filter for a three-phase PWM-controlled motor, modeling rotor position, angular velocity, and q-axis current, using Hall sensor feedback and a discretized state-space model. The nRF52832, a Cortex-M4F microcontroller, has sufficient computational power for real-time Kalman filtering, PWM generation, and sensor interfacing (via MPU9250 and Hall sensors). The provided C files (`GPIO.c`, `GPIO.h`, `main.c`, `mpu9250.c`, `mpu9250.h`) include PWM control and Hall sensor handling, aligning with the simulation's requirements. However, porting requires translating MATLAB/Octave code into C, adapting to nRF52832 peripherals, and integrating with the nRF5 SDK.

2 Key Components and Alignment

2.1 Hardware Compatibility

- **Maxon Motor**: The repository specifies an 8mm diameter 3-phase DC motor. The simulation parameters (e.g., $R = 5.5\,\Omega$, $L = 0.0002\,\mathrm{H}$, $K_e = 0.002387\,\mathrm{V/(rad/s)}$, $K_t = 0.0024\,\mathrm{Nm/A}$, $J = 3 \times 10^{-7}\,\mathrm{kg\ m^2}$, $B = 2 \times 10^{-5}$) are likely tailored to a similar motor, suggesting compatibility.
- **nRF52832**: Supports PWM generation (via `nrf_pwm`), GPIO for Hall sensors, and SPI for MPU9250 communication, as seen in `GPIO.c` and `mpu9250.c`.
- **Sensors**: The simulation uses Hall sensors for rotor position and current measurements. `GPIO.c` includes Hall sensor inputs (`hs_1`, `hs_2`, `hs_3`) and a phase handler, while `mpu9250.c` provides gyro and accelerometer data, which could enhance the Kalman filter with sensor fusion (though not used in the current simulation).

2.2 Firmware Foundation

- `GPIO.c`: Initializes PWM (1 MHz, 100-count top value) and handles phase commutation based on Hall sensor inputs, matching the simulation's three-phase PWM control.
- `mpu9250.c`: Interfaces with the MPU9250 for 9-axis sensor data, which could be used for advanced sensor fusion but is not required by `kalmanPWM.m`.
- `main.c`: Sets up a timer interrupt, which can be extended for periodic Kalman filter updates.
- `nRF5 SDK`: Provides drivers for PWM, GPIOTE, SPI, and timers, supporting the required peripherals.

2.3 Kalman Filter Requirements

- The Kalman filter uses a 3×3 state-space model (position, velocity, current) with matrix operations. The nRF52832's Cortex-M4F with FPU can handle these floating-point computations efficiently.
- The simulation's time step ($\Delta t = 0.001\,\mathrm{s}$) and PWM frequency (1 kHz) are realistic for the nRF52832's timers and PWM module.

3 Challenges

1. **MATLAB to C Translation**:
   - MATLAB's matrix operations (e.g., $A \cdot x$, $\text{inv}(H \cdot P_{\text{pred}} \cdot H' + R)$) must be implemented in C, using a lightweight matrix library or custom functions.
   - The `control` package's `ss` and `c2d` functions are not available in C. Discretized matrices (A, B) must be precomputed or hardcoded.
2. **Real-Time Constraints**:
   - The nRF52832 must perform Kalman filter updates, PWM generation, and Hall sensor processing in real time (1 ms loop). The Cortex-M4F at 64 MHz can handle this, but optimization is needed.
   - Memory constraints (64 KB SRAM, 512 KB Flash) require efficient data structures and minimal dynamic allocation.
3. **Sensor Integration**:
   - The simulation assumes Hall sensors provide rotor position with noise. The `phase_handler` in `GPIO.c` maps to discrete phases (1–6), not continuous angles, requiring additional processing.
   - Current measurement (`i_meas`) requires an ADC or external current sensor, not implemented in the provided code.
4. **PWM Implementation**:
   - The simulation generates PWM signals with a 1 kHz carrier and sinusoidal duty cycles. `GPIO.c` uses a fixed duty cycle (`0x8040`), requiring modification for dynamic PWM control.
5. **MPU9250 Integration**:
   - The simulation does not use MPU9250 data, but the hardware includes it. Integrating gyro data could improve velocity estimation but requires extending the Kalman filter model.

4 Steps to Port `kalmanPWM.m` to nRF52832

1. **Precompute System Matrices**:
   - Use MATLAB/Octave to compute discretized matrices (A, B) using `c2d` with $\Delta t = 0.001$. Hardcode in C:

```c
float A[3][3] = {{...}, {...}, {...}}; // From sys_disc.A
float B[3][1] = {{...}, {...}, {...}}; // From sys_disc.B
float H[2][3] = {{1, 0, 0}, {0, 0, 1}}; // Measurement matrix
float Q[3][3] = {{1e-4, 0, 0}, {0, 1e-2, 0}, {0, 0, 1e-3}}; //
    Process noise
float R[2][2] = {{0.01, 0}, {0, 0.0004}}; // Measurement noise
```

2. **Implement Matrix Operations**:
   - Use CMSIS-DSP or custom functions for 3×3 matrix operations. Example for matrix multiplication:

```c
void matrix_multiply(float A[3][3], float B[3][1], float result
    [3][1]) {
     for (int i = 0; i < 3; i++) {
         result[i][0] = 0;
         for (int j = 0; j < 3; j++) {
             result[i][0] += A[i][j] * B[j][0];
         }
     }
}
```

3. **Kalman Filter Algorithm**:

   - Implement the Kalman filter loop:

```c
float x_hat[3][1] = {{0}, {0}, {0}}; // State estimate
float P[3][3] = {{1, 0, 0}, {0, 1, 0}, {0, 0, 1}}; //
    Covariance
void kalman_filter(float V_q, float z_k[2][1]) {
    float x_pred[3][1], P_pred[3][3], K[3][2], temp[2][3],
        temp2[2][2];
    // Prediction
    matrix_multiply(A, x_hat, x_pred);
    matrix_add(x_pred, matrix_multiply_scalar(B, V_q), x_pred);
    matrix_multiply(A, matrix_multiply(P, A_transpose), P_pred)
        ;
    matrix_add(P_pred, Q, P_pred);
    // Update
    matrix_multiply(H, P_pred, temp);
    matrix_multiply(temp, H_transpose, temp2);
    matrix_add(temp2, R, temp2);
    matrix_inverse(temp2, K);
    matrix_multiply(P_pred, H_transpose, K);
    matrix_multiply_scalar(K, 1.0 / matrix_determinant(temp2),
        K);
    matrix_subtract(z_k, matrix_multiply(H, x_pred), temp2);
    matrix_multiply(K, temp2, x_hat);
    matrix_add(x_pred, x_hat, x_hat);
    matrix_multiply(eye(3), matrix_multiply(K, H), temp);
    matrix_subtract(eye(3), temp, temp);
    matrix_multiply(temp, P_pred, P);
}
```

4. **Modify PWM Control**:

   - Update pwm_init and increase_phase in GPIO.c for sinusoidal PWM duty cycles:

```c
float t = 0;
void timer_pwm_handler(nrf_timer_event_t event_type, void*
    p_context) {
     if (event_type == NRF_TIMER_EVENT_COMPARE0) {
         t += 0.001; // Delta_t
         float sine_scale = 5.0 / (24.0 * 0.5);
         float sine_a = 0.5 * (1 + sine_scale * sin(2 * M_PI *
            10 * t + M_PI/2));
         float sine_b = 0.5 * (1 + sine_scale * sin(2 * M_PI *
            10 * t - 2 * M_PI / 3 + M_PI/2));
         float sine_c = 0.5 * (1 + sine_scale * sin(2 * M_PI *
            10 * t + 2 * M_PI / 3 + M_PI/2));
```

```
        seq_values[0].channel_0 = (uint16_t)(sine_a * 100) | 0x8000;
        seq_values[0].channel_1 = (uint16_t)(sine_b * 100) | 0x8000;
        seq_values[0].channel_2 = (uint16_t)(sine_c * 100) | 0x8000;
        nrf_pwm_values_t new_pwm_values = { .p_individual = seq_values };
        nrf_drv_pwm_sequence_values_update(&m_pwm0, 1, new_pwm_values);
    }
}
```

5. **Hall Sensor Processing**:

   - Modify `phase_handler` to estimate continuous rotor position:

```
float get_theta_from_phase(int32_t motorPhase) {
    switch (motorPhase) {
        case 5: return 0.0; // Phase 1-2
        case 4: return 2 * M_PI / 3; // Phase 1-3
        case 6: return 4 * M_PI / 3; // Phase 2-3
        case 2: return M_PI; // Phase 2-1
        case 3: return 5 * M_PI / 3; // Phase 3-1
        case 1: return M_PI / 3; // Phase 3-2
        default: return 0.0;
    }
}
```

6. **Current Measurement**:

   - Add ADC support (`nrf_drv_saadc`) for q-axis current:

```
float get_V_q(float V_a, float V_b, float V_c, float theta_e) {
    return (2.0/3.0) * (V_a * cos(theta_e) + V_b * cos(theta_e - 2 * M_PI / 3) + V_c * cos(theta_e + 2 * M_PI / 3));
}
```

7. **Integrate with Main Loop**:

   - Modify `main.c` for periodic Kalman filter updates:

```
void timer_kalman_handler(nrf_timer_event_t event_type, void* p_context) {
    if (event_type == NRF_TIMER_EVENT_COMPARE0) {
        float theta_true = get_theta_from_phase(motorPhase);
        float i_meas = read_adc_current(); // Implement ADC reading
        float z_k[2][1] = {{theta_true}, {i_meas}};
        float V_q = get_V_q(u[0], u[1], u[2], theta_e);
        kalman_filter(V_q, z_k);
    }
}
```

8. **Testing and Optimization**:

   - Test incrementally: PWM, Hall sensors, current measurements, Kalman filter.

    - Optimize for memory and performance, avoiding dynamic allocation.
    - Validate against simulation RMSE values (e.g., $\text{RMSE}_\theta = 0.080\,\text{rad}$, $\text{RMSE}_\omega = 2.824\,\text{rad/s}$, $\text{RMSE}_i = 0.016\,\text{A}$).

5 Additional Considerations

- **MPU9250 Integration**: Extend the Kalman filter to include gyro data for velocity estimation, updating the state-space model and H matrix.
- **Debugging**: Use `nrf_log` or SEGGER RTT for real-time debugging. Output estimated states (`x_est`) and compare with simulation results.
- **Power and Safety**: Ensure the hardware supports the 24V supply. Follow the repository's safety note about short circuits and heat generation.

6 Conclusion

Porting `kalmanPWM.m` to the nRF52832 is feasible using the provided C files and nRF5 SDK v15.2.0. The main tasks involve translating the Kalman filter to C, adapting PWM and Hall sensor handling, and adding current measurement. The hardware and firmware provide a solid foundation, but challenges include real-time constraints, continuous angle estimation, and matrix operations. By following the outlined steps, the system can achieve functionality comparable to the MATLAB simulation, with potential enhancements using MPU9250 data. For pricing or support, contact Electronic Realization L.L.C. at cy@elec-real.com, as noted in the repository.