

# e-foc — Design Booklet

Theory, Design, and Architecture

Gabriel Francisco dos Santos

[gabriel.fra.santos@gmail.com](mailto:gabriel.fra.santos@gmail.com)

October 05, 2026

# Contents

|                                            |          |
|--------------------------------------------|----------|
| <b>Part I — Theory</b>                     | <b>1</b> |
| Overview . . . . .                         | 2        |
| Prerequisites . . . . .                    | 2        |
| Mathematical Foundation . . . . .          | 3        |
| Block Diagrams . . . . .                   | 7        |
| Numerical Properties . . . . .             | 7        |
| Worked Example . . . . .                   | 8        |
| Controller Slots . . . . .                 | 10       |
| Limitations & Assumptions . . . . .        | 10       |
| References . . . . .                       | 11       |
| Overview . . . . .                         | 11       |
| Prerequisites . . . . .                    | 11       |
| Mathematical Foundation . . . . .          | 12       |
| Block Diagrams . . . . .                   | 14       |
| Numerical Properties . . . . .             | 16       |
| Reference Motors . . . . .                 | 16       |
| References . . . . .                       | 17       |
| Overview . . . . .                         | 18       |
| Algorithm Map . . . . .                    | 18       |
| Mathematical Foundation . . . . .          | 18       |
| Numerical Properties . . . . .             | 18       |
| Algorithm Index . . . . .                  | 20       |
| Parameter Sources . . . . .                | 21       |
| References . . . . .                       | 21       |
| Overview . . . . .                         | 22       |
| Prerequisites . . . . .                    | 22       |
| Mathematical Foundation . . . . .          | 22       |
| Output Voltage Limit . . . . .             | 24       |
| Numerical Properties . . . . .             | 24       |
| Limitations & Assumptions . . . . .        | 25       |
| References . . . . .                       | 25       |
| Overview . . . . .                         | 25       |
| Prerequisites . . . . .                    | 26       |
| Mathematical Foundation . . . . .          | 26       |
| A1 — Decoupled PID + Feedforward . . . . . | 26       |

|                                                                  |    |
|------------------------------------------------------------------|----|
| Output Voltage Limit . . . . .                                   | 27 |
| Numerical Properties . . . . .                                   | 28 |
| Limitations & Assumptions . . . . .                              | 28 |
| References . . . . .                                             | 28 |
| Overview . . . . .                                               | 29 |
| Prerequisites . . . . .                                          | 29 |
| Mathematical Foundation . . . . .                                | 29 |
| A2 — Deadbeat Current Control . . . . .                          | 30 |
| Output Voltage Limit . . . . .                                   | 31 |
| Numerical Properties . . . . .                                   | 31 |
| Limitations & Assumptions . . . . .                              | 31 |
| References . . . . .                                             | 31 |
| Overview . . . . .                                               | 32 |
| Prerequisites . . . . .                                          | 32 |
| Mathematical Foundation . . . . .                                | 32 |
| A3 — Sliding-Mode Current Control . . . . .                      | 33 |
| Output Voltage Limit . . . . .                                   | 34 |
| Numerical Properties . . . . .                                   | 34 |
| Limitations & Assumptions . . . . .                              | 34 |
| References . . . . .                                             | 35 |
| Overview . . . . .                                               | 35 |
| Prerequisites . . . . .                                          | 35 |
| Mathematical Foundation . . . . .                                | 36 |
| Numerical Properties . . . . .                                   | 38 |
| Limitations & Assumptions . . . . .                              | 38 |
| References . . . . .                                             | 38 |
| Overview . . . . .                                               | 39 |
| Prerequisites . . . . .                                          | 39 |
| Mathematical Foundation . . . . .                                | 39 |
| S1 — LQI Speed Control . . . . .                                 | 39 |
| Numerical Properties . . . . .                                   | 40 |
| Limitations & Assumptions . . . . .                              | 41 |
| References . . . . .                                             | 41 |
| Overview . . . . .                                               | 42 |
| Prerequisites . . . . .                                          | 42 |
| Mathematical Foundation . . . . .                                | 42 |
| S2 — Active Disturbance Rejection Speed Control (ADRC) . . . . . | 42 |
| Numerical Properties . . . . .                                   | 43 |
| Limitations & Assumptions . . . . .                              | 44 |
| References . . . . .                                             | 44 |
| Overview . . . . .                                               | 45 |
| Prerequisites . . . . .                                          | 45 |
| Mathematical Foundation . . . . .                                | 45 |
| S3 — Two-DOF Speed Control . . . . .                             | 45 |
| Numerical Properties . . . . .                                   | 46 |
| Limitations & Assumptions . . . . .                              | 46 |
| References . . . . .                                             | 47 |
| Overview . . . . .                                               | 47 |

|                                                 |    |
|-------------------------------------------------|----|
| Prerequisites . . . . .                         | 47 |
| Mathematical Foundation . . . . .               | 48 |
| Numerical Properties . . . . .                  | 49 |
| Limitations & Assumptions . . . . .             | 50 |
| References . . . . .                            | 50 |
| Overview . . . . .                              | 51 |
| Prerequisites . . . . .                         | 51 |
| Mathematical Foundation . . . . .               | 51 |
| P1 — LQR / LQI Position Control . . . . .       | 51 |
| Numerical Properties . . . . .                  | 53 |
| Limitations & Assumptions . . . . .             | 53 |
| References . . . . .                            | 53 |
| Overview . . . . .                              | 54 |
| Prerequisites . . . . .                         | 54 |
| Mathematical Foundation . . . . .               | 54 |
| P2 — Cascade P→PI Position Control . . . . .    | 55 |
| Numerical Properties . . . . .                  | 55 |
| Limitations & Assumptions . . . . .             | 56 |
| References . . . . .                            | 56 |
| Overview . . . . .                              | 56 |
| Prerequisites . . . . .                         | 57 |
| Mathematical Foundation . . . . .               | 57 |
| P3 — Two-DOF Position Control . . . . .         | 57 |
| Numerical Properties . . . . .                  | 58 |
| Limitations & Assumptions . . . . .             | 58 |
| References . . . . .                            | 58 |
| Overview . . . . .                              | 59 |
| Prerequisites . . . . .                         | 59 |
| Mathematical Foundation . . . . .               | 59 |
| P4 — Iterative Learning Control (ILC) . . . . . | 59 |
| Numerical Properties . . . . .                  | 60 |
| Limitations & Assumptions . . . . .             | 60 |
| References . . . . .                            | 61 |
| Overview . . . . .                              | 61 |
| Prerequisites . . . . .                         | 62 |
| Mathematical Foundation . . . . .               | 62 |
| Numerical Properties . . . . .                  | 63 |
| Limitations & Assumptions . . . . .             | 63 |
| References . . . . .                            | 63 |
| Overview . . . . .                              | 63 |
| Prerequisites . . . . .                         | 64 |
| Mathematical Foundation . . . . .               | 64 |
| Numerical Properties . . . . .                  | 68 |
| Limitations . . . . .                           | 69 |
| References . . . . .                            | 69 |
| Overview . . . . .                              | 69 |
| Prerequisites . . . . .                         | 70 |
| Mathematical Foundation . . . . .               | 71 |

|                                     |    |
|-------------------------------------|----|
| Block Diagrams . . . . .            | 74 |
| Numerical Properties . . . . .      | 74 |
| Worked Example . . . . .            | 76 |
| Limitations & Assumptions . . . . . | 77 |
| References . . . . .                | 77 |
| Overview . . . . .                  | 77 |
| Prerequisites . . . . .             | 78 |
| Mathematical Foundation . . . . .   | 78 |
| Block Diagrams . . . . .            | 80 |
| Numerical Properties . . . . .      | 80 |
| Limitations & Assumptions . . . . . | 81 |
| References . . . . .                | 81 |

## **Part II — Architecture** 84

|                                      |    |
|--------------------------------------|----|
| Assumptions & Constraints . . . . .  | 85 |
| System Overview . . . . .            | 86 |
| Component Decomposition . . . . .    | 86 |
| Interfaces & Contracts . . . . .     | 93 |
| Data Flow . . . . .                  | 94 |
| Cross-Cutting Concerns . . . . .     | 96 |
| Open Questions & Decisions . . . . . | 96 |
| Integration Testing . . . . .        | 97 |

## **Part III — Design** 100

|                                     |     |
|-------------------------------------|-----|
| Responsibilities . . . . .          | 101 |
| Component Details . . . . .         | 101 |
| Interfaces . . . . .                | 104 |
| Data Model . . . . .                | 105 |
| Block Diagram . . . . .             | 105 |
| Constraints & Limitations . . . . . | 106 |
| Responsibilities . . . . .          | 107 |
| Component Details . . . . .         | 107 |
| Interfaces . . . . .                | 109 |
| Data Model . . . . .                | 110 |
| Constraints & Limitations . . . . . | 110 |
| Controller Alternatives . . . . .   | 111 |
| Responsibilities . . . . .          | 112 |
| Component Details . . . . .         | 112 |
| Interfaces . . . . .                | 114 |
| Data Model . . . . .                | 115 |
| Sequence Diagrams . . . . .         | 116 |
| Constraints & Limitations . . . . . | 116 |
| Controller Alternatives . . . . .   | 117 |
| Responsibilities . . . . .          | 117 |
| Component Details . . . . .         | 119 |
| Interfaces . . . . .                | 122 |

|                                                       |     |
|-------------------------------------------------------|-----|
| Data Model . . . . .                                  | 123 |
| Sequence Diagrams . . . . .                           | 124 |
| Block Diagram . . . . .                               | 125 |
| Constraints & Limitations . . . . .                   | 125 |
| Responsibilities . . . . .                            | 126 |
| Component Details . . . . .                           | 126 |
| Interfaces . . . . .                                  | 137 |
| Data Model . . . . .                                  | 138 |
| State Machine . . . . .                               | 139 |
| Block Diagram . . . . .                               | 141 |
| Constraints & Limitations . . . . .                   | 141 |
| Open Questions . . . . .                              | 142 |
| Responsibilities . . . . .                            | 144 |
| Component Details . . . . .                           | 145 |
| Interfaces . . . . .                                  | 162 |
| Data Model . . . . .                                  | 165 |
| Error Handling . . . . .                              | 166 |
| Responsibilities . . . . .                            | 167 |
| Component Details . . . . .                           | 167 |
| Interfaces . . . . .                                  | 170 |
| Responsibilities . . . . .                            | 171 |
| Architecture . . . . .                                | 171 |
| Component Details . . . . .                           | 172 |
| Interfaces . . . . .                                  | 179 |
| Online Resistance and Inductance Estimation . . . . . | 179 |
| Responsibilities . . . . .                            | 181 |
| Component Details . . . . .                           | 182 |
| Interfaces . . . . .                                  | 188 |
| Online Inertia and Friction Estimation . . . . .      | 188 |
| Responsibilities . . . . .                            | 190 |
| Component Details . . . . .                           | 191 |
| Interfaces . . . . .                                  | 196 |
| Responsibilities . . . . .                            | 197 |
| Component Details . . . . .                           | 198 |
| Interfaces . . . . .                                  | 204 |
| Hardware-Test EEPROM Commands . . . . .               | 204 |
| Responsibilities . . . . .                            | 206 |
| Component Details . . . . .                           | 206 |
| Interfaces . . . . .                                  | 210 |
| Data Model . . . . .                                  | 211 |
| Sequence Diagrams . . . . .                           | 211 |
| Block Diagram . . . . .                               | 212 |
| Constraints & Limitations . . . . .                   | 212 |
| Open Questions . . . . .                              | 213 |
| Responsibilities . . . . .                            | 214 |
| Component Details . . . . .                           | 214 |
| Interfaces . . . . .                                  | 217 |
| Data Model . . . . .                                  | 217 |

---

|                                           |     |
|-------------------------------------------|-----|
| State Machine . . . . .                   | 218 |
| Sequence Diagrams . . . . .               | 218 |
| Responsibilities . . . . .                | 218 |
| Component Details . . . . .               | 221 |
| Interfaces . . . . .                      | 223 |
| Data Model . . . . .                      | 224 |
| State Machine . . . . .                   | 224 |
| Sequence Diagrams . . . . .               | 224 |
| Block Diagram . . . . .                   | 224 |
| Timing . . . . .                          | 224 |
| Constraints & Limitations . . . . .       | 227 |
| Open Questions . . . . .                  | 227 |
| Overview . . . . .                        | 228 |
| Responsibilities . . . . .                | 229 |
| Component Details . . . . .               | 229 |
| Interfaces . . . . .                      | 232 |
| Feature-to-Requirements Mapping . . . . . | 232 |

# Part I — Theory

---

| Field     | Value                        |
|-----------|------------------------------|
| Title     | Field-Oriented Control (FOC) |
| Type      | theory                       |
| Status    | approved                     |
| Version   | 1.0.0                        |
| Component | foc                          |
| Date      | 2025-01-01                   |

---

## Overview

Field-Oriented Control (FOC) — also called vector control — is the standard high-performance control strategy for Permanent Magnet Synchronous Motors (PMSM) and Brushless DC motors (BLDC). Its central insight is that a three-phase AC motor can be controlled as simply as a DC motor if all quantities are expressed in a reference frame that rotates synchronously with the rotor flux.

In the rotating (dq) reference frame the three-phase stator currents reduce to two DC components: -  $i_d$  — the flux-producing (direct) component, nominally zero for maximum torque per ampere (MTPA). -  $i_q$  — the torque-producing (quadrature) component, directly proportional to electrical torque.

Current controllers in the rotating frame regulate these DC quantities without phase lag caused by the AC coupling. The inverse transforms and Space Vector Modulation (SVM) reconstruct the three-phase PWM duty cycles that drive the inverter.

---

## Prerequisites

| Symbol       | Meaning                                             | Unit     |
|--------------|-----------------------------------------------------|----------|
| $R_s$        | Stator resistance per phase                         | $\Omega$ |
| $L_s$        | Stator inductance per phase (assuming $L_d = L_q$ ) | H        |
| $\psi_f$     | Permanent magnet flux linkage                       | Wb       |
| $p$          | Number of pole pairs                                | —        |
| $\theta_e$   | Electrical rotor angle                              | rad      |
| $\theta_m$   | Mechanical rotor angle                              | rad      |
| $\omega_e$   | Electrical angular velocity ( $p \cdot \omega_m$ )  | rad/s    |
| $V_{dc}$     | DC bus voltage                                      | V        |
| $\tau_e$     | Electrical torque                                   | N · m    |
| $1/\sqrt{3}$ | Reciprocal of $\sqrt{3}$ , $\approx 0.5774$         | —        |

---

The relationship between electrical and mechanical angles is:

$$\theta_e = p \cdot \theta_m$$


---

## Mathematical Foundation

### 1. PMSM Voltage Model (abc frame)

The PMSM stator voltage equations in the stationary abc reference frame are:

$$\begin{aligned} v_a &= R_s i_a + L_s \frac{di_a}{dt} + e_a \\ v_b &= R_s i_b + L_s \frac{di_b}{dt} + e_b \\ v_c &= R_s i_c + L_s \frac{di_c}{dt} + e_c \end{aligned}$$

where  $e_a, e_b, e_c$  are the back-EMF voltages produced by the rotating permanent magnets:

$$e_a = -\psi_f \omega_e \sin(\theta_e), \quad e_b = -\psi_f \omega_e \sin\left(\theta_e - \frac{2\pi}{3}\right), \quad e_c = -\psi_f \omega_e \sin\left(\theta_e + \frac{2\pi}{3}\right)$$

These equations are coupled through the mutual inductance and back-EMF, making direct control difficult. The Clarke and Park transforms decouple them into independently controllable quantities.

### 2. Clarke Transform — $abc \rightarrow \alpha\beta$ (Stationary Frame)

The Clarke transform projects the three-phase quantities onto a two-dimensional orthogonal stationary frame ( $\alpha$ - $\beta$ ). The **amplitude-invariant** form (factor  $\frac{2}{3}$ ) preserves the physical voltage and current magnitudes:

$$\begin{pmatrix} i_\alpha \\ i_\beta \end{pmatrix} = \frac{2}{3} \begin{pmatrix} 1 & -\frac{1}{2} & -\frac{1}{2} \\ 0 & \frac{\sqrt{3}}{2} & -\frac{\sqrt{3}}{2} \end{pmatrix} \begin{pmatrix} i_a \\ i_b \\ i_c \end{pmatrix}$$

For a balanced three-phase system ( $i_a + i_b + i_c = 0$ ), the zero-sequence component vanishes and the transform simplifies. With only two current sensors measuring  $i_a$  and  $i_b$  (and  $i_c = -i_a - i_b$ ):

$$i_\alpha = i_a, \quad i_\beta = \frac{i_a + 2i_b}{\sqrt{3}}$$

**Verification:** For a balanced current set  $i_a = I \cos \theta_e$ ,  $i_b = I \cos(\theta_e - \frac{2\pi}{3})$ :

$$i_\beta = \frac{I \cos \theta_e + 2I \left(-\frac{1}{2} \cos \theta_e + \frac{\sqrt{3}}{2} \sin \theta_e\right)}{\sqrt{3}} = \frac{I\sqrt{3} \sin \theta_e}{\sqrt{3}} = I \sin \theta_e \checkmark$$

The stator current space vector in the  $\alpha$ - $\beta$  frame thus has magnitude  $I$  and rotates at  $\omega_e$ :  $\mathbf{I}_s = i_\alpha + j i_\beta = I e^{j\theta_e}$ .

**Full three-sensor form** (when all three currents are available):

$$i_\alpha = \frac{2}{3} \left(i_a - \frac{1}{2}i_b - \frac{1}{2}i_c\right), \quad i_\beta = \frac{1}{\sqrt{3}}(i_b - i_c)$$

### 3. Park Transform — $\alpha\beta \rightarrow dq$ (Rotating Frame)

The Park transform rotates the  $\alpha\beta$  space vector by  $-\theta_e$  into the rotor-fixed dq frame:

$$\begin{pmatrix} i_d \\ i_q \end{pmatrix} = \begin{pmatrix} \cos \theta_e & \sin \theta_e \\ -\sin \theta_e & \cos \theta_e \end{pmatrix} \begin{pmatrix} i_\alpha \\ i_\beta \end{pmatrix}$$

$$i_d = i_\alpha \cos \theta_e + i_\beta \sin \theta_e$$

$$i_q = -i_\alpha \sin \theta_e + i_\beta \cos \theta_e$$

For the balanced rotating current  $\mathbf{I}_s = I e^{j\theta_e}$ :

$$i_d + j i_q = I e^{j\theta_e} \cdot e^{-j\theta_e} = I \quad \Rightarrow \quad i_d = I, \quad i_q = 0$$

This confirms: when the current vector is aligned with the d-axis (rotor flux axis),  $i_q = 0$  (zero torque). When the current vector is 90° ahead of the rotor (MTPA),  $i_d = 0$  and  $i_q = I$  (maximum torque).

**Inverse Park Transform** (dq  $\rightarrow \alpha\beta$ , for voltage reconstruction):

$$\begin{pmatrix} v_\alpha \\ v_\beta \end{pmatrix} = \begin{pmatrix} \cos \theta_e & -\sin \theta_e \\ \sin \theta_e & \cos \theta_e \end{pmatrix} \begin{pmatrix} v_d \\ v_q \end{pmatrix}$$

$$v_\alpha = v_d \cos \theta_e - v_q \sin \theta_e$$

$$v_\beta = v_d \sin \theta_e + v_q \cos \theta_e$$

### 4. PMSM Model in the dq Frame

After applying the Park transform, the PMSM voltage equations in the rotating dq frame become:

$$v_d = R_s i_d + L_s \frac{di_d}{dt} - \omega_e L_s i_q$$

$$v_q = R_s i_q + L_s \frac{di_q}{dt} + \omega_e L_s i_d + \omega_e \psi_f$$

The cross-coupling terms  $\omega_e L_s i_q$  and  $\omega_e L_s i_d$  are decoupled by feed-forward compensation. The back-EMF term  $\omega_e \psi_f$  can also be added as a feed-forward. After decoupling, both d and q channels reduce to independent first-order RL systems:

$$v'_d = R_s i_d + L_s \frac{di_d}{dt}, \quad v'_q = R_s i_q + L_s \frac{di_q}{dt}$$

These are regulated by independent current controllers, one per dq axis.

## 5. Electrical Torque

The electromagnetic torque produced by the motor is:

$$\tau_e = \frac{3}{2} p \psi_f i_q$$

Since  $\psi_f$  and  $p$  are motor constants, torque is directly proportional to  $i_q$ . Setting the  $i_q$  reference from an outer speed or position loop gives a torque-current-mode inner loop — the fundamental advantage of FOC over scalar control.

The dq-axis current regulators are interchangeable at runtime. Gain design and normalisation depend on the active algorithm — see the current loop controller chapters in [documentation/theory/](#).

## 6. Space Vector Modulation (SVM)

SVM synthesises any reference voltage vector  $\mathbf{V}_{ref} = V_\alpha + jV_\beta$  as a time-average of the eight inverter states. The three-phase half-bridge inverter produces 8 switching states:

| Vector | Switches (A,B,C) | $V_\alpha$ (normalised) | $V_\beta$ (normalised) |
|--------|------------------|-------------------------|------------------------|
| $V_0$  | (0,0,0)          | 0                       | 0                      |
| $V_1$  | (1,0,0)          | $\frac{2}{3}$           | 0                      |
| $V_2$  | (1,1,0)          | $\frac{1}{3}$           | $\frac{1}{\sqrt{3}}$   |
| $V_3$  | (0,1,0)          | $-\frac{1}{3}$          | $\frac{1}{\sqrt{3}}$   |
| $V_4$  | (0,1,1)          | $-\frac{2}{3}$          | 0                      |
| $V_5$  | (0,0,1)          | $-\frac{1}{3}$          | $-\frac{1}{\sqrt{3}}$  |
| $V_6$  | (1,0,1)          | $\frac{1}{3}$           | $-\frac{1}{\sqrt{3}}$  |
| $V_7$  | (1,1,1)          | 0                       | 0                      |

The six active vectors ( $V_1$ – $V_6$ ) form a regular hexagon of circumradius  $\frac{2}{3}$  in these normalised units — that is the length of each vertex vector in the table above. The inscribed circle (maximum linear SVM range) is the hexagon’s apothem,  $\frac{2}{3} \cos 30^\circ = \frac{1}{\sqrt{3}} \approx 0.577$ , corresponding to  $V_{dc}/\sqrt{3}$  in physical volts. The two agree: it is the circumradius that is  $\frac{2}{3}$ , and reading that as the linear range overstates it by 15%.

See: [documentation/theory/images/svm\\_hexagon.svg](#) (generated by [documentation/tools/-generate\\_plots.gp](#))

### Common-Mode Injection Method

This implementation uses the **common-mode injection** method (also called min-max injection), which avoids explicit sector detection. The three reference phase voltages are computed from  $V_\alpha, V_\beta$ :

$$\begin{aligned} v_A &= V_\alpha \\ v_B &= -\frac{V_\alpha}{2} + \frac{\sqrt{3}}{2} V_\beta \end{aligned}$$

$$v_C = -\frac{V_\alpha}{2} - \frac{\sqrt{3}}{2} V_\beta$$

A common-mode offset is added to centre the signals within the PWM range:

$$v_{common} = -\frac{\max(v_A, v_B, v_C) + \min(v_A, v_B, v_C)}{2}$$

Each phase duty cycle is then:

$$d_k = \text{clamp}\left(\frac{v_k + v_{common}}{\sqrt{3}} + 0.5, 0, 1\right), \quad k \in \{A, B, C\}$$

The factor  $1/\sqrt{3}$  in the duty cycle formula arises from the normalisation: at the hexagon boundary, the maximum phase voltage (after common-mode injection) is  $\sqrt{3}/2$  of the bus voltage, and the factor maps this to the  $[0, 1]$  duty cycle range centred at 0.5.

**Advantages over sector-based SVM:** - No angle or sector lookup required. - Identical results to classical SVM for three-phase symmetric loads. - Naturally handles over-modulation by saturation (clamping).

### SVM Linear Modulation Region

The common-mode injection guarantees full hexagon utilisation. The maximum phase voltage in the linear region corresponds to modulation index  $m = 1.0$ :

$$|V_{ref}|_{max} = \frac{V_{dc}}{\sqrt{3}} \approx 0.577 V_{dc}$$

Beyond this circle, one or more phases saturate (clamp) and the output enters over-modulation.

## 9. Trigonometric Lookup Table

The Park/inverse-Park transforms require  $\sin \theta_e$  and  $\cos \theta_e$  at the FOC control rate (20 kHz). To avoid calling `sinf/cosf` in the ISR:

- A **512-entry sine lookup table** is stored in flash, 16-byte aligned for cache efficiency.
- The table covers one full electrical revolution: `LUT[k] = sin(2π·k/512)`.
- Cosine is obtained by a 90° index offset: `cos(θ) = LUT[(k + 128) % 512]`.
- The electrical angle is mapped to a table index: `k = (uint16_t)(θ_norm * 512) & 0x1FF`.
- Angular range: angle is normalised to  $[0, 1)$  representing  $[0, 2\pi)$ .

The fixed-step table introduces a bounded angle-quantisation error of at most  $\pi/512 \approx 0.35^\circ$  when truncating to the nearest entry. The implementation interpolates linearly between adjacent entries, which bounds the amplitude error at  $\Delta^2/8 \approx 1.9 \times 10^{-5}$  instead. At 3000 rpm with  $p = 5$  the electrical angle advances  $\omega_e/20000 = 0.0785 \text{ rad} \approx 4.5^\circ$  per step, roughly 6.4 table entries, so it is the interpolation rather than the table density that bounds the error.

## Block Diagrams

### Complete FOC Loop

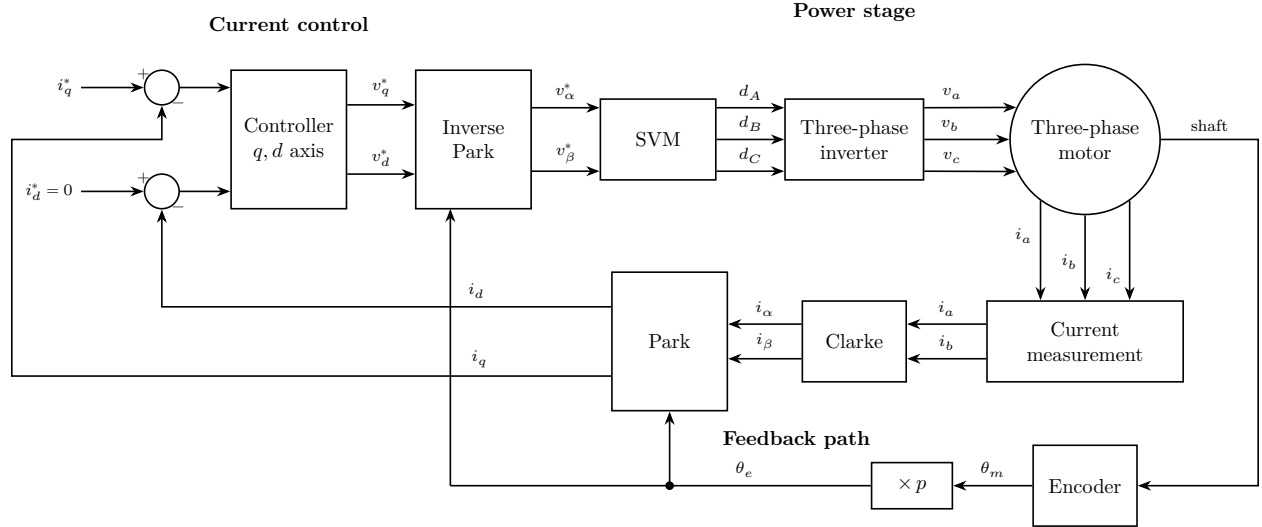


Figure 1: Classical field-oriented control (FOC) structure for a three-phase permanent magnet AC motor. The measured phase currents are transformed from the three-phase reference frame to the stationary  $\alpha\beta$  frame using the Clarke transform and subsequently to the rotating  $dq$  reference frame using the Park transform. The current controller regulates the  $d$ -axis and  $q$ -axis currents independently. The resulting voltage references are transformed back to the  $\alpha\beta$  frame through the inverse Park transform and converted into three-phase PWM duty cycles using space vector modulation (SVM). The electrical angle is derived from the encoder measurement according to  $\theta_e = p\theta_m$ , where  $p$  is the number of pole pairs.

### Speed Control Cascade

### Position Control Cascade

### Coordinate System Relationships

See also: [documentation/theory/images/foc\\_coordinates.svg](#) and [documentation/theory/images/svm\\_hexagon.svg](#)

## Numerical Properties

| Property          | Value / Condition                                                                      |
|-------------------|----------------------------------------------------------------------------------------|
| Control rate      | 20 kHz (50 $\mu$ s period)                                                             |
| Transform latency | < 20 CPU cycles (Clarke + Park combined)                                               |
| LUT angle error   | $\leq \pi/512 \approx 0.35^\circ$ truncated; $\approx 1.9 \times 10^{-5}$ interpolated |

---

| Property            | Value / Condition                                                                                                               |
|---------------------|---------------------------------------------------------------------------------------------------------------------------------|
| SVM duty resolution | $2^{-16}$ of the period — <code>PhasePwmDutyCycles</code> carries <code>hal::DutyCycle</code> (Q16 in a <code>uint32_t</code> ) |
| Controller output   | Applied $v_{dq}$ limited to the unit circle $\ v\  \leq 1$ ; anti-windup on that limit                                          |
| Torque linearity    | $\tau_e \propto i_q$ within MTPA region                                                                                         |
| Over-modulation     | Clamp-based saturation; duty cycles bounded to $[0, 1]$                                                                         |
| Gain normalisation  | Physical V/A gains must be scaled by $\sqrt{3}/V_{dc}$ to match per-unit SVM output                                             |
| LUT memory          | $512 \times 4$ bytes = 2048 bytes flash                                                                                         |

---

## Sensitivity Analysis

---

| Parameter          | Effect on Control Quality                                                                                        |
|--------------------|------------------------------------------------------------------------------------------------------------------|
| $R_s$ error        | Current controller steady-state error (for non-integral controllers); compensated by integral term where present |
| $L_s$ error        | Controller bandwidth deviation; no steady-state error for integral-action controllers                            |
| $\theta_e$ error   | Cross-coupling between d and q axes; reduces torque at large error                                               |
| $V_{dc}$ variation | Affects gain normalisation; dynamic Vdc measurement recommended                                                  |
| ADC offset         | Bias on $i_d, i_q$ ; must be calibrated at startup                                                               |

---



---

## Worked Example

Motor parameters:  $R_s = 1.2 \, \Omega$ ,  $L_s = 0.5 \, \text{mH}$ ,  $p = 4$ ,  $\psi_f = 0.05 \, \text{Wb}$ ,  $V_{dc} = 24 \, \text{V}$ .

**Step 1 — Torque at  $i_q = 5 \, \text{A}$ :**

$$\tau_e = \frac{3}{2} \times 4 \times 0.05 \times 5 = 1.5 \, \text{N} \cdot \text{m}$$

**Step 2 — SVM for  $V_\alpha = 8 \, \text{V}$ ,  $V_\beta = 5 \, \text{V}$  ( $V_{dc} = 24 \, \text{V}$ ):**

The modulator takes per-unit voltages, where  $1 \, \text{pu} = V_{dc}/\sqrt{3} = 13.86 \, \text{V}$ . Since  $|V_{ref}| = \sqrt{64 + 25} \approx 9.43 \, \text{V} < 13.86 \, \text{V}$ , the reference lies inside the linear region and no over-modulation clamping occurs.

$$\begin{aligned}
 v_\alpha^{pu} &= 8/13.86 = 0.577, & v_\beta^{pu} &= 5/13.86 = 0.361 \\
 v_A &= 0.577, & v_B &= -0.289 + 0.866 \cdot 0.361 = 0.024, & v_C &= -0.289 - 0.313 = -0.601 \\
 v_{common} &= -\frac{0.577 + (-0.601)}{2} = 0.012 \\
 d_A &= \text{clamp}\left(\frac{0.589}{\sqrt{3}} + 0.5, 0, 1\right) = 0.840, & d_B &= 0.521, & d_C &= 0.160
 \end{aligned}$$


---

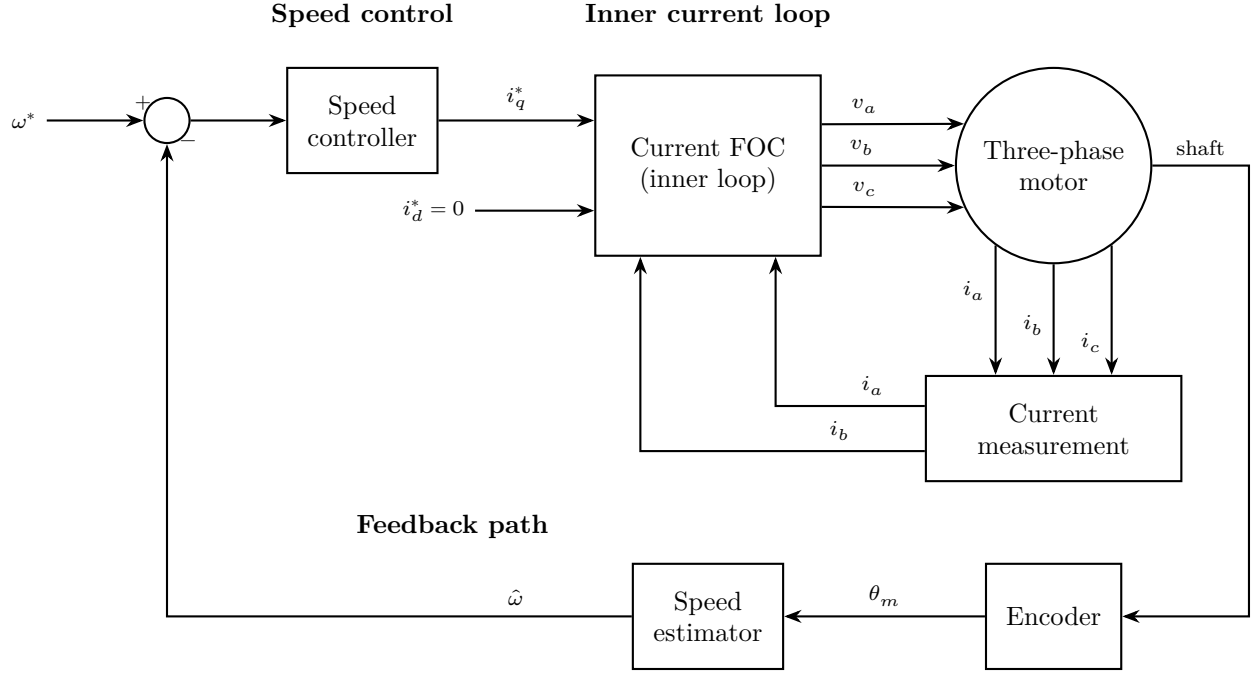


Figure 2: Speed control cascade. The speed controller produces the  $q$ -axis current reference  $i_q^*$  for the inner current FOC loop of Fig. 1, while  $i_d^* = 0$  is passed directly to that loop. Two of the three phase currents are measured and returned to the current loop; the speed estimate  $\hat{\omega}$  is derived from successive encoder readings. The active speed controller algorithm is selected at runtime.

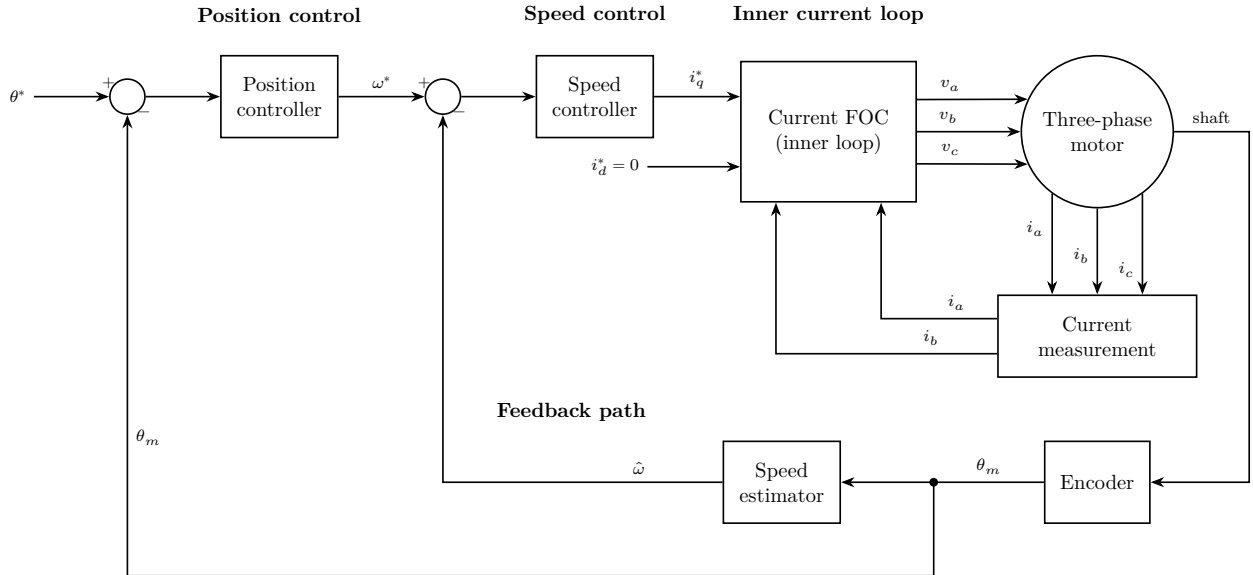


Figure 3: Position control cascade. The position controller produces a speed reference  $\omega^*$ ; the speed controller converts it to a  $q$ -axis current reference  $i_q^*$  for the inner current FOC loop of Fig. 1. The mechanical angle  $\theta_m$  from the encoder closes the position loop directly and also feeds the speed estimator, whose output  $\hat{\omega}$  closes the speed loop. The active algorithm for each loop is selected at runtime.

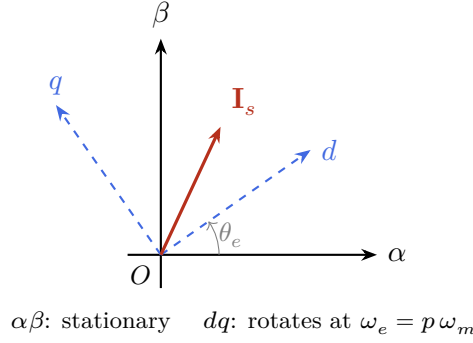


Figure 4: FOC coordinate frames. The  $dq$  frame rotates at electrical speed  $\omega_e = p \omega_m$ . The Park transform decomposes  $\mathbf{I}_s$  into  $i_d$  (flux axis) and  $i_q$  (torque axis).

## Controller Slots

Each of the three control loops exposes a runtime-selectable controller slot. The active algorithm is configured via CLI or CAN without a firmware rebuild, and persists across power cycles via NVM.

- **Current loop** — regulates  $i_d$  and  $i_q$  at 20 kHz.
- **Speed loop** — regulates  $\omega$  at 1 kHz, produces  $i_q^*$  for the current loop.
- **Position loop** — regulates  $\theta$  at 1 kHz, produces  $\omega^*$  for the speed loop.

For design equations of each available algorithm, see the per-algorithm theory chapters. An index of all algorithms with the selection map and parameter sources is in `documentation/theory/advanced-controllers.md`.

Plant models (current, speed, position) and discretisation are in `documentation/theory/foc-plant-models.md`.

The runtime selection mechanism — heap-free variant storage, type-aware dispatch, state gating, NVM persistence, and CLI/CAN interface — is documented in `documentation/design/controller-selection.md`.

---

## Limitations & Assumptions

- **Assumes:** Surface-mounted PMSM with  $L_d = L_q$  (non-salient). For interior PMSM ( $L_d \neq L_q$ ), MTPA condition changes and the torque equation gains an additional reluctance term.
- **Assumes:** Balanced three-phase supply ( $i_a + i_b + i_c = 0$ ). If a phase is open, Clarke reduction from two sensors is invalid.
- **Assumes:** The electrical rotor angle  $\theta_e$  is accurately known (from encoder + alignment offset). Angle errors directly degrade torque production and can cause instability.
- **Does not handle:** Flux weakening above base speed (field weakening via non-zero  $i_d$ ).
- **Does not handle:** Sensorless operation (this implementation uses an encoder for  $\theta_e$ ).

## References

1. Krishnan, R. — *Permanent Magnet Synchronous and Brushless DC Motor Drives*, CRC Press, 2010.
2. Holmes, D.G. & Lipo, T.A. — *Pulse Width Modulation for Power Converters*, IEEE Press / Wiley, 2003.
3. Bose, B.K. — *Modern Power Electronics and AC Drives*, Prentice Hall, 2002.
4. Texas Instruments Application Report SPRAA76A — *Adjusted Space Vector Pulse Width Modulation for Voltage Source Inverters*, 2005.

| Field     | Value                                                         |
|-----------|---------------------------------------------------------------|
| Title     | FOC Plant Models — Discretization and State-Space Foundations |
| Type      | theory                                                        |
| Status    | draft                                                         |
| Version   | 0.2.0                                                         |
| Component | foc-plant-models                                              |
| Date      | 2026-09-21                                                    |

**Theory document:** Explains the mathematical and engineering principles behind a component or algorithm. This document is descriptive — it records the *why* and *how* at a scientific level, independent of any specific implementation. Equations use KaTeX (*inline* and

*block*

). Block diagrams use Mermaid or ASCII art.

## Overview

This document derives the discrete-time state-space plant models used by all advanced FOC controllers. It is a shared prerequisite for: - the current, speed and position loop controllers indexed in `documentation/theory/advanced-controllers.md`

The reader is assumed familiar with Clarke/Park transforms, SVM, and PI current control as described in `documentation/theory/foc.md`.

## Prerequisites

| Symbol     | Meaning                                         | Unit     |
|------------|-------------------------------------------------|----------|
| $R_s$      | Stator resistance per phase                     | $\Omega$ |
| $L_s$      | Stator inductance ( $L_d = L_q$ , surface PMSM) | H        |
| $\psi_f$   | Permanent magnet flux linkage                   | Wb       |
| $p$        | Number of pole pairs                            | —        |
| $\omega_e$ | Electrical angular velocity                     | rad/s    |

| Symbol     | Meaning                                        | Unit                                                |
|------------|------------------------------------------------|-----------------------------------------------------|
| $K_t$      | Torque constant = $\frac{3}{2}p\psi_f$         | $\text{N} \cdot \text{m}/\text{A}$                  |
| $J$        | Rotor moment of inertia                        | $\text{kg} \cdot \text{m}^2$                        |
| $B_f$      | Viscous friction coefficient                   | $\text{N} \cdot \text{m} \cdot \text{s}/\text{rad}$ |
| $T_L$      | Load torque (disturbance), see the split below | $\text{N} \cdot \text{m}$                           |
| $\omega_m$ | Mechanical angular velocity                    | $\text{rad}/\text{s}$                               |
| $\theta_m$ | Mechanical rotor angle                         | $\text{rad}$                                        |
| $T_s^i$    | Current loop sample period = $1/20000$ s       | s                                                   |
| $T_s^o$    | Outer loop sample period = $1/1000$ s          | s                                                   |
| $A_d, B_d$ | Discrete-time plant matrices                   | —                                                   |

## Mathematical Foundation

### 1. PMSM Current Loop Plant (dq Frame)

The PMSM voltage equations in the rotor-synchronous dq frame (from `documentation/theory/-foc.md` Section 4), assuming a surface-mounted motor with  $L_d = L_q = L_s$ :

$$v_d = R_s i_d + L_s \frac{di_d}{dt} - \omega_e L_s i_q$$

$$v_q = R_s i_q + L_s \frac{di_q}{dt} + \omega_e L_s i_d + \omega_e \psi_f$$

Rearranged as explicit state equations:

$$\frac{di_d}{dt} = -\frac{R_s}{L_s} i_d + \omega_e i_q + \frac{1}{L_s} v_d$$

$$\frac{di_q}{dt} = -\frac{R_s}{L_s} i_q - \omega_e i_d - \frac{\psi_f \omega_e}{L_s} + \frac{1}{L_s} v_q$$

The cross-coupling terms  $+\omega_e i_q$  (on d) and  $-\omega_e i_d$  (on q), and the back-EMF disturbance  $-\psi_f \omega_e / L_s$  (on q), prevent the two axes from being treated as independent RL plants. Advanced current controllers either cancel these terms via feedforward or treat them as bounded disturbances.

### Decoupled Per-Axis Plant

After applying the feedforward voltages defined in the current-loop controllers document, each axis reduces to an independent first-order RL system with decoupled input  $v'$ :

$$\frac{di}{dt} = -\frac{R_s}{L_s} i + \frac{1}{L_s} v'$$

## Discrete-Time Current Plant

ZOH discretization at  $T_s^i$  gives per-axis matrices:

$$A_d^i = e^{-R_s T_s^i / L_s}, \quad B_d^i = \frac{1 - A_d^i}{R_s}$$

**This exact pair is what the implementation uses**, evaluated with a real exponential — not the Euler approximation below.

For  $R_s T_s^i / L_s \ll 1$  (typical:  $< 0.1$ ), the Euler approximation is accurate:

$$A_d^i \approx 1 - \frac{R_s T_s^i}{L_s}, \quad B_d^i \approx \frac{T_s^i}{L_s}$$

Note that  $B_d^i$  must be taken from the same discretization as  $A_d^i$ . Pairing the exact  $A_d^i = e^{-R_s T_s^i / L_s}$  with the Euler  $B_d^i = T_s^i / L_s$  is not a ZOH model of anything; the matching exact input matrix is  $(1 - A_d^i) / R_s$ .

The per-axis discrete plant:

$$i[k+1] = A_d^i \cdot i[k] + B_d^i \cdot v'[k]$$

RLS estimates  $\hat{R}_s$  and  $\hat{L}_s$  are substituted at configuration time.

## 2. Mechanical (Speed) Plant

The mechanical rotor dynamics driven by electromagnetic torque:

$$J \frac{d\omega_m}{dt} = K_t i_q - B_f \omega_m - T_L$$

The simulated plant splits  $T_L$  into two terms. A *load* opposes motion whatever its direction, so it changes sign with  $\omega_m$  and vanishes at rest, like friction. An *external torque* keeps its sign whatever the rotor does, like gravity on an arm or a step applied by a test to disturb the loop, and it is the term a position controller has to hold against. Both are additive on the right-hand side:

$$T_L = T_{load} \cdot \text{sgn}(\omega_m) + T_{ext}$$

Treating the current loop as ideal ( $i_q \approx i_q^*$ ), the outer-loop control input is  $u = i_q^*$ .

$$\frac{d\omega_m}{dt} = -\frac{B_f}{J} \omega_m + \frac{K_t}{J} u - \frac{T_L}{J}$$

This is a first-order plant (state:  $\omega_m$ , input:  $i_q^*$ , disturbance:  $T_L/J$ ).

Discretization at  $T_s^o$ . The exact ZOH pair is:

$$A_d^o = e^{-B_f T_s^o/J}, \quad B_d^o = \frac{K_t}{B_f} (1 - A_d^o)$$

The implementation uses the forward-Euler pair instead:

$$A_d^o = 1 - \frac{B_f T_s^o}{J}, \quad B_d^o = \frac{K_t T_s^o}{J}$$

Both matrices come from the same approximation, which is the part that matters: the exact  $A_d^o$  paired with the Euler  $B_d^o$  would be a model of no system at all. The Euler pair is valid while  $B_f T_s^o/J \ll 1$ , which holds comfortably at 1 kHz for the inertias and frictions the mechanical identification returns, and unlike the exact pair it stays defined for a frictionless load, where  $K_t(1 - A_d^o)/B_f$  is 0/0.

$$\omega_m[k+1] = A_d^o \cdot \omega_m[k] + B_d^o \cdot u[k]$$

RLS estimates  $\hat{J}$  and  $\hat{B}_f$  from `documentation/theory/friction-inertia-estimation.md` are substituted at configuration time.

---

### 3. Position Plant (Double Integrator)

Adding the kinematic relation  $\dot{\theta}_m = \omega_m$  yields a two-state plant:

$$\begin{pmatrix} \dot{\theta}_m \\ \dot{\omega}_m \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 0 & -B_f/J \end{pmatrix} \begin{pmatrix} \theta_m \\ \omega_m \end{pmatrix} + \begin{pmatrix} 0 \\ K_t/J \end{pmatrix} u$$

Discretized at  $T_s^o$  with the same forward-Euler  $A_d^o$  and  $B_d^o$  as the speed plant:

$$A_d^p = \begin{pmatrix} 1 & T_s^o \\ 0 & A_d^o \end{pmatrix}, \quad B_d^p = \begin{pmatrix} 0 \\ B_d^o \end{pmatrix}$$

The  $[1, T_s^o]$  row approximation holds when  $B_f T_s^o/J \ll 1$ . For high-friction systems, the exact matrix exponential should be used.

---

## Block Diagrams

### Current Loop Plant Structure

### Speed and Position Plant Cascade

---

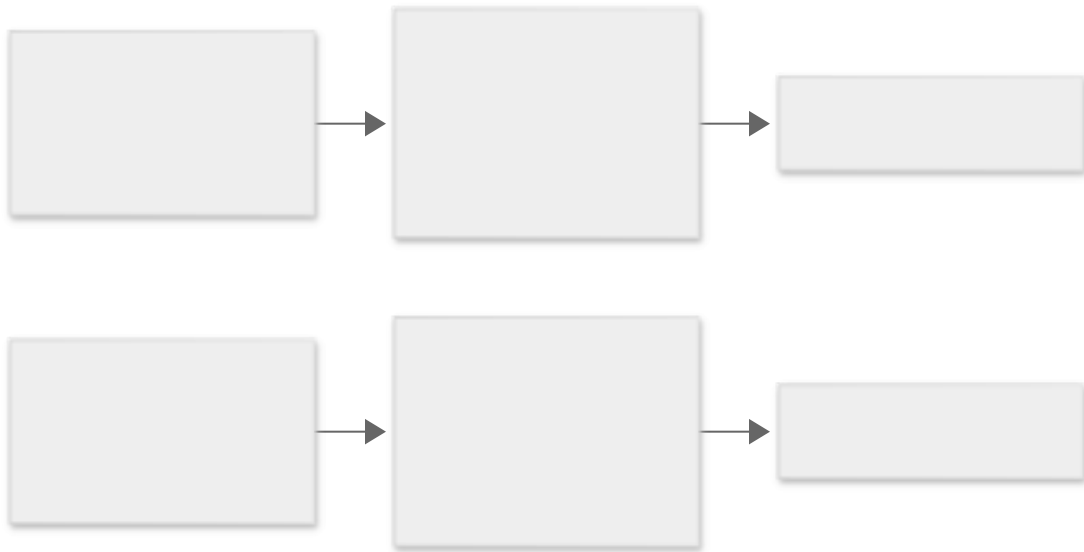


Figure 5: Block diagram

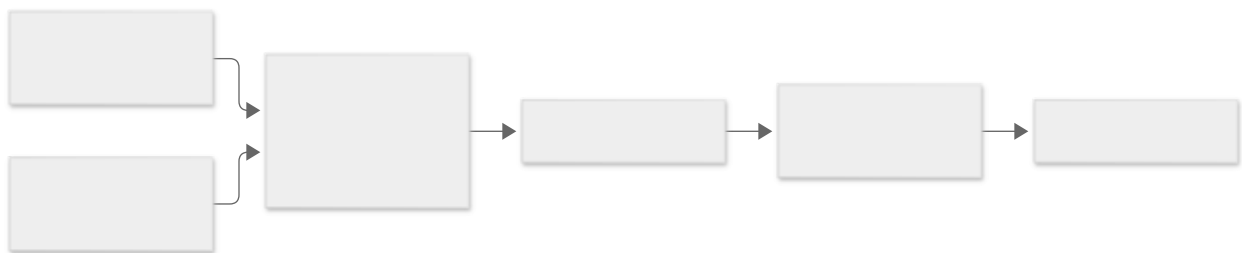


Figure 6: Block diagram

## Numerical Properties

| Property             | Current plant                   | Speed plant                   | Position plant             |
|----------------------|---------------------------------|-------------------------------|----------------------------|
| States               | 1 per axis (Id, Iq)             | 1 ( $\omega_m$ )              | 2 ( $\theta_m, \omega_m$ ) |
| Inputs               | 1 per axis (v'd, v'q)           | 1 (Iq*)                       | 1 (Iq*)                    |
| Time constant        | $L_s/R_s$ (typ. 0.4 ms)         | $J/B_f$ (typ. 0.1–2 s)        | Integrating                |
| Discretization error | $< 1\%$ for $R_s T_s/L_s < 0.1$ | $< 1\%$ for $B_f T_s/J < 0.1$ | Row approx valid at 1 kHz  |
| Parameter source     | Electrical RLS                  | Mechanical RLS                | Mechanical RLS             |

## Reference Motors

The plant model is exercised, in the unit tests, the simulator and the software-in-the-loop suite, with two motors whose parameters are published by their vendors and by the evaluation kits built around them. The values below are per phase for a wye winding, which is what the model and the identification services use. The headers under `motor_parameters/` carry exactly these numbers.

| Parameter                         | Symbol                     | Anaheim Automation<br>BLY172S-24V-4000 | Teknic<br>M-2310P-LN-04K           |
|-----------------------------------|----------------------------|----------------------------------------|------------------------------------|
| Rated bus voltage                 | $V_{dc}$                   | 24 V                                   | 40 V                               |
| Rated speed, power, torque        |                            | 4000 rpm, 53 W, 0.127 N · m            | 6000 rpm, 170 W, 0.274 N · m       |
| Phase resistance                  | $R_s$                      | 0.405 $\Omega$                         | 0.36 $\Omega$                      |
| Phase inductance                  | $L_d = L_q$                | 0.64 mH                                | 0.20 mH                            |
| Flux linkage                      | $\psi_f$                   | 5.44 mWb                               | 6.40 mWb                           |
| Pole pairs                        | $p$                        | 4                                      | 4                                  |
| Rotor inertia                     | $J$                        | 4.8e-6 kg · m <sup>2</sup>             | 7.06e-6 kg · m <sup>2</sup>        |
| Viscous friction                  | $B_f$                      | 1.0e-5 N · m · s/rad (assumed)         | 1.5e-5 N · m · s/rad (assumed)     |
| Torque constant                   | $K_t = \frac{3}{2}p\psi_f$ | 32.6 mN · m/A                          | 38.4 mN · m/A                      |
| Electrical time constant          | $L_s/R_s$                  | 1.58 ms                                | 0.56 ms                            |
| Drive current rating in the model | $I_{max}$                  | 10 A                                   | 20 A                               |
| Header                            |                            | <code>AnaheimBly172s24v4000.hpp</code> | <code>teknicM2310pLn04k.hpp</code> |

How the numbers were obtained:

- **Line-to-line to per-phase.** Datasheets quote terminal values; for a wye winding  $R_s = R_{LL}/2$  and  $L_s = L_{LL}/2$ . The Teknic sheet gives 0.72  $\Omega$  and 0.40 mH line-to-line. The Anaheim sheet gives 1.20 mH line-to-line (0.60 mH per phase); the 0.64 mH and 0.405  $\Omega$  in the header are TI's InstaSPIN identification of the same motor, which agrees with the sheet to within the spread expected between a six-step datasheet rating and a sinusoidal measurement.

- **Flux linkage from back-EMF.** With a line-to-line peak constant  $K_e$  per 1000 rpm ( $\omega_m = 104.7$  rad/s,  $\omega_e = p\omega_m$ ):  $\psi_f = K_e/(\sqrt{3}\omega_e)$ . Teknic:  $4.64/(\sqrt{3} \cdot 418.9) = 6.40$  mWb. TI reports the Anaheim flux as 0.03416 V/Hz, a peak phase voltage per electrical hertz, so  $\psi_f = 0.03416/2\pi = 5.44$  mWb.
- **Inertia.**  $1 \text{ oz} \cdot \text{in} \cdot \text{s}^2 = 7.0616 \times 10^{-3} \text{ kg} \cdot \text{m}^2$ : 0.00068 oz · in · s<sup>2</sup> (Anaheim) and 0.001 oz · in · s<sup>2</sup> (Teknic).
- **Viscous friction.** Neither vendor publishes a viscous term or a no-load current to derive one from. The values are chosen so that friction torque at rated speed is about 3 % of rated torque (4.2 mN · m at 419 rad/s for the Anaheim, 9.4 mN · m at 628 rad/s for the Teknic), and the identification scenarios treat them as plant truth, not as a claim about the real machines.
- **Drive rating.** `maxSupportedCurrent` is the inverter's limit, not the motor's: 10 A and 20 A are what a small drive for each class supports and what the injection-limit check in every calibration procedure compares against. The 15 % DC resistance test draws about 5.5 A (Anaheim, 24 V) and 10 A (Teknic, 40 V), inside both ratings.

The two motors bracket the plant in the directions that matter to the estimators: the Teknic has the shorter electrical time constant (11 control ticks at 20 kHz, still well inside the Euler stability region), and its  $R_s T_s / L_s$  of 0.18 at the 10 kHz identification rate makes the ZOH bias of the inductance estimator visible (see `resistance-inductance-estimation.md`), where the Anaheim's 0.063 keeps it small.

Sources:

- Anaheim Automation, *BLY17 Series Product Sheet* (L010228) and the [BLY172S-24V-4000 product page](#): 24 V, 4000 rpm, 53 W, 18 oz · in, 1.20 mH line-to-line, 3.35 V/krpm, 0.00068 oz · in · s<sup>2</sup>.
- Texas Instruments MotorWare `user.h`, motor `Anaheim_BLY172S`:  $R_s = 0.4051 \Omega$ ,  $L_s = 0.6399$  mH, rated flux 0.03416 V/Hz, 4 pole pairs.
- Teknic, *Industrial-Grade NEMA 23 Motors* datasheet ([N23\\_Industrial\\_Grade\\_Motors\\_v6.3.pdf](#)): M-2310P-LN-04K, 40 V, 6000 rpm, 0.72  $\Omega$  and 0.40 mH line-to-line, 4.64 Vpk/krpm, 0.001 oz · in · s<sup>2</sup>.
- NXP, *MCUXpresso SDK FOC user guide*, [Teknic motor](#), which restates the same electrical values for the FRDM-MC kits, and the TI [LVSERVOMTR](#) page for the 2MTR-DYNO.

## References

1. Krishnan, R. — *Permanent Magnet Synchronous and Brushless DC Motor Drives*, CRC Press, 2010.
2. Åström, K.J. & Wittenmark, B. — *Computer-Controlled Systems: Theory and Design*, 3rd ed., Prentice Hall, 1997. (Chapter 3: ZOH discretization.)

| Field     | Value                            |
|-----------|----------------------------------|
| Title     | Advanced FOC Controllers — Index |
| Type      | theory                           |
| Status    | draft                            |
| Version   | 0.3.0                            |
| Component | foc-controllers                  |

---

|       |            |
|-------|------------|
| Field | Value      |
| Date  | 2026-08-31 |

---

## Overview

This index covers the controller algorithms available for the three nested FOC loops. Each loop exposes a runtime-selectable controller slot configured via CLI or CAN without a firmware rebuild. Per-algorithm chapters contain the full mathematical foundation and design equations.

The shared plant models (dq current, mechanical speed, position) and discretisation are in `documentation/theory/foc-plant-models.md`.

The runtime selection mechanism — heap-free variant storage, type-aware dispatch, state gating, NVM persistence, and CLI/CAN interface — is in `documentation/design/controller-selection.md`.

---

## Algorithm Map

---

## Mathematical Foundation

All controllers derive from three plant models established in `documentation/theory/foc-plant-models.md`:

| Plant                              | States               | Input                   | Loop rate |
|------------------------------------|----------------------|-------------------------|-----------|
| PMSM current (per axis, decoupled) | $i_d, i_q$           | normalised voltage $v'$ | 20 kHz    |
| Mechanical speed                   | $\omega_m$           | $i_q^*$                 | 1 kHz     |
| Position (double integrator)       | $\theta_m, \omega_m$ | $i_q^*$                 | 1 kHz     |

---

Full derivations, ZOH discretisation, and parameter sources are in `documentation/theory/foc-plant-models.md`. Per-algorithm design equations are in the individual chapters listed in *Algorithm Index* below.

---

## Numerical Properties

Summary comparison across all loops — see each per-algorithm chapter for full detail.

**Current loop** — 20 kHz ISR:

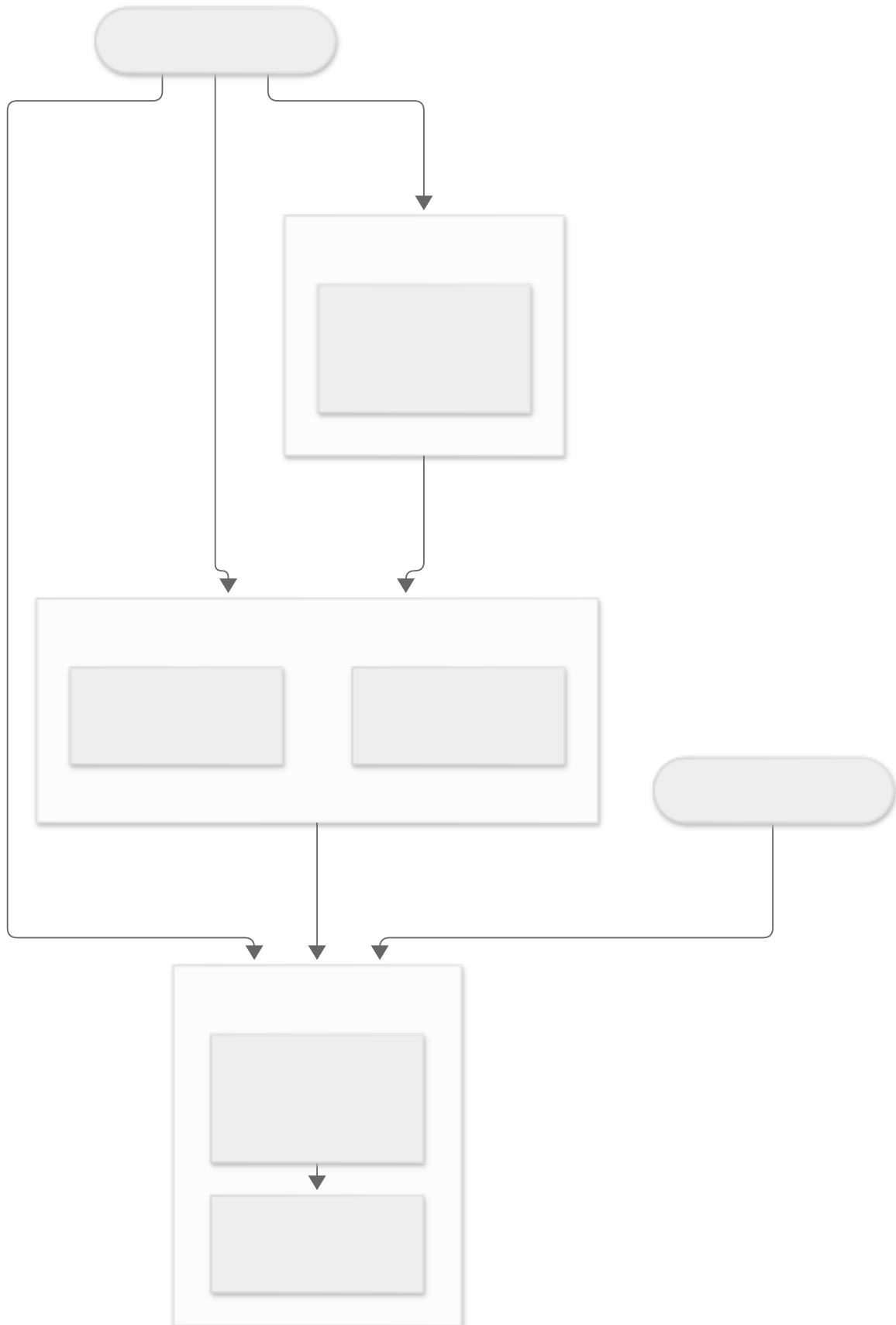


Figure 7: Block diagram

| ID | Algorithm     | ISR cost | Robustness          | Requires RLS |
|----|---------------|----------|---------------------|--------------|
| —  | PI (baseline) | ~6 MACs  | High (integral)     | No           |
| A1 | Decoupled PID | ~10 MACs | Low (model-dep.)    | Partial      |
| A2 | Deadbeat      | ~4 MACs  | Low (exact model)   | Yes (tight)  |
| A3 | Sliding-mode  | ~12 MACs | High (gain-bounded) | Partial      |

**Speed loop** — 1 kHz handler:

| ID | Algorithm     | Ops/cycle | Disturbance rejection | Requires RLS |
|----|---------------|-----------|-----------------------|--------------|
| —  | PI (baseline) | ~6 MACs   | Integral quality      | No           |
| S1 | LQI           | 2 MACs    | Integral quality      | Yes (J, Bf)  |
| S2 | ADRC          | 6 MACs    | Explicit cancellation | Yes (Kt, J)  |
| S3 | Two-DOF       | ~8 MACs   | Integral quality      | No           |

**Position loop** — 1 kHz handler:

| ID | Algorithm    | Ops/cycle      | Steady-state error  | Requires RLS |
|----|--------------|----------------|---------------------|--------------|
| —  | P (baseline) | 2 MACs         | Zero at rest        | No           |
| P1 | LQR / LQI    | 4 MACs         | Zero (LQI)          | Yes (J, Bf)  |
| P2 | Cascade P    | 2 MACs         | Speed-loop dep.     | No           |
| P3 | Two-DOF      | ~8 MACs        | Configurable        | No           |
| P4 | ILC          | 2 MACs + array | Near-zero (learned) | No           |

## Algorithm Index

| Loop     | ID | Algorithm                   | File                                      |
|----------|----|-----------------------------|-------------------------------------------|
| Current  | —  | PI (baseline)               | current-loop-pi.md (Chapter 4)            |
| Current  | A1 | Decoupled PID + Feedforward | current-loop-decoupled-pid.md (Chapter 5) |
| Current  | A2 | Deadbeat                    | current-loop-deadbeat.md (Chapter 6)      |
| Current  | A3 | Sliding-Mode                | current-loop-sliding-mode.md (Chapter 7)  |
| Speed    | —  | PI (baseline)               | speed-loop-pi.md (Chapter 8)              |
| Speed    | S1 | LQI                         | speed-loop-lqi.md (Chapter 9)             |
| Speed    | S2 | ADRC                        | speed-loop-adrc.md (Chapter 10)           |
| Speed    | S3 | Two-DOF                     | speed-loop-two-dof.md (Chapter 11)        |
| Position | —  | P (baseline)                | position-loop-pid.md (Chapter 12)         |
| Position | P1 | LQR / LQI                   | position-loop-lqr-lqi.md (Chapter 13)     |
| Position | P2 | Cascade P→PI                | position-loop-cascade-p.md (Chapter 14)   |
| Position | P3 | Two-DOF                     | position-loop-two-dof.md (Chapter 15)     |
| Position | P4 | Iterative Learning Control  | position-loop-ilc.md (Chapter 16)         |

---

| Loop | ID | Algorithm             | File                                   |
|------|----|-----------------------|----------------------------------------|
| Any  | —  | Friction Compensation | position-loop-friction.md (Chapter 17) |

---

## Parameter Sources

| Source                  | Parameters                                                        | Available after                 |
|-------------------------|-------------------------------------------------------------------|---------------------------------|
| Motor datasheet         | Pole pairs $p$ , flux linkage $\psi_f$ , peak current $I_{q,max}$ | Before first run                |
| Electrical RLS          | $R_s$ , $L_s$ , $V_{dc}$                                          | After electrical identification |
| Mechanical RLS          | $J$ , $B_f$ , $K_t$                                               | After mechanical identification |
| Friction identification | $T_c$ , $T_s$ , $\omega_{st}$                                     | After dedicated friction sweep  |

---

Current-loop algorithms A1–A3 become selectable after electrical identification. Speed and position algorithms S1, S2, P1 additionally require mechanical identification. P2, P3, P4, and S3 require neither and are always available once the motor is calibrated.

---

## References

Full reference lists are in each per-algorithm chapter. Foundational references:

1. Krishnan, R. — *Permanent Magnet Synchronous and Brushless DC Motor Drives*, CRC Press, 2010.
2. Åström, K.J. & Wittenmark, B. — *Computer-Controlled Systems*, 3rd ed., Prentice Hall, 1997.
3. Han, J. — “From PID to Active Disturbance Rejection Control”, *IEEE Trans. Ind. Electron.*, 56(3):900–906, 2009.
4. Bristow, D.A. et al. — “A Survey of Iterative Learning Control”, *IEEE Control Systems Magazine*, 26(3):96–114, 2006.
5. Armstrong-Hélouvry, B. et al. — “A Survey of Models ... for Machines with Friction”, *Automatica*, 30(7):1083–1138, 1994.

| Field     | Value                        |
|-----------|------------------------------|
| Title     | Current Loop — PI (Baseline) |
| Type      | theory                       |
| Status    | draft                        |
| Version   | 1.0.0                        |
| Component | current-loop-pi              |
| Date      | 2026-08-31                   |

---

**Theory document:** Explains the mathematical and engineering principles behind a component or algorithm. This document is descriptive — it records the *why* and *how* at a scientific level, independent of any specific implementation. Equations use KaTeX (*inline* and

*block*

). Block diagrams use Mermaid or ASCII art.

---

## Overview

The baseline current controller is a discrete incremental PI regulator applied independently to the d- and q-axis current errors in the rotating dq frame. It is always available — no RLS convergence is required — and its pole-zero cancellation design places the closed-loop bandwidth precisely at the specified value  $\omega_{bw}^i$ . All advanced current controllers (A1–A3) supplement or replace this baseline while preserving the same output interface.

Operates exclusively in the **20 kHz FOC ISR**.

---

## Prerequisites

| Symbol         | Meaning                                     | Unit     |
|----------------|---------------------------------------------|----------|
| $R_s$          | Stator resistance per phase                 | $\Omega$ |
| $L_s$          | Stator inductance ( $L_d = L_q$ )           | H        |
| $V_{dc}$       | DC bus voltage (measured dynamically)       | V        |
| $T_s^i$        | Current loop sample period ( $= 50 \mu s$ ) | s        |
| $\omega_{bw}$  | Desired closed-loop current bandwidth       | rad/s    |
| $A_d^i, B_d^i$ | Discrete RL plant matrices                  | —        |

See `documentation/theory/foc-plant-models.md` for all base symbols and ZOH derivations.

---

## Mathematical Foundation

### Plant Model

After feedforward decoupling the cross-coupling terms (see `documentation/theory/foc.md` §4), each dq axis reduces to an independent first-order RL plant with decoupled input  $v'$ :

$$\frac{di}{dt} = -\frac{R_s}{L_s} i + \frac{1}{L_s} v'$$

ZOH-discretized at  $T_s^i$  (from `documentation/theory/foc-plant-models.md` §1):

$$i[k+1] = A_d^i i[k] + B_d^i v'[k], \quad A_d^i = e^{-R_s T_s^i / L_s}, \quad B_d^i = \frac{1 - A_d^i}{R_s}$$

## PI Gain Design — Pole-Zero Cancellation

The continuous-time RL plant has a single pole at  $s = -R_s/L_s$ . A PI controller:

$$C(s) = K_p \cdot \frac{s + K_i/K_p}{s}$$

achieves pole-zero cancellation by placing the PI zero at the plant pole:

$$\frac{K_i}{K_p} = \frac{R_s}{L_s}$$

The closed-loop transfer function then reduces to a first-order system with bandwidth  $\omega_{bw}$ :

$$K_p = L_s \omega_{bw}, \quad K_i = R_s \omega_{bw}$$

**Practical limit:**  $\omega_{bw} \leq \pi/T_s^i$  (Nyquist). The maximum achievable bandwidth at 20 kHz is roughly  $\omega_{bw} \leq 10000$  rad/s; conservative designs use 2000–5000 rad/s.

## Gain Normalisation

The modulator accepts per-unit voltages where  $1 \text{ pu} = V_{dc}/\sqrt{3}$ . Physical PI gains must be scaled before being applied to the normalised output:

$$K_p^{norm} = K_p \cdot \frac{\sqrt{3}}{V_{dc}} = L_s \omega_{bw} \cdot \frac{\sqrt{3}}{V_{dc}}$$

$V_{dc}$  is measured dynamically and used to update normalisation in real time. Without this scaling the effective bandwidth changes with bus voltage.

## Incremental (Velocity) Form

The implementation uses the incremental (velocity) PI form to avoid integrator initialisation issues. Each axis proposes

$$\begin{aligned} \Delta v'[k] &= K_p \cdot (e[k] - e[k-1]) + K_i T_s^i e[k] \\ \tilde{v}'[k] &= v'[k-1] + \Delta v'[k] \end{aligned}$$

where  $v'[k-1]$  is the value the actuator **actually applied** on the previous sample, not the value this axis proposed.

## Anti-Windup

The actuator constraint here is the modulation circle (see *Output Voltage Limit* below). It couples  $d$  and  $q$ , so it cannot be expressed as the independent per-axis clamp a stock PID carries: a per-axis clamp to  $[-1, 1]$  never fires for a demand such as  $v'_d = v'_q = 0.8$ , yet that vector has magnitude 1.13 and the circle limiter scales it down. Both integrators would then accumulate against a limit they cannot see.

The loop therefore uses back-calculation against the realised output. Both axes propose, the circle limit is applied once to the resulting vector, and the applied components are committed back as  $v'[k-1]$  for the next sample:

$$v'[k] = \Pi(\tilde{v}'_d[k], \tilde{v}'_q[k])$$

where  $\Pi$  is the projection onto the unit disc. When the demand is inside the circle the projection is the identity and the law reduces to the plain velocity form.

---

## Output Voltage Limit

Every current controller in this loop emits a normalised dq voltage pair passed to inverse Park and then to SVM. The modulator remains linear only inside the circle inscribed in the voltage hexagon (`documentation/theory/foc.md §6`):

$$\sqrt{(v'_d)^2 + (v'_q)^2} \leq 1 \quad \Longleftrightarrow \quad \sqrt{v_d^2 + v_q^2} \leq \frac{V_{dc}}{\sqrt{3}}$$

This is a **circular** constraint on the vector, not an independent per-axis bound. When the vector demand exceeds the unit circle, both components are scaled by  $1/\|v'\|$ , preserving voltage angle (and therefore current vector direction) while sacrificing magnitude.

Clamping  $v'_d$  and  $v'_q$  separately to  $[-1, 1]$  admits vectors up to magnitude  $\sqrt{2}$ , driving the modulator into over-modulation over roughly three-quarters of the electrical period.

---

## Numerical Properties

| Property                  | Value                                    |
|---------------------------|------------------------------------------|
| ISR cost                  | ~6 MACs (2 axes $\times$ 3 operations)   |
| Settling time             | $\sim 1/\omega_{bw}$                     |
| Robustness to Rs/Ls error | High — integral action corrects mismatch |
| Requires RLS convergence  | No — motor datasheet values sufficient   |
| Requires $\psi_f$         | No                                       |
| Tuning knobs              | 1 ( $\omega_{bw}$ ) + 1 ( $V_{dc}$ live) |

---

## Limitations & Assumptions

- Cross-coupling terms  $\omega_e L_s i_q$  and  $\omega_e L_s i_d$  are treated as disturbances. At high electrical speed  $\omega_e$ , the coupling voltage reduces effective bandwidth and introduces cross-axis torque ripple. Use A1 (Decoupled PID) to cancel this at speed.
- Assumes balanced three-phase operation ( $i_a + i_b + i_c = 0$ ).
- Requires accurate electrical angle  $\theta_e$ . Angle errors couple directly into the dq frame.
- Output subject to the circular SVM constraint — see *Output Voltage Limit* above.

## References

1. Krishnan, R. — *Permanent Magnet Synchronous and Brushless DC Motor Drives*, CRC Press, 2010.
2. Åström, K.J. & Wittenmark, B. — *Computer-Controlled Systems*, 3rd ed., Prentice Hall, 1997.
3. Holmes, D.G. & Lipo, T.A. — *Pulse Width Modulation for Power Converters*, IEEE Press, 2003.

| Field     | Value                                          |
|-----------|------------------------------------------------|
| Title     | Current Loop — A1: Decoupled PID + Feedforward |
| Type      | theory                                         |
| Status    | draft                                          |
| Version   | 1.0.0                                          |
| Component | current-loop-decoupled-pid                     |
| Date      | 2026-08-31                                     |

**Theory document:** Explains the mathematical and engineering principles behind a component or algorithm. This document is descriptive — it records the *why* and *how* at a scientific level, independent of any specific implementation. Equations use KaTeX (*inline* and

*block*

). Block diagrams use Mermaid or ASCII art.

## Overview

A1 augments the baseline PI current controller with an explicit feedforward that cancels the cross-axis coupling terms  $\omega_e L_s i_q$  and  $\omega_e L_s i_d$  and the back-EMF disturbance  $\omega_e \psi_f$ . At high electrical speed  $\omega_e$ , these coupling voltages become large relative to the RL drive voltage, reducing effective PI bandwidth and introducing cross-axis torque ripple. Explicit cancellation restores full PI bandwidth independently of speed.

Operates exclusively in the **20 kHz FOC ISR**. Requires electrical RLS estimates ( $L_s$ ,  $R_s$ ) and the motor constant  $\psi_f$ .

## Prerequisites

| Symbol               | Meaning                                | Unit       |
|----------------------|----------------------------------------|------------|
| $A_d^i, B_d^i$       | Discrete current plant matrices        | —          |
| $v_d^{PI}, v_q^{PI}$ | PI controller outputs (decoupled axes) | normalised |
| $\omega_e$           | Electrical angular velocity            | rad/s      |
| $L_s$                | Stator inductance                      | H          |
| $\psi_f$             | Permanent magnet flux linkage          | Wb         |
| $V_{dc}$             | DC bus voltage                         | V          |

See `documentation/theory/foc-plant-models.md` for all base symbols. See `documentation/theory/current-loop-pi.md` for the baseline PI design and gain normalisation.

## Mathematical Foundation

All current controllers operate on the **decoupled per-axis RL plant** derived in `documentation/theory/foc-plant-models.md` §1. After feedforward decoupling, each axis reduces to an independent first-order system:

$$i[k+1] = A_d^i \cdot i[k] + B_d^i \cdot v'[k], \quad A_d^i = e^{-R_s T_s^i / L_s}, \quad B_d^i = \frac{1 - A_d^i}{R_s}$$

A1 derives the feedforward law that recovers this decoupled plant from the coupled PMSM voltage equations and then applies a standard PI.

## A1 — Decoupled PID + Feedforward

**Motivation:** Plain PI controllers treat the cross-coupling terms  $\omega_e L_s i_q$  and  $-\omega_e L_s i_d$  as disturbances. At high electrical speed  $\omega_e$ , the coupling voltage becomes large relative to the RL drive voltage, reducing effective PI bandwidth and introducing cross-axis torque ripple. Explicit cancellation restores full PI bandwidth independently of speed.

**Feedforward law:** The total applied voltage cancels coupling before presenting the decoupled RL plant to the PI regulator:

$$v_d = v_d^{PI} - \omega_e L_s i_q$$

$$v_q = v_q^{PI} + \omega_e L_s i_d + \omega_e \psi_f$$

After applying these voltages, both axes reduce to the independent RL plant and the PI controllers are designed identically to the standard case.

**Normalisation:** Feedforward terms are scaled to the same normalised unit as the PI output (normalisation scale  $\sqrt{3}/V_{dc}$ ):

$$f_d = -\omega_e L_s i_q \cdot \frac{\sqrt{3}}{V_{dc}}, \quad f_q = (\omega_e L_s i_d + \omega_e \psi_f) \cdot \frac{\sqrt{3}}{V_{dc}}$$

**Gain design:** PI gains are identical to the pole-zero cancellation design in `documentation/theory/current-loop-pi.md`. The feedforward is additive and does not change the closed-loop transfer function.

**RLS dependency:**  $L_s$  from the electrical RLS estimator.  $\psi_f$  is a motor constant calibrated during alignment.  $V_{dc}$  is measured dynamically.

**Performance:** At  $\omega_e = 1000$  rad/s with  $L_s = 0.5$  mH and  $i_q = 5$  A, the uncorrected coupling is  $\omega_e L_s i_q = 2.5$  V — 10% of a 24 V bus. This is the dominant current control error at speed for an uncompensated PI.

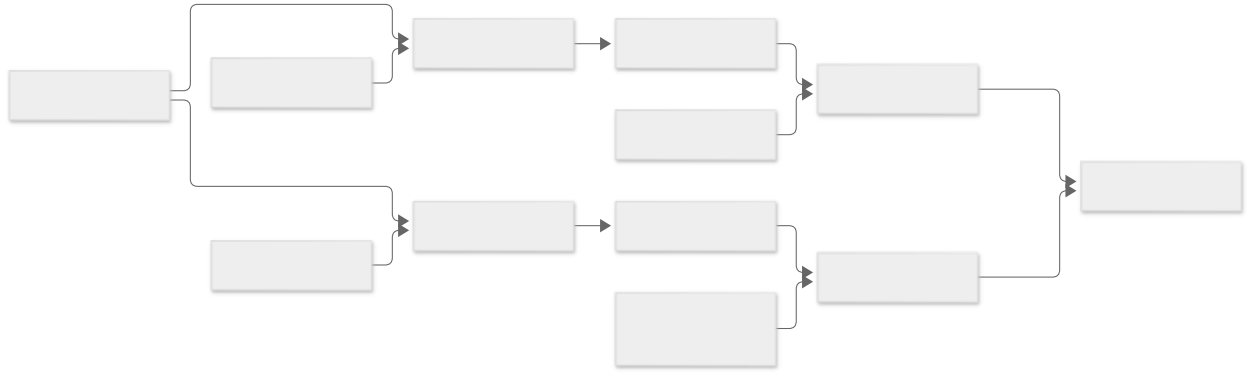


Figure 8: Block diagram

## Output Voltage Limit

The output voltage vector is subject to the circular SVM constraint:

$$\sqrt{(v'_d)^2 + (v'_q)^2} \leq 1$$

See `documentation/theory/current-loop-pi.md` — *Output Voltage Limit* — for the full derivation and the interaction with integral anti-windup.

The limit is applied **once**, to the sum of the PI output and the decoupling feedforward:

$$v'[k] = \Pi(\tilde{v}'_{PI}[k] + v'_{ff}[k])$$

Limiting the PI output on its own first and the sum again afterwards reserves no headroom for the feedforward, so decoupling degrades exactly at the high speeds where the cross-coupling and back-EMF terms it cancels are largest. The share committed back to the PI's integrator is the

applied vector minus the feedforward, so the feedforward is never integrated as if the loop had asked for it.

## Numerical Properties

| Property                      | Value                                          |
|-------------------------------|------------------------------------------------|
| ISR cost                      | $\sim 10$ MACs (PI + feedforward additions)    |
| Settling time                 | $\sim 1/\omega_{bw}$                           |
| Robustness to $R_s/L_s$ error | Low — feedforward is model-dependent           |
| Requires $\psi_f$             | Yes (q-axis back-EMF cancellation)             |
| Requires RLS convergence      | Partial ( $L_s$ for feedforward; $R_s$ for PI) |
| Tuning knobs                  | 1 ( $\omega_{bw}$ )                            |

## Limitations & Assumptions

- Feedforward accuracy is proportional to  $L_s$  accuracy: 20%  $L_s$  error leaves 20% residual coupling.
- Requires  $\psi_f$  for back-EMF cancellation on the q-axis.  $\psi_f$  is set at alignment and not updated online.
- Assumes balanced three-phase operation ( $i_a + i_b + i_c = 0$ ).
- Requires accurate electrical angle  $\theta_e$ . Angle errors couple directly into the dq frame.
- Output subject to the circular SVM constraint — see *Output Voltage Limit* above.

## References

1. Krishnan, R. — *Permanent Magnet Synchronous and Brushless DC Motor Drives*, CRC Press, 2010.
2. Åström, K.J. & Wittenmark, B. — *Computer-Controlled Systems*, 3rd ed., Prentice Hall, 1997.

| Field     | Value                       |
|-----------|-----------------------------|
| Title     | Current Loop — A2: Deadbeat |
| Type      | theory                      |
| Status    | draft                       |
| Version   | 1.0.0                       |
| Component | current-loop-deadbeat       |
| Date      | 2026-08-31                  |

**Theory document:** Explains the mathematical and engineering principles behind a component or algorithm. This document is descriptive — it records the *why* and *how*

at a scientific level, independent of any specific implementation. Equations use KaTeX (*inline* and

*block*

). Block diagrams use Mermaid or ASCII art.

---

## Overview

A2 (Deadbeat) inverts the discrete current plant model to compute the exact voltage that drives the current to its reference in one sampling step — the minimum physically achievable settling time (50  $\mu$ s at 20 kHz). This maximises servo stiffness: the torque response is instantaneous relative to any outer loop, so load disturbances are rejected before position error can accumulate. A two-step variant reduces noise amplification at the cost of geometric (non-deadbeat) settling.

Operates exclusively in the **20 kHz FOC ISR**. Requires tight electrical RLS convergence ( $L_s$ ,  $R_s$ ) and the A1 feedforward for decoupling at speed.

---

## Prerequisites

| Symbol         | Meaning                         | Unit |
|----------------|---------------------------------|------|
| $A_d^i, B_d^i$ | Discrete current plant matrices | —    |
| $i^*$          | Current reference (d or q axis) | A    |
| $i[k]$         | Measured current                | A    |
| $V_{dc}$       | DC bus voltage                  | V    |

See `documentation/theory/foc-plant-models.md` for ZOH derivations of  $A_d^i$ ,  $B_d^i$ . See `documentation/theory/current-loop-decoupled-pid.md` for the A1 feedforward used alongside A2.

---

## Mathematical Foundation

All current controllers operate on the **decoupled per-axis RL plant** derived in `documentation/theory/foc-plant-models.md` §1:

$$i[k+1] = A_d^i \cdot i[k] + B_d^i \cdot v'[k]$$

A2 inverts this plant model directly and therefore applies the **same A1 feedforward** to its inversion result — an exact plant inversion is only exact once the coupling it does not model has been cancelled.

---

## A2 — Deadbeat Current Control

**One-step control law:** Setting  $i[k+1] = i^*$  and solving for  $v'$ :

$$v'[k] = \frac{i^* - A_d^i i[k]}{B_d^i}$$

The result is clamped to  $[-1, 1]$  (normalised). When the step demand exceeds available voltage, settling takes more than one sample but remains optimal for the available headroom.

**Two-step variant** (noise robustness): The one-step law amplifies measurement noise by  $1/B_d^i = L_s/T_s^i$ , which is large for small inductances. The two-step law solves for the **minimum-norm input sequence** over a two-sample horizon and applies its first element, with the reference gain corrected so that the closed loop retains unity DC gain:

$$v'[k] = \frac{((A_d^i)^2 - A_d^i + 1) i^* - (A_d^i)^3 i[k]}{B_d^i ((A_d^i)^2 + 1)}$$

This reduces the state gain — and hence noise amplification — by roughly half.

**Receding-horizon behaviour:** because only the first element of the sequence is applied and the solution is recomputed every sample, the two-step law does *not* settle in exactly two samples. The closed-loop pole is  $A_d^i / ((A_d^i)^2 + 1)$ , giving a geometric rather than deadbeat response. Tracking is nonetheless exact in steady state: the reference gain is obtained by solving

$$B_d^i g_{ref} = 1 - A_d^i + B_d^i g_{state}$$

which forces unity DC gain. Taking the raw minimum-norm reference gain instead leaves a static error of  $A_d^i / ((A_d^i)^2 - A_d^i + 1)$  — around 1% at  $A_d^i = 0.90$  and 20% at 0.61 — which is worst in the small- $L_s$  regime the variant targets.

The one-step law is unaffected — it places the closed-loop pole at the origin and tracks exactly.

**Model accuracy requirement:** A 10%  $L_s$  error leaves a 10% residual error after one step. Deadbeat is the natural progression after Decoupled PID has validated RLS convergence.

**Decoupling:** the inversion above solves the decoupled plant, so the A1 feedforward terms are added to  $v'$  before the modulation-circle limit. Without them  $\omega_e L_s i_q$  enters as an unmodelled disturbance and the one-step settling property is lost at speed.

**Normalisation:**  $v'[k]$  (physical volts) normalised identically to the PI output:  $v'_{norm}[k] = v'[k] \cdot \sqrt{3}/V_{dc}$ .

**Tuning:** None — fully determined by  $A_d^i, B_d^i$  from RLS. Only design choice: one-step vs. two-step variant.

## Output Voltage Limit

The output voltage vector is subject to the circular SVM constraint:

$$\sqrt{(v'_d)^2 + (v'_q)^2} \leq 1$$

See `documentation/theory/current-loop-pi.md` — *Output Voltage Limit* — for the full derivation. Deadbeat has no integral action, so it is unaffected by windup at the voltage limit.

---

## Numerical Properties

| Property                  | Value                                     |
|---------------------------|-------------------------------------------|
| ISR cost                  | ~4 MACs (cheapest advanced controller)    |
| Settling time             | 1 sample (one-step); geometric (two-step) |
| Robustness to Rs/Ls error | Low — exact model required                |
| Requires $\psi_f$         | No (but A1 feedforward does)              |
| Requires RLS convergence  | Yes — tight convergence required          |
| Tuning knobs              | 0 (variant choice only)                   |

---



---

## Limitations & Assumptions

- Most sensitive to  $L_s$  error of all current controllers. Do not activate until electrical RLS has converged (typically after the first few seconds of operation under load).
  - The two-step variant halves noise amplification and still tracks exactly, but gives up dead-beat settling for a geometric response with pole  $A_d^i / ((A_d^i)^2 + 1)$ . Prefer it for noisy current measurement or small  $L_s$ ; prefer one-step for maximum servo stiffness.
  - Assumes balanced three-phase operation ( $i_a + i_b + i_c = 0$ ).
  - Requires accurate electrical angle  $\theta_e$ . Angle errors couple directly into the dq frame.
  - Output subject to the circular SVM constraint — see *Output Voltage Limit* above.
- 

## References

1. Krishnan, R. — *Permanent Magnet Synchronous and Brushless DC Motor Drives*, CRC Press, 2010.
2. Åström, K.J. & Wittenmark, B. — *Computer-Controlled Systems*, 3rd ed., Prentice Hall, 1997.

---

| Field | Value                           |
|-------|---------------------------------|
| Title | Current Loop — A3: Sliding-Mode |

---

---

| Field     | Value                     |
|-----------|---------------------------|
| Type      | theory                    |
| Status    | draft                     |
| Version   | 1.0.0                     |
| Component | current-loop-sliding-mode |
| Date      | 2026-08-31                |

---

**Theory document:** Explains the mathematical and engineering principles behind a component or algorithm. This document is descriptive — it records the *why* and *how* at a scientific level, independent of any specific implementation. Equations use KaTeX (*inline* and

*block*

). Block diagrams use Mermaid or ASCII art.

---

## Overview

A3 (Sliding-Mode) is robust to model uncertainty — the most critical early in operation before RLS has converged and during thermal transients that shift  $R_s$ . The SMC drives the current error onto a stable sliding surface regardless of parameter mismatch, within a defined gain margin. Unlike A1 and A2, A3 deliberately omits the A1 feedforward and treats residual coupling and noise as a bounded disturbance covered by the switching gain  $K_{sw}$ .

Operates exclusively in the **20 kHz FOC ISR**. Partial RLS dependency —  $A_d^i$ ,  $B_d^i$  are used but the controller remains stable over a wide range of mismatch.

---

## Prerequisites

---

| Symbol         | Meaning                           | Unit |
|----------------|-----------------------------------|------|
| $A_d^i, B_d^i$ | Discrete current plant matrices   | —    |
| $\phi$         | Sliding-mode boundary layer width | A    |
| $K_{sw}$       | Sliding-mode switching gain       | A    |

---

See `documentation/theory/foc-plant-models.md` for ZOH derivations.

---

## Mathematical Foundation

All current controllers operate on the **decoupled per-axis RL plant** derived in `documentation/theory/foc-plant-models.md §1`:

$$i[k+1] = A_d^i \cdot i[k] + B_d^i \cdot v'[k]$$

A3 deliberately omits the A1 feedforward and treats the residual coupling and noise as a bounded disturbance covered by the switching gain  $K_{sw}$ ; that robustness is the reason to select it before RLS has converged.

---

## A3 — Sliding-Mode Current Control

**Sliding surface:** For each axis independently, defined on the tracking error  $e = i - i^*$ :

$$s_d = i_d - i_d^*, \quad s_q = i_q - i_q^*$$

The surface  $s = 0$  is the desired zero-error manifold.

**Equivalent control and the reaching law:** The error propagates as  $e[k+1] = A_d^i e[k] + B_d^i u[k] + (A_d^i - 1)i^*[k]$ . The law chooses its voltage so that the error obeys the exponential reaching law

$$e[k+1] = \lambda e[k] - K_{sw} \text{sat}\left(\frac{e[k]}{\phi}\right), \quad \lambda = e^{-\omega_{bw} T_s}$$

which needs

$$u[k] = -\frac{A_d^i - \lambda}{B_d^i} e[k] - \frac{K_{sw}}{B_d^i} \text{sat}\left(\frac{e[k]}{\phi}\right) + \frac{1 - A_d^i}{B_d^i} i^*[k]$$

where  $\text{sat}(x) = \text{clamp}(x, -1, 1)$ . Because  $B_d^i = (1 - A_d^i)/R_s$ , the last term is exactly  $R_s i^*[k]$  — the resistive drop the reference current requires. All terms carry a leading minus sign on the error because it is measured as  $i - i^*$ : a current below its reference gives  $e < 0$  and must command a *positive* voltage. The boundary layer  $\phi > 0$  replaces the discontinuous sign function with saturation to prevent chattering.

The toolbox's `SlidingModeControl` computes its equivalent term as  $-(SB)^{-1}SAe$ , i.e. it cancels whatever the state matrix it is given predicts. It is handed  $A_d^i - \lambda$ , not  $A_d^i$ . Handing it  $A_d^i$  — the form this law shipped with — makes the equivalent term deadbeat: it drives the next error to zero on its own, and the switching term then pushes the error  $K_{sw}$  past zero every time it leaves the boundary layer. From rest a step of  $\phi$  overshoot by  $K_{sw}/\phi$ , 40 % with the shipped defaults, and oscillated with ratio  $-K_{sw}/\phi$  after that; the SIL measured 46–50 % on a 0.5 A step.

**Parameter choices:** -  $\lambda$  follows the current-loop bandwidth, so large steps approach the surface at the loop's own rate rather than at the fixed  $K_{sw}$  per sample. -  $K_{sw}$ : must exceed the worst-case disturbance. Because  $u_{sw}$  is obtained by dividing by  $B_d^i$ , the gain carries the same unit as the sliding surface (A). Sizing it so the switching term commands at most 30% of the maximum phase voltage  $V_{dc}/\sqrt{3}$  gives the starting point:  $K_{sw} = 0.3 \cdot B_d^i \cdot V_{dc}/\sqrt{3}$ . -  $\phi$ : boundary layer width (A). Typical: 0.1–0.5 A. Smaller gives tighter tracking but more high-frequency actuation.

**Discrete stability constraint** — this bounds the parameters together and is not optional. Inside the boundary layer the error obeys  $e[k+1] = (\lambda - K_{sw}/\phi) e[k]$ . It contracts monotonically, without crossing the surface, when  $0 \leq \lambda - K_{sw}/\phi < 1$ , i.e.  $K_{sw} \leq \lambda \phi$ ; it still converges, alternating in sign, while  $K_{sw} < (1 + \lambda) \phi$ . The shipped defaults are  $K_{sw} = 0.2$  A and  $\phi = 0.5$  A; at the default bandwidth  $\lambda \approx 0.73$ , so the ratio 0.4 leaves a monotonic approach.

**Robustness:** Insensitive to  $R_s/L_s$  mismatch as long as mismatch is bounded by  $K_{sw}$ . Covers  $\pm 50\%$  thermal variation in  $R_s$  and the initial RLS convergence transient.

**Relation to toolbox:** `SlidingModeControl<float,1,1>` implements equivalent + switching with boundary-layer saturation on the error alone; the equilibrium term  $R_s i^*$  is added by `SlidingModeCurrentController`. Plant matrices  $A_d^i, B_d^i$  are constructed from RLS estimates.

---

## Output Voltage Limit

The output voltage vector is subject to the circular SVM constraint:

$$\sqrt{(v'_d)^2 + (v'_q)^2} \leq 1$$

See `documentation/theory/current-loop-pi.md` — *Output Voltage Limit* — for the full derivation. A3 has no integral action, so it is unaffected by windup at the voltage limit.

---

## Numerical Properties

| Property                      | Value                                              |
|-------------------------------|----------------------------------------------------|
| ISR cost                      | $\sim 12$ MACs (equivalent + switching, 2 axes)    |
| Settling time                 | $\sim 1/\omega_{bw}$ (boundary-layer limited)      |
| Robustness to $R_s/L_s$ error | High — gain-bounded; covers $\pm 50\%$ $R_s$ drift |
| Requires $\psi_f$             | No                                                 |
| Requires RLS convergence      | Partial — stable with datasheet values             |
| Tuning knobs                  | $2 (K_{sw}, \phi)$                                 |

---

## Limitations & Assumptions

- Boundary layer  $\phi$  introduces a steady-state band proportional to  $\phi/K_{sw}$ . Set  $\phi$  small enough that this band is below ADC resolution.
- $K_{sw}$  must exceed the worst-case disturbance; undersizing prevents reaching the sliding surface.
- **Stability constraint:**  $K_{sw} < \phi$  is mandatory for discrete convergence.
- Assumes balanced three-phase operation ( $i_a + i_b + i_c = 0$ ).
- Requires accurate electrical angle  $\theta_e$ . Angle errors couple directly into the dq frame.

- Output subject to the circular SVM constraint — see *Output Voltage Limit* above.

---

## References

1. Krishnan, R. — *Permanent Magnet Synchronous and Brushless DC Motor Drives*, CRC Press, 2010.
2. Utkin, V. — *Sliding Modes in Control and Optimization*, Springer, 1992. (Equivalent control; boundary-layer chattering suppression.)
3. Åström, K.J. & Wittenmark, B. — *Computer-Controlled Systems*, 3rd ed., Prentice Hall, 1997.

| Field     | Value                      |
|-----------|----------------------------|
| Title     | Speed Loop — PI (Baseline) |
| Type      | theory                     |
| Status    | draft                      |
| Version   | 1.0.0                      |
| Component | speed-loop-pi              |
| Date      | 2026-08-31                 |

**Theory document:** Explains the mathematical and engineering principles behind a component or algorithm. This document is descriptive — it records the *why* and *how* at a scientific level, independent of any specific implementation. Equations use KaTeX (*inline* and

*block*

). Block diagrams use Mermaid or ASCII art.

---

## Overview

The baseline speed controller is a discrete incremental PI regulator that outputs a  $q$ -axis current reference  $i_q^*$  for the inner current loop. Pole-zero cancellation against the mechanical speed plant places the closed-loop speed bandwidth at  $\omega_{bw}^s$  directly. No mechanical RLS convergence is required — datasheet values for  $J$  and  $B_f$  are sufficient, and the integral term corrects steady-state error from model mismatch.

Operates exclusively in the **1 kHz outer handler**.

---

## Prerequisites

| Symbol | Meaning                 | Unit                         |
|--------|-------------------------|------------------------------|
| $J$    | Rotor moment of inertia | $\text{kg} \cdot \text{m}^2$ |

| Symbol        | Meaning                                    | Unit                                                |
|---------------|--------------------------------------------|-----------------------------------------------------|
| $B_f$         | Viscous friction coefficient               | $\text{N} \cdot \text{m} \cdot \text{s}/\text{rad}$ |
| $K_t$         | Torque constant = $\frac{3}{2}p\psi_f$     | $\text{N} \cdot \text{m}/\text{A}$                  |
| $T_s^o$       | Outer loop sample period (= 1 ms)          | s                                                   |
| $\omega_{bw}$ | Desired closed-loop speed bandwidth        | rad/s                                               |
| $I_{q,max}$   | Maximum $q$ -axis current (hardware limit) | A                                                   |

See `documentation/theory/foc-plant-models.md` for all base symbols and ZOH derivations.

## Mathematical Foundation

### Plant Model

Treating the current loop as ideal ( $i_q \approx i_q^*$ ), the mechanical speed dynamics are:

$$J \frac{d\omega_m}{dt} = K_t i_q^* - B_f \omega_m - T_L$$

This is a first-order plant with a single real pole. The continuous-time transfer function from  $i_q^*$  to  $\omega_m$  (ignoring load disturbance  $T_L$ ) is:

$$G_\omega(s) = \frac{K_t}{Js + B_f} = \frac{K_t/J}{s + B_f/J}$$

ZOH-discretized at  $T_s^o$  (from `documentation/theory/foc-plant-models.md` §2):

$$\omega_m[k+1] = A_d^o \omega_m[k] + B_d^o u[k], \quad A_d^o = e^{-B_f T_s^o/J}, \quad B_d^o = \frac{K_t T_s^o}{J}$$

### PI Gain Design — Pole-Zero Cancellation

The mechanical plant has a single pole at  $s = -B_f/J$ . A PI controller:

$$C(s) = K_p \cdot \frac{s + K_i/K_p}{s}$$

achieves pole-zero cancellation by placing the PI zero at the plant pole:

$$\frac{K_i}{K_p} = \frac{B_f}{J}$$

The open-loop gain is then  $K_p K_t / (Js)$ , giving a first-order closed loop with bandwidth  $\omega_{bw}$ . Solving for gains:

$$K_p = \frac{J \omega_{bw}}{K_t}, \quad K_i = \frac{B_f \omega_{bw}}{K_t}$$

## What the implementation uses

The derivation above is the textbook one; `PidSpeedController::ApplyGains` departs from it in two ways, and both matter when reading the code against this chapter.

$$K_p = \frac{2 J \omega_{bw}}{K_t}, \quad \frac{K_i}{K_p} = \max\left(\frac{B_f}{J}, \frac{\omega_{bw}}{4}\right)$$

**The factor 2 on  $K_p$**  is not produced by the derivation, which places crossover at  $\omega_{bw}$  exactly. It is an empirical margin carried by the implementation; this chapter records it rather than justifying it, because no derivation here yields it.

**The integral zero is floored** at  $\omega_{bw}/4$ , so the pole-zero cancellation this section is named after happens only when  $B_f/J$  is already the larger of the two. For the identified parameters of a small PMSM it is not —  $B_f/J$  is typically well under  $\omega_{bw}/4$  — so in practice the floor sets the zero and the plant pole is *not* cancelled. The floor exists so that  $K_i$  stays non-zero when the mechanical identification returns  $B_f = 0$ , which would otherwise leave a pure proportional law with a standing error.

**Where the floor sits decides load rejection.** With the factor 2 on  $K_p$  the loop closes as  $s^2 + 2\omega_{bw}s + 2\omega_{bw}z$ , where  $z$  is the integral zero. Its slow root sits near  $z$  and is what works a load torque off: the speed error after a torque step  $\tau$  decays as  $\frac{\tau}{J(p_2 - p_1)} (e^{p_1 t} - e^{p_2 t})$ . A floor of  $\omega_{bw}/10$  put that root near  $\omega_{bw}/10$  — about 10 rad/s at a 100 rad/s bandwidth, a time constant near 100 ms — so a 0.5 N·m step on the reference rotor (7.06  $\mu\text{kg} \cdot \text{m}^2$ ) was still 10.4 rad/s off a 20 rad/s setpoint 0.4 s later and needed nearly half a second to return inside 3 rad/s. At  $\omega_{bw}/4$  the slow root moves to about  $0.29\omega_{bw}$  and the same recovery takes under 200 ms. The zero also enters the reference path, so it trades against step overshoot: at  $\omega_{bw}/4$  it nearly cancels the slow root and a step from rest overshoots by under 10 % on the same rotor; at  $\omega_{bw}/2$  the overshoot passes 15 %.

Both gains are then divided by  $I_{q,max}$  before they reach the PID, and its output is multiplied by  $I_{q,max}$  again, so the controller works in normalised units and the scaling cancels.

**Practical limit:** speed bandwidth must satisfy the cascade separation principle:  $\omega_{bw}^s \leq \omega_{bw}^i/10$ , where  $\omega_{bw}^i$  is the current-loop bandwidth. Typical:  $\omega_{bw}^s = 100\text{--}500$  rad/s at a 1 kHz control rate.

## Incremental Form and Anti-Windup

The implementation uses the incremental PI to avoid initialisation issues:

$$\Delta u[k] = K_p (e[k] - e[k-1]) + K_i T_s^o e[k]$$

The  $T_s^o$  factor is folded into the gain the implementation stores, so the `ki` in the code is the  $K_i T_s^o$  of this expression, not the  $K_i$  of the continuous design above.

$$u[k] = \text{clamp}(u[k-1] + \Delta u[k], -I_{q,max}, I_{q,max})$$

where  $e[k] = \omega_m^*[k] - \omega_m[k]$ . Clamping to  $\pm I_{q,max}$  limits the current demand to the hardware envelope and provides inherent anti-windup — no separate back-calculation is needed.

## Numerical Properties

| Property                   | Value                                 |
|----------------------------|---------------------------------------|
| Ops per 1 kHz cycle        | ~6 MACs                               |
| Steady-state speed error   | Zero (integral action)                |
| Load disturbance rejection | Integral quality — bandwidth-limited  |
| Tracking vs. stiffness     | Coupled (single DOF)                  |
| Requires mechanical RLS    | No — datasheet $J$ , $B_f$ sufficient |
| Tuning knobs               | 1 ( $\omega_{bw}$ )                   |

## Limitations & Assumptions

- Treats the current loop as ideal ( $i_q \approx i_q^*$ ). Valid when the current-loop bandwidth is at least  $10\times$  the speed bandwidth.
- Pole-zero cancellation accuracy depends on knowledge of  $B_f/J$ . Model mismatch shifts the cancellation but integral action removes any resulting steady-state error.
- Single-DOF structure — cannot independently shape command tracking and disturbance rejection. Use S3 (Two-DOF) when both are required simultaneously.
- Speed estimate must be smooth. A noisy encoder derivative degrades this controller equally to all speed controllers; an alpha-beta filter or Kalman smoother is recommended upstream.

## References

1. Krishnan, R. — *Permanent Magnet Synchronous and Brushless DC Motor Drives*, CRC Press, 2010.
2. Åström, K.J. & Wittenmark, B. — *Computer-Controlled Systems*, 3rd ed., Prentice Hall, 1997.
3. Åström, K.J. & Häggglund, T. — *Advanced PID Control*, ISA, 2006.

| Field     | Value                |
|-----------|----------------------|
| Title     | Speed Loop — S1: LQI |
| Type      | theory               |
| Status    | draft                |
| Version   | 1.0.0                |
| Component | speed-loop-lqi       |
| Date      | 2026-08-31           |

**Theory document:** Explains the mathematical and engineering principles behind a component or algorithm. This document is descriptive — it records the *why* and *how* at a scientific level, independent of any specific implementation. Equations use KaTeX (*inline* and

*block*

). Block diagrams use Mermaid or ASCII art.

---

## Overview

S1 (LQI — Linear-Quadratic-Integral) state-feedback computes gains that minimise an explicit cost function directly from the mechanical RLS estimates  $\hat{J}$  and  $\hat{B}_f$ . A PI speed controller is heuristically tuned and its gains have no direct physical meaning. LQI provides systematic tuning: the weights  $q_\omega$ ,  $q_I$ , and  $R$  have transparent physical interpretations. The integral augmentation guarantees zero steady-state error to a constant speed reference.

Operates exclusively in the **1 kHz outer handler**. Requires mechanical RLS convergence ( $J$ ,  $B_f$ ).

---

## Prerequisites

| Symbol         | Meaning                            | Unit |
|----------------|------------------------------------|------|
| $A_d^o, B_d^o$ | Discrete speed plant matrices      | —    |
| $Q, R$         | LQR weighting matrices             | —    |
| $P$            | DARE solution (optimal cost-to-go) | —    |
| $I_{q,max}$    | Maximum $q$ -axis current          | A    |

See `documentation/theory/foc-plant-models.md §2` for the speed plant derivation.

---

## Mathematical Foundation

All speed controllers operate on the **discrete mechanical speed plant** from `documentation/theory/foc-plant-models.md §2`. Treating the current loop as ideal ( $i_q \approx i_q^*$ ), the speed dynamics under control input  $u = i_q^*$  are:

$$\omega_m[k+1] = A_d^o \cdot \omega_m[k] + B_d^o \cdot u[k], \quad A_d^o \approx 1 - \frac{B_f T_s^o}{J}, \quad B_d^o = \frac{K_t T_s^o}{J}$$


---

## S1 — LQI Speed Control

**Integral augmentation:** Augment the discrete speed plant with an integrator state  $x_I$  that accumulates the speed error:

$$x_I[k+1] = x_I[k] + (\omega_m^*[k] - \omega_m[k])$$

The augmented plant matrices:

$$\mathbf{A}_{aug} = \begin{pmatrix} A_d^o & 0 \\ -1 & 1 \end{pmatrix}, \quad \mathbf{B}_{aug} = \begin{pmatrix} B_d^o \\ 0 \end{pmatrix}$$

**Gain design via DARE:**

$$P = A_{aug}^T P A_{aug} - A_{aug}^T P B_{aug} (R + B_{aug}^T P B_{aug})^{-1} B_{aug}^T P A_{aug} + Q$$

$$\mathbf{K} = [K_\omega, K_I] = (R + B_{aug}^T P B_{aug})^{-1} B_{aug}^T P A_{aug}$$

**Control law** (run-time: two multiplications and one addition):

$$u[k] = -K_\omega (\omega_m[k] - \omega_m^*[k]) - K_I x_I[k]$$

Structurally identical to a discrete PI but with gains derived from physical parameters and an explicit performance cost.

**Tuning** ( $Q = \text{diag}(q_\omega, q_I)$ , scalar  $R$ ): -  $q_\omega$ : speed error penalty — tighter tracking, higher control effort. -  $q_I$ : integral state penalty — faster integrator wind-down, less overshoot. -  $R$ : input penalty — directly caps the  $i_q^*$  command. - Starting point:  $q_\omega = 1$ ,  $q_I = 0.1$ . -  $R$  must scale with the plant's own normalized input gain squared,  $b^2 = (K_t T_s^o I_{q,max} / J)^2$ , not a flat constant: a fixed  $R$  is only the right order of magnitude when  $b$  is  $O(1)$ , but  $b$  varies by orders of magnitude across motors (small-inertia, high-torque-constant motors give a large  $b$ ). `LqiSpeedController` derives it as `R = NormalizedEffortWeight(bandwidth, samplingFrequency) * b^2`, reusing the same bandwidth-to-effort mapping the position-loop LQI/LQR controllers use (`core/foc/speed_loop/SpeedPlantModel.hpp`). Leaving  $R$  unscaled by  $b^2$  produced an effectively deadbeat gain with no margin against the always-present loop delay of a real (non-ideal) current loop — the root cause of the LQI speed law's former overshoot and poor disturbance rejection (`documentation/design/software-in-the-loop.md` Part I).

**Gain computation** runs once at configuration time off the hot path using the **DARE** solver from the numerical toolbox. The 1 kHz handler executes only the dot product.

**Numerical conditioning:** the DARE is solved with the control input expressed per-unit of  $I_{q,max}$  and with the integral state accumulating the raw error sum rather than  $e \cdot T_s^o$ . Both substitutions leave  $\mathbf{K}$  unchanged — only the ratio  $Q/R$  and the state scaling matter — but they keep the entries of  $P$  of order unity, which the single-precision fixed-point iteration needs in order to reach its absolute convergence threshold.

**Anti-windup:** the integral state is frozen (conditional integration) on any sample where the unsaturated command falls outside  $\pm I_{q,max}$ , and resumes as soon as the command re-enters the envelope.

---

## Numerical Properties

---

| Property                   | Value                                             |
|----------------------------|---------------------------------------------------|
| Ops per 1 kHz cycle        | 2 MACs (dot product $\mathbf{K} \cdot x_{aug}$ )  |
| Steady-state speed error   | Zero (integral augmentation)                      |
| Load disturbance rejection | PI quality (integral-based)                       |
| Tracking vs. stiffness     | Coupled (single DOF)                              |
| DARE timing                | $< 2 \mu\text{s}$ at 120 MHz, once at config time |
| Requires mechanical RLS    | Yes ( $J, B_f$ )                                  |
| Tuning knobs               | 3 ( $q_\omega, q_I, R$ )                          |

---



---

## Limitations & Assumptions

- Treats the current loop as ideal ( $i_q \approx i_q^*$ ). Valid when current-loop bandwidth is at least  $10\times$  the speed-loop bandwidth.
  - Integral wind-up: the integral state  $x_I$  must be clamped when  $i_q^*$  saturates. Anti-windup is required.
  - DARE conditioning: if  $B_d^o = K_t T_s^o / J \approx 0$  (extremely large inertia), the DARE may be ill-conditioned. Verify the controllability Gramian before solving.
  - Speed estimate must be smooth. A noisy encoder derivative degrades all speed controllers equally.
- 

## References

1. Krishnan, R. — *Permanent Magnet Synchronous and Brushless DC Motor Drives*, CRC Press, 2010.
2. Åström, K.J. & Wittenmark, B. — *Computer-Controlled Systems*, 3rd ed., Prentice Hall, 1997.

---

| Field     | Value                 |
|-----------|-----------------------|
| Title     | Speed Loop — S2: ADRC |
| Type      | theory                |
| Status    | draft                 |
| Version   | 1.0.0                 |
| Component | speed-loop-adrc       |
| Date      | 2026-08-31            |

---

**Theory document:** Explains the mathematical and engineering principles behind a component or algorithm. This document is descriptive — it records the *why* and *how* at a scientific level, independent of any specific implementation. Equations use KaTeX (*inline* and

*block*

). Block diagrams use Mermaid or ASCII art.

---

## Overview

S2 (Active Disturbance Rejection Control) makes the load torque disturbance explicit: an Extended State Observer (ESO) estimates the total disturbance (load torque + model error) in real time and the control law cancels it. This gives dramatically faster load-step rejection than integral action alone, with only two tuning parameters ( $\omega_c$ ,  $\omega_o$ ). LQI handles the disturbance only implicitly through its integral state; ADRC cancels it before the control law sees it.

Operates exclusively in the **1 kHz outer handler**. Requires  $K_t$  and  $J$  from mechanical RLS.

## Prerequisites

| Symbol             | Meaning                       | Unit  |
|--------------------|-------------------------------|-------|
| $A_d^o, B_d^o$     | Discrete speed plant matrices | —     |
| $\hat{x}, \hat{f}$ | Observer state estimates      | —     |
| $\omega_o$         | ESO bandwidth                 | rad/s |
| $\omega_c$         | Control bandwidth             | rad/s |
| $b_0$              | ADRC plant gain = $K_t/J$     | —     |

See `documentation/theory/foc-plant-models.md §2` for the speed plant derivation.

## Mathematical Foundation

All speed controllers operate on the **discrete mechanical speed plant** from `documentation/theory/foc-plant-models.md §2`:

$$\omega_m[k+1] = A_d^o \cdot \omega_m[k] + B_d^o \cdot u[k]$$

The unknown load torque  $T_L/J$  enters as a state perturbation. S2 estimates it in real time with an Extended State Observer and cancels it before the control law sees it.

## S2 — Active Disturbance Rejection Speed Control (ADRC)

**Extended plant:** Lump total disturbance  $f$  (absorbing both  $T_L$  and model error) into the state:

$$\frac{d\omega_m}{dt} = b_0 u + f, \quad b_0 = \frac{K_t}{J}$$

**ESO** (second-order, bandwidth  $\omega_o$ , binomial gain placement):

$$L = [\beta_1, \beta_2]^T, \quad \beta_1 = 2\omega_o, \quad \beta_2 = \omega_o^2$$

Discrete update at  $T_s^o$ :

$$\begin{aligned} e_{obs}[k] &= \omega_m[k] - \hat{\omega}_m[k] \\ \hat{\omega}_m[k+1] &= \hat{\omega}_m[k] + T_s^o (\beta_1 e_{obs}[k] + \hat{f}[k] + b_0 u[k]) \\ \hat{f}[k+1] &= \hat{f}[k] + T_s^o \cdot \beta_2 e_{obs}[k] \end{aligned}$$

**Control law:**

$$u_0[k] = \omega_c (\omega_m^*[k] - \hat{\omega}_m[k])$$

$$u[k] = \frac{u_0[k] - \hat{f}[k]}{b_0}$$

**Tuning (two parameters):** -  $\omega_c$ : desired closed-loop speed bandwidth (rad/s). Sets command-tracking rise time. -  $\omega_o$ : ESO bandwidth. Rule of thumb:  $\omega_o = 3\text{--}10 \times \omega_c$ . -  $b_0 = K_t/J$  from RLS. Tolerates  $\pm 50\%$  error without stability loss.

**Performance:** Under a step load torque, the ADRC suppresses the speed dip in approximately  $1/\omega_o$  seconds — typically an order of magnitude faster than integral action alone.

**Relation to toolbox:** `ActiveDisturbanceRejectionControl<float,1>` implements this ESO and control law — the template parameter is the plant order, and the extended state carrying  $\hat{f}$  is added on top of it. `Compute( $\omega_{ref}$ ,  $\omega_{meas}$ )` is called once per 1 kHz handler cycle.

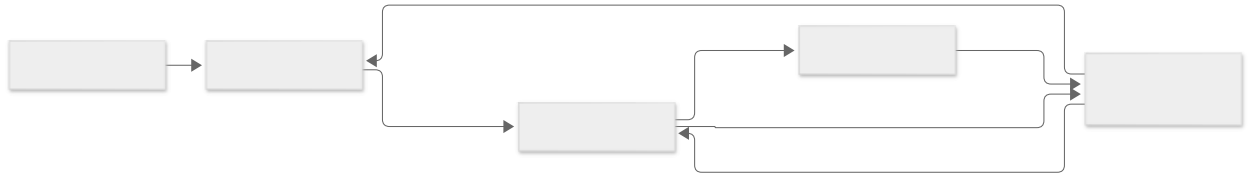


Figure 9: Block diagram

## Numerical Properties

| Property                   | Value                                              |
|----------------------------|----------------------------------------------------|
| Ops per 1 kHz cycle        | 6 MACs (ESO update + control law)                  |
| Steady-state speed error   | Zero (ESO cancels constant disturbances)           |
| Load disturbance rejection | Explicit cancellation — $\sim 1/\omega_o$ response |
| Tracking vs. stiffness     | Coupled (single DOF)                               |
| Requires mechanical RLS    | Yes ( $K_t$ , $J$ )                                |

---

| Property     | Value                      |
|--------------|----------------------------|
| Tuning knobs | 2 ( $\omega_c, \omega_o$ ) |

---

## Limitations & Assumptions

- Treats the current loop as ideal ( $i_q \approx i_q^*$ ). Valid when current-loop bandwidth is at least  $10\times$  the speed-loop bandwidth.
  - ESO estimates  $\hat{f}$  with a lag proportional to  $1/\omega_o$ . Disturbances faster than  $\omega_o$  are not fully cancelled.
  - High  $\omega_o$  amplifies speed measurement noise. Use a velocity smoother upstream if needed.
  - The ESO is discretized with forward Euler, whose poles leave the unit circle for  $\omega_o T_s^o > 0.83$ . The toolbox enforces the conservative precondition  $\omega_o T_s^o < 0.5/n$ ; the configured observer bandwidth is clamped to  $\omega_o \leq 0.4/T_s^o$  to stay inside it, which caps the usable  $\omega_o/\omega_c$  ratio at 1 kHz.
  - The clipped  $i_q^*$  is fed back into the ESO each cycle, so a sustained current clip does not bias  $\hat{f}$ .
  - Speed estimate must be smooth. A noisy encoder derivative degrades all speed controllers equally.
- 

## References

1. Krishnan, R. — *Permanent Magnet Synchronous and Brushless DC Motor Drives*, CRC Press, 2010.
2. Han, J. — “From PID to Active Disturbance Rejection Control”, *IEEE Trans. Ind. Electron.*, 56(3):900–906, 2009.
3. Gao, Z. — “Scaling and Bandwidth-Parameterization Based Controller Tuning”, *Proc. ACC*, 2003.

---

| Field     | Value                    |
|-----------|--------------------------|
| Title     | Speed Loop — S3: Two-DOF |
| Type      | theory                   |
| Status    | draft                    |
| Version   | 1.0.0                    |
| Component | speed-loop-two-dof       |
| Date      | 2026-08-31               |

---

**Theory document:** Explains the mathematical and engineering principles behind a component or algorithm. This document is descriptive — it records the *why* and *how* at a scientific level, independent of any specific implementation. Equations use KaTeX (*inline* and

*block*

). Block diagrams use Mermaid or ASCII art.

---

## Overview

S3 (Two-DOF speed control) resolves a fundamental limitation of single-DOF speed controllers: tightening the feedback loop for faster setpoint following degrades load stiffness and vice versa. A reference pre-filter  $F(z)$  in the forward path shapes command tracking independently from the feedback loop designed for disturbance rejection. This is particularly valuable for servo speed control where both fast tracking and high stiffness are required simultaneously.

Operates exclusively in the **1 kHz outer handler**. No mechanical RLS required.

---

## Prerequisites

| Symbol         | Meaning                          | Unit |
|----------------|----------------------------------|------|
| $A_d^o, B_d^o$ | Discrete speed plant matrices    | —    |
| $\tau_{ff}$    | Two-DOF pre-filter time constant | s    |
| $K_p, K_i$     | PI feedback controller gains     | —    |

See `documentation/theory/foc-plant-models.md` §2 for the speed plant derivation. See `documentation/theory/speed-loop-pi.md` for PI gain design.

---

## Mathematical Foundation

All speed controllers operate on the **discrete mechanical speed plant** from `documentation/theory/foc-plant-models.md` §2:

$$\omega_m[k+1] = A_d^o \cdot \omega_m[k] + B_d^o \cdot u[k]$$

S3 adds a reference pre-filter  $F(z)$  in the forward path that shapes tracking independently from the feedback loop designed for disturbance stiffness.

---

## S3 — Two-DOF Speed Control

**Structure:** The PI feedback controller receives a pre-filtered setpoint:

$$e_{fb}[k] = F(z) \omega_m^*[k] - \omega_m[k]$$

The pre-filter is a first-order low-pass parameterised by  $\tau_{ff}$ :

$$F(z) = \frac{1 - e^{-T_s^o/\tau_{ff}}}{z - e^{-T_s^o/\tau_{ff}}}$$

**Decoupled transfer functions:**

$$T_{track}(z) = F(z) C_{fb}(z) G(z) (1 + C_{fb}(z) G(z))^{-1}$$

$$T_{dist}(z) = G(z) (1 + C_{fb}(z) G(z))^{-1}$$

$\tau_{ff}$  shapes tracking speed; PI gains shape disturbance rejection. Neither constrains the other.

**Relation to toolbox:** an `ExponentialMovingAverage` realises  $F(z)$  and an incremental PI provides  $C_{fb}(z)$ , with  $\alpha = 1 - e^{-T_s^o/\tau_{ff}}$  mapping the pre-filter time constant onto the filter.

**Tuning:** - PI gains ( $K_p$ ,  $K_i$ ): design for disturbance rejection bandwidth  $\omega_{bw}$  as in standard PI (see `documentation/theory/speed-loop-pi.md`). -  $\tau_{ff}$ : set  $\leq 1/\omega_{bw}$  for tracking at least as fast as the feedback loop. Set larger to deliberately smooth abrupt setpoint changes.

---

## Numerical Properties

| Property                   | Value                                            |
|----------------------------|--------------------------------------------------|
| Ops per 1 kHz cycle        | ~8 MACs (pre-filter + PI)                        |
| Steady-state speed error   | Zero (PI integral action)                        |
| Load disturbance rejection | PI quality                                       |
| Tracking vs. stiffness     | Decoupled ( $\tau_{ff}$ independent of PI gains) |
| Requires mechanical RLS    | No                                               |
| Tuning knobs               | 3 ( $K_p$ , $K_i$ , $\tau_{ff}$ )                |

---

## Limitations & Assumptions

- Treats the current loop as ideal ( $i_q \approx i_q^*$ ). Valid when current-loop bandwidth is at least  $10\times$  the speed-loop bandwidth.
  - $\tau_{ff}$  must be smaller than the closed-loop mechanical time constant to avoid pre-filter lag causing overshoot.
  - If the incoming speed reference is already smooth,  $\tau_{ff}$  can equal the PI time constant, which degenerates the Two-DOF to a standard PI.
  - Speed estimate must be smooth. A noisy encoder derivative degrades all speed controllers equally.
-

## References

1. Åström, K.J. & Wittenmark, B. — *Computer-Controlled Systems*, 3rd ed., Prentice Hall, 1997.
2. Åström, K.J. & Häggglund, T. — *Advanced PID Control*, ISA, 2006. (Two-DOF PID structure, reference pre-filter design.)

| Field     | Value                        |
|-----------|------------------------------|
| Title     | Position Loop — P (Baseline) |
| Type      | theory                       |
| Status    | draft                        |
| Version   | 1.0.0                        |
| Component | position-loop-pid            |
| Date      | 2026-08-31                   |

**Theory document:** Explains the mathematical and engineering principles behind a component or algorithm. This document is descriptive — it records the *why* and *how* at a scientific level, independent of any specific implementation. Equations use KaTeX (*inline* and

*block*

). Block diagrams use Mermaid or ASCII art.

## Overview

The baseline position controller converts position error into a speed reference  $\omega_m^*$  for the inner speed loop. Under cascade separation — inner speed loop bandwidth at least  $5\times$  the position bandwidth — the closed-loop position bandwidth equals  $K_p$  directly. No parameter identification is required.

The law is a **PI**, not the pure P this chapter derives. The derivation below is the P design the cascade shipped with and still the way to read the proportional path; the integral term removes the standing error the P design leaves under a constant velocity reference. Its zero and the weight the proportional term gives the reference are fixed by the design in *Integral zero and reference weighting* below, not by the loop tunings.

Operates exclusively in the **1 kHz outer handler**.

## Prerequisites

| Symbol           | Meaning                                              | Unit  |
|------------------|------------------------------------------------------|-------|
| $\omega_{bw}^p$  | Desired closed-loop position bandwidth               | rad/s |
| $\omega_{bw}^s$  | Inner speed-loop closed-loop bandwidth               | rad/s |
| $\omega_{m,max}$ | Maximum mechanical angular velocity (hardware limit) | rad/s |

---

| Symbol  | Meaning                           | Unit |
|---------|-----------------------------------|------|
| $T_s^o$ | Outer loop sample period (= 1 ms) | s    |

---

See `documentation/theory/foc-plant-models.md §3` for the position plant derivation.

---

## Mathematical Foundation

### Plant Model

Under cascade separation (treating the speed loop as ideal,  $\omega_m \approx \omega_m^*$ ), the position dynamics reduce to a pure integrator:

$$\frac{d\theta_m}{dt} = \omega_m$$

Transfer function from  $\omega_m^*$  to  $\theta_m$ :

$$G_\theta(s) = \frac{1}{s}$$

### P Controller Design

A proportional controller  $C(s) = K_p$  closes the integrator loop. This is the proportional path of the implemented PI; see the note in the Overview:

$$\boxed{\omega_m^*[k] = K_p \cdot \text{wrap}(\theta_m^*[k] - \theta_m[k])}$$

The closed-loop transfer function is:

$$\frac{\Theta_m(s)}{\Theta_m^*(s)} = \frac{K_p}{s + K_p}$$

The position bandwidth equals the proportional gain directly:

$$\omega_{bw}^p = K_p$$

**Bandwidth selection:** Cascade separation requires:

$$K_p = \omega_{bw}^p \leq \frac{\omega_{bw}^s}{5}$$

Exceeding this limit causes the position loop to drive the speed loop into instability.

## Angle-Error Wrapping

Position error must be wrapped to  $(-\pi, \pi]$  to ensure the shortest arc is always taken:

$$e_\theta[k] = \text{wrap}(\theta_m^*[k] - \theta_m[k])$$

$$\text{wrap}(x) = x - 2\pi \text{round}\left(\frac{x}{2\pi}\right)$$

Without wrapping, a +3.1 rad reference reached from -3.1 rad would command a 6.2 rad (354°) rotation through the long arc instead of the 0.08 rad short arc.

## Integral zero and reference weighting

The integral zero sits at a fifth of the position bandwidth,  $K_i = 0.2 \omega_{bw}^p K_p$ , and the proportional term acts on  $b\theta^* - \theta$  with  $b = 0.75$  rather than on the error:

$$\omega_m^*[k] = \omega_m^*[k-1] + K_p (b \Delta\theta^*[k] - \Delta\theta_m[k]) + K_i T_s^o e_\theta[k]$$

in the velocity form the implementation uses, with every difference wrapped as the error is.

The zero used to sit at the integral weight times the bandwidth, a twentieth of it with the default weight. That places a slow closed-loop pole just beside the zero, and the pair leaves a tail whose size is the residue of that pole: under an ideal speed loop at  $\omega_{bw}^p = 18.85$  rad/s the poles fall at -0.995 and -17.9 rad/s against a zero at -0.94, and a 1.5 rad step keeps some 0.089 rad of overshoot that decays over a second — 0.055 rad is still there half a second later. A torque step pushing the same way as that tail starts its excursion already most of the way across a 0.08 rad recovery band.

Moving the zero up to  $\omega_{bw}^p/5$  alone shortens the tail but makes it larger, because the zero also lies in the reference path. Weighting the reference in the proportional term takes the zero out of the reference path without touching disturbance rejection, which sees the full  $K_p$  through  $\Delta\theta_m$ . With  $b = 0.75$  a 1.5 rad step overshoots by well under 1 % and is within 0.01 rad of the setpoint 450 ms after the step.

## Output Limiting

The speed reference is clamped to  $\pm\omega_{m,max}$  to prevent demanding speeds beyond the mechanical or electrical limits of the drive:

$$\omega_m^*[k] = \text{clamp}(K_p \cdot e_\theta[k], -\omega_{m,max}, \omega_{m,max})$$

## Numerical Properties

---

| Property            | Value                              |
|---------------------|------------------------------------|
| Ops per 1 kHz cycle | 2 MACs (1 multiply + wrap + clamp) |

---

---

| Property                    | Value                                                                                     |
|-----------------------------|-------------------------------------------------------------------------------------------|
| Steady-state position error | Non-zero under constant velocity reference for the P design; removed by the integral term |
| Steady-state error at rest  | Zero (no steady velocity $\rightarrow$ no position error)                                 |
| Requires J, Bf              | No                                                                                        |
| Tuning knobs                | 1 ( $K_p = \omega_{bw}^p$ ); the integral zero and reference weight are design constants  |

---



---

## Limitations & Assumptions

- Treats the speed loop as ideal. Valid when  $\omega_{bw}^s \geq 5 \omega_{bw}^p$ .
  - Under a constant velocity reference  $\dot{\theta}_m^* \neq 0$ , a pure P controller has non-zero steady-state tracking error. The implementation carries an integral term for this, and velocity feedforward (as in P2 / Cascade P) addresses it without the integral's phase cost.
  - No disturbance rejection beyond what the speed loop provides.
  - Encoder noise enters the position estimate. A state observer upstream improves robustness.
- 

## References

1. Krishnan, R. — *Permanent Magnet Synchronous and Brushless DC Motor Drives*, CRC Press, 2010.
2. Åström, K.J. & Wittenmark, B. — *Computer-Controlled Systems*, 3rd ed., Prentice Hall, 1997.

---

| Field     | Value                         |
|-----------|-------------------------------|
| Title     | Position Loop — P1: LQR / LQI |
| Type      | theory                        |
| Status    | draft                         |
| Version   | 1.0.0                         |
| Component | position-loop-lqr-lqi         |
| Date      | 2026-08-31                    |

---

**Theory document:** Explains the mathematical and engineering principles behind a component or algorithm. This document is descriptive — it records the *why* and *how* at a scientific level, independent of any specific implementation. Equations use KaTeX (*inline* and

*block*

). Block diagrams use Mermaid or ASCII art.

---

## Overview

P1 (LQR/LQI) simultaneously regulates both  $\theta_m$  and  $\omega_m$  with a single gain vector derived from the physical plant parameters and an explicit performance cost. A position P-controller on top of a speed PI gives ad-hoc gain selection with no principled trade-off between position error, speed overshoot, and control effort. LQR state-feedback replaces this with DARE-computed gains. The LQI extension adds an integral state for zero steady-state error under constant load.

Operates exclusively in the **1 kHz outer handler**. Requires mechanical RLS convergence ( $J$ ,  $B_f$ ,  $K_t$ ).

## Prerequisites

| Symbol               | Meaning                          | Unit             |
|----------------------|----------------------------------|------------------|
| $A_d^p, B_d^p$       | Discrete position plant matrices | —                |
| $K_\theta, K_\omega$ | LQR position and velocity gains  | A/rad, A · s/rad |
| $Q, R$               | LQR weighting matrices           | —                |
| $P$                  | DARE solution                    | —                |

See `documentation/theory/foc-plant-models.md §3` for the position plant derivation.

## Mathematical Foundation

All position-loop controllers operate on the **discrete two-state position plant** from `documentation/theory/foc-plant-models.md §3`:

$$\begin{pmatrix} \theta_m[k+1] \\ \omega_m[k+1] \end{pmatrix} = A_d^p \begin{pmatrix} \theta_m[k] \\ \omega_m[k] \end{pmatrix} + B_d^p u[k], \quad A_d^p = \begin{pmatrix} 1 & T_s^o \\ 0 & A_d^o \end{pmatrix}, \quad B_d^p = \begin{pmatrix} 0 \\ B_d^o \end{pmatrix}$$

## P1 — LQR / LQI Position Control

**LQR gain design:** For the discrete position plant  $(A_d^p, B_d^p)$ , solve the DARE:

$$P = (A_d^p)^T P A_d^p - (A_d^p)^T P B_d^p (R + (B_d^p)^T P B_d^p)^{-1} (B_d^p)^T P A_d^p + Q$$

Optimal gain vector with  $Q = \text{diag}(q_\theta, q_\omega)$ :

$$\mathbf{K} = [K_\theta, K_\omega] = (R + (B_d^p)^T P B_d^p)^{-1} (B_d^p)^T P A_d^p$$

**LQR control law** (2 multiplications + 1 addition at run time):

$$u[k] = -K_\theta (\theta_m[k] - \theta_m^*[k]) - K_\omega (\omega_m[k] - \omega_m^*[k])$$

**LQI extension:** Add an integral state for zero steady-state error under constant load, and solve the same DARE structure extended to dimension 3.

### What the LQI implementation actually solves

The section above is the textbook formulation. The implemented LQI departs from it in four ways, and reading `LqiPositionController::Solve` against the formulation above will not work without them.

**Time-scaled coordinates.** The state is  $x = (x_I, e_\theta, \omega_m T_s^o)^T$  — speed is carried as *radians per sample*, not radians per second, and the integral accumulates the raw per-sample deviation. Scaling every state by  $T_s^o$  this way removes  $T_s^o$  from the plant matrices entirely:

$$A = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & A_d^o \end{pmatrix}, \quad B = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

$B$  is unity, not  $B_d^o$ : the input is normalised, and the plant's actual input gain is carried outside the Riccati solve as `currentPerNormalizedInput`, which converts the optimal command back into Amperes. This is what keeps the solve numerically well conditioned across the range of inertias the mechanical identification returns.

**The integral state is first, not last.** The order is  $[x_I, e_\theta, \omega_m T_s^o]$ , so the gain vector reads  $[K_I, K_\theta, K_\omega]$  — the reverse of the  $[K_\theta, K_\omega, K_I]$  ordering the augmentation above suggests.

**The error sign.**  $e_\theta = \theta_m - \theta_m^*$ , the wrapped error negated, and the solver returns  $u = -Kx$ . The two together give the same law as the LQR section; taken separately, either one alone inverts the feedback.

**Q is normalised, and R comes from the bandwidth.** The weights are ratios against the position-error weight rather than absolute penalties, and the input penalty is set from the desired bandwidth rather than from the current envelope:

$$Q = \text{diag}\left(\frac{q_I}{q_\theta}, 1, \frac{q_\omega}{q_\theta}\right), \quad R = \frac{1}{(\omega_{bw}^p T_s^o)^2}$$

with  $\omega_{bw}^p T_s^o$  clamped to  $[10^{-3}, 0.5]$  and each weight ratio clamped to  $[10^{-3}, 10^3]$ , so no tuning input can make the solve ill-posed. The  $R = 1/I_{q,max}^2$  starting point given under **Tuning** below applies to the textbook formulation, not to this one.

**Tuning:** -  $q_\theta$ : position error penalty. Increase to tighten position accuracy. -  $q_\omega$ : velocity error penalty. Increase to reduce overshoot and ringing. -  $R$ : input penalty. Maps directly to peak  $i_q^*$ ; set  $R = 1/I_{q,max}^2$  as a starting point.

**Stability:** The closed-loop matrix  $A_d^p - B_d^p K$  must have all eigenvalues strictly inside the unit circle. Guaranteed by the DARE solution when the plant is controllable.

**Relation to toolbox:** `Lqr<float,2,1>` provides the gain container. DARE computed at configuration time via `DiscreteAlgebraicRiccatiEquation`. `IntegralStateFeedbackLqi` provides the LQI variant.



Figure 10: Block diagram

## Numerical Properties

| Property                    | Value                                             |
|-----------------------------|---------------------------------------------------|
| Ops per 1 kHz cycle         | 4 MACs ( $\mathbf{K} \cdot x$ , 2-state)          |
| Steady-state position error | Zero with LQI (integral augmentation)             |
| Requires J, Bf              | Yes                                               |
| DARE timing                 | Once at config time, $< 2 \mu\text{s}$ at 120 MHz |
| Tuning knobs                | 3 ( $q_\theta$ , $q_\omega$ , $R$ )               |

## Limitations & Assumptions

- Treats the speed and current loops as ideal. Current-loop transient delay acts as a pure time delay reducing achievable LQR bandwidth. Use a conservative  $q_\omega/q_\theta$  ratio.
- LQI integral state must be clamped during mechanical hard stops (anti-windup required).
- Encoder noise enters position and velocity estimates. A Luenberger or Kalman observer upstream improves robustness.

## References

1. Krishnan, R. — *Permanent Magnet Synchronous and Brushless DC Motor Drives*, CRC Press, 2010.
2. Åström, K.J. & Wittenmark, B. — *Computer-Controlled Systems*, 3rd ed., Prentice Hall, 1997.

| Field | Value                         |
|-------|-------------------------------|
| Title | Position Loop — P2: Cascade P |
| Type  | theory                        |

---

| Field     | Value                   |
|-----------|-------------------------|
| Status    | draft                   |
| Version   | 1.0.0                   |
| Component | position-loop-cascade-p |
| Date      | 2026-08-31              |

---

**Theory document:** Explains the mathematical and engineering principles behind a component or algorithm. This document is descriptive — it records the *why* and *how* at a scientific level, independent of any specific implementation. Equations use KaTeX (*inline* and

*block*

). Block diagrams use Mermaid or ASCII art.

---

## Overview

P2 (Cascade P→PI) is the industry-standard servo position architecture used by SERCOS, EtherCAT, and most industrial servo drives. A proportional position controller outputs a speed reference which is tracked by the inner speed loop. The single parameter  $K_v$  (velocity loop gain) directly sets the position-loop bandwidth in a physically transparent way — the baseline architecture servo engineers expect to configure. No parameter identification is required.

Operates exclusively in the **1 kHz outer handler**.

---

## Prerequisites

---

| Symbol          | Meaning                                                    | Unit          |
|-----------------|------------------------------------------------------------|---------------|
| $K_v$           | Velocity loop gain (position controller proportional gain) | rad/s per rad |
| $K_{ff}$        | Velocity feedforward fraction                              | —             |
| $\omega_{bw}^s$ | Inner speed-loop closed-loop bandwidth                     | rad/s         |

---

See `documentation/theory/foc-plant-models.md §3` for the position plant derivation.

---

## Mathematical Foundation

All position-loop controllers operate on the **discrete two-state position plant** from `documentation/theory/foc-plant-models.md §3`:

$$A_d^p = \begin{pmatrix} 1 & T_s^o \\ 0 & A_d^o \end{pmatrix}, \quad B_d^p = \begin{pmatrix} 0 \\ B_d^o \end{pmatrix}$$

P2 treats the speed loop as ideal and closes only the outer position loop with a proportional gain.

## P2 — Cascade P→PI Position Control

**Control law** (position P controller):

$$\omega_m^*[k] = K_v \cdot (\theta_m^*[k] - \theta_m[k])$$

The speed reference  $\omega_m^*[k]$  is passed to the selected speed-loop controller (PID, LQI, ADRC, or Two-DOF). Cascade P is a position-loop choice; it does not constrain the speed-loop algorithm.

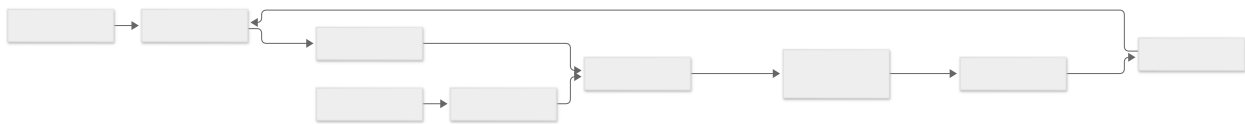


Figure 11: Block diagram

**Position-loop bandwidth:**

$$\omega_{pos} \approx K_v$$

Valid under cascade separation:  $K_v \leq \omega_{speed}/5$ .

**Velocity feedforward extension:**

$$\omega_m^*[k] = K_v (\theta_m^*[k] - \theta_m[k]) + K_{ff} \dot{\theta}_m^*[k]$$

With  $K_{ff} = 1$ , steady-state velocity tracking error is theoretically zero. Use  $K_{ff} \in (0, 1)$  when the velocity reference signal carries noise.

**Tuning:** -  $K_v$  (rad/s / rad): start at  $\omega_{speed}/5$ . -  $K_{ff} \in [0, 1]$ : start at 0; increase after speed loop is tuned and stable.

## Numerical Properties

| Property                    | Value                              |
|-----------------------------|------------------------------------|
| Ops per 1 kHz cycle         | 2 MACs                             |
| Steady-state position error | Speed-loop dependent; zero at rest |
| Industry prevalence         | Dominant in industrial servo       |
| Requires J, Bf              | No                                 |
| Tuning knobs                | 1–2 ( $K_v$ , optional $K_{ff}$ )  |

## Limitations & Assumptions

- Treat speed loop as ideal. Requires a properly tuned inner speed loop.
- $K_v \leq \omega_{speed}/5$  is a hard stability limit. Exceeding it causes oscillation.
- Velocity feedforward  $K_{ff}$  requires a clean velocity reference. Apply a derivative filter before computing  $\dot{\theta}_m^*$  from a noisy position reference.
- Encoder noise enters position and velocity estimates. A state observer upstream improves robustness.

## References

1. Krishnan, R. — *Permanent Magnet Synchronous and Brushless DC Motor Drives*, CRC Press, 2010.
2. Åström, K.J. & Wittenmark, B. — *Computer-Controlled Systems*, 3rd ed., Prentice Hall, 1997.
3. Åström, K.J. & Hägglund, T. — *Advanced PID Control*, ISA, 2006.

| Field     | Value                       |
|-----------|-----------------------------|
| Title     | Position Loop — P3: Two-DOF |
| Type      | theory                      |
| Status    | draft                       |
| Version   | 1.0.0                       |
| Component | position-loop-two-dof       |
| Date      | 2026-08-31                  |

**Theory document:** Explains the mathematical and engineering principles behind a component or algorithm. This document is descriptive — it records the *why* and *how* at a scientific level, independent of any specific implementation. Equations use KaTeX (*inline* and

*block*

). Block diagrams use Mermaid or ASCII art.

## Overview

P3 (Two-DOF position control) eliminates the fundamental tracking vs. stiffness tradeoff inherent in single-DOF position controllers. The reference pre-filter  $F(z)$  shapes how the position setpoint is approached, independently from the feedback loop designed for load-disturbance stiffness. This is the generalisation of Cascade P (P2): when  $F(z) = 1$  and the feedback is proportional, P3 collapses to P2.

Operates exclusively in the **1 kHz outer handler**. No parameter identification required.

## Prerequisites

| Symbol         | Meaning                          | Unit |
|----------------|----------------------------------|------|
| $\tau_{ff}$    | Two-DOF pre-filter time constant | s    |
| $A_d^p, B_d^p$ | Discrete position plant matrices | —    |

See `documentation/theory/foc-plant-models.md §3` for the position plant derivation. See `documentation/theory/position-loop-cascade-p.md` for the Cascade P special case.

## Mathematical Foundation

All position-loop controllers operate on the **discrete two-state position plant** from `documentation/theory/foc-plant-models.md §3`. P3 is the Two-DOF generalisation of the single position-loop feedback controller: a reference pre-filter  $F(z)$  decouples tracking bandwidth from disturbance stiffness bandwidth.

## P3 — Two-DOF Position Control

**Structure:** A PD or LQR feedback controller acts on the pre-filtered position error:

$$e_{fb}[k] = F(z) \theta_m^*[k] - \theta_m[k]$$

**Wrapping constraint:**  $\theta_m$  lives on the circle, not on the line, so  $F(z)$  must not be run directly on the wrapped angle: filtering from  $+3.0$  rad towards  $-3.0$  rad would interpolate through 0 and execute a  $344^\circ$  rotation instead of the  $16^\circ$  move across the seam. The first-order prefilter is therefore realised on the wrapped error:

$$\theta_f[k] = \text{wrap}\left(\theta_f[k-1] + \alpha \cdot \text{wrap}(\theta_m^*[k] - \theta_f[k-1])\right)$$

with  $\alpha = 1 - e^{-T_o/\tau_{ff}}$  and  $\text{wrap}(\cdot)$  folding onto  $(-\pi, \pi]$ . The state  $\theta_f$  is seeded from the measured angle on reset and on the first sample; seeding it at zero would ramp the reference in from absolute zero on every enable. For  $\tau_{ff} \leq 0$  the prefilter is transparent and the raw setpoint passes through.

**Relation to P2:** When  $F(z) = 1$  and the feedback controller is a proportional P, the structure collapses to Cascade P→PI. Two-DOF position control is the generalisation —  $F(z)$  independently shapes tracking and the feedback controller independently shapes stiffness.

**Relation to toolbox:** `Feedforward2Dof` implements this structure.

**Tuning:**  $\tau_{ff}$  for tracking speed; feedback gains for disturbance rejection. As S3.

## Numerical Properties

| Property                    | Value                                            |
|-----------------------------|--------------------------------------------------|
| Ops per 1 kHz cycle         | ~8 MACs (pre-filter + feedback controller)       |
| Steady-state position error | Configurable (depends on feedback controller)    |
| Tracking vs. stiffness      | Decoupled ( $\tau_{ff}$ independent of feedback) |
| Requires J, Bf              | No                                               |
| Tuning knobs                | $\tau_{ff}$ + feedback controller gains          |

## Limitations & Assumptions

- Treat speed loop as ideal. P3 requires a properly tuned inner speed loop.
- Pre-filter pole must be faster than the desired closed-loop position bandwidth. Verify that  $1/\tau_{ff} > \omega_{pos}$ .
- The wrapping realisation is mandatory — direct application of  $F(z)$  to the angle signal is numerically incorrect for setpoints that cross the  $\pm\pi$  seam.
- Encoder noise enters position and velocity estimates. A state observer upstream improves robustness.

## References

1. Åström, K.J. & Wittenmark, B. — *Computer-Controlled Systems*, 3rd ed., Prentice Hall, 1997.
2. Åström, K.J. & Hägglund, T. — *Advanced PID Control*, ISA, 2006. (Two-DOF PID structure, reference pre-filter design.)

| Field     | Value                                          |
|-----------|------------------------------------------------|
| Title     | Position Loop — P4: Iterative Learning Control |
| Type      | theory                                         |
| Status    | draft                                          |
| Version   | 1.0.0                                          |
| Component | position-loop-ilc                              |
| Date      | 2026-08-31                                     |

**Theory document:** Explains the mathematical and engineering principles behind a component or algorithm. This document is descriptive — it records the *why* and *how* at a scientific level, independent of any specific implementation. Equations use KaTeX (*inline* and

*block*

). Block diagrams use Mermaid or ASCII art.

---

**Not implemented.** No such algorithm exists in the firmware: `foc::PositionAlgorithm` offers `pid`, `cascadeP`, `lqr`, `lqi` and `twoDof`, and nothing else is selectable over CAN or the CLI. This chapter is design groundwork for a law that has not been built, kept for the theory rather than as a description of shipped behaviour. Read it as a proposal, not as a specification.

---

## Overview

P4 (ILC — Iterative Learning Control) is for repetitive servo tasks — pick-and-place, CNC passes, inspection cycles. The same position tracking error recurs on every trial because the same reference is executed against the same mechanical dynamics. ILC exploits this: it learns the per-sample feedforward correction that cancels the repeating error over successive trials until the residual is near the sensor noise floor. No other controller achieves this tracking accuracy for periodic tasks.

ILC does not replace the feedback controller — it augments any stable position feedback controller (P1, P2, or P3). Operates exclusively in the **1 kHz outer handler**.

---

## Prerequisites

| Symbol | Meaning                         | Unit |
|--------|---------------------------------|------|
| $Q$    | ILC robustness filter scalar    | —    |
| $\ell$ | ILC learning gain               | —    |
| $N$    | Trial length in samples (fixed) | —    |

---

See `documentation/theory/foc-plant-models.md` §3 for the position plant derivation.

---

## Mathematical Foundation

All position-loop controllers operate on the **discrete two-state position plant** from `documentation/theory/foc-plant-models.md` §3. P4 augments any stable feedback controller with a per-sample learned feedforward. Friction compensation adds a nonlinear  $I_q$  correction outside the linear plant model.

---

## P4 — Iterative Learning Control (ILC)

**D-type update law:** Let  $e^{(j)}[k] = \theta_m^*[k] - \theta_m^{(j)}[k]$  be the position error on trial  $j$  at sample  $k$ :

$$u_{ILC}^{(j+1)}[k] = Q \cdot \left( u_{ILC}^{(j)}[k] + \ell \cdot e^{(j)}[k+1] \right)$$

where: -  $Q \in (0, 1]$ : robustness filter.  $Q < 1$  suppresses high-frequency noise at the cost of slower convergence. Typical:  $Q = 0.95$ . -  $\ell > 0$ : learning gain. Convergence guaranteed for  $\ell \in (0, 2)$  on a unity-gain plant.

**Total control signal:** Learned correction added as feedforward to any position feedback controller:

$$u_{total}^{(j)}[k] = u_{fb}^{(j)}[k] + u_{ILC}^{(j)}[k]$$

$u_{fb}$  is the output of the active position feedback controller (P2, P1, or P3). ILC does not replace the feedback controller — it augments it.

**Storage:** One full trial of  $N$  samples stored in a bounded array. At 1 kHz, a 5-second trial requires  $N = 5000$  floats (20 kB).  $N$  is fixed at build time; ILC is the only position algorithm whose storage scales with trial duration.

**Convergence condition:**  $|Q(1 - \ell P_{nom})| < 1$  across all frequencies, where  $P_{nom}$  is the nominal position plant. Guaranteed by  $Q < 1$  and  $\ell$  within its stability range.

**Tuning:** -  $N$ : trial length in samples (must exactly match the repetitive task period). -  $Q$ : robustness vs. convergence speed tradeoff. -  $\ell$ : learning rate. Start at 0.5, increase toward 1.0.

---

## Numerical Properties

| Property                    | Value                                               |
|-----------------------------|-----------------------------------------------------|
| Ops per 1 kHz cycle         | 2 MACs + 1 array read                               |
| Steady-state position error | Near-zero after learning cycles                     |
| Storage                     | $N \times \text{float}$ (bounded array, fixed $N$ ) |
| Suitable for                | Periodic / repetitive references only               |
| Requires J, Bf              | No                                                  |
| Tuning knobs                | 3 ( $N, Q, \ell$ )                                  |

---

## Limitations & Assumptions

- Only valid for strictly repetitive references. A changed reference period or non-repeatable reference (jogging) will cause the learned correction from the previous trial to degrade tracking.
- Trial length  $N$  is fixed at selection time. It cannot change at runtime.
- ILC requires a stable inner feedback controller (P1, P2, or P3). ILC alone does not stabilise position — it only improves tracking of the feedback loop.
- Encoder noise enters position and velocity estimates. A state observer upstream improves robustness.

---

## References

1. Bristow, D.A., Tharayil, M. & Alleyne, A.G. — “A Survey of Iterative Learning Control”, *IEEE Control Systems Magazine*, 26(3):96–114, 2006.
2. Åström, K.J. & Wittenmark, B. — *Computer-Controlled Systems*, 3rd ed., Prentice Hall, 1997.

| Field     | Value                                             |
|-----------|---------------------------------------------------|
| Title     | Position Loop — Friction Compensation Feedforward |
| Type      | theory                                            |
| Status    | draft                                             |
| Version   | 1.0.0                                             |
| Component | position-loop-friction                            |
| Date      | 2026-08-31                                        |

**Theory document:** Explains the mathematical and engineering principles behind a component or algorithm. This document is descriptive — it records the *why* and *how* at a scientific level, independent of any specific implementation. Equations use KaTeX (*inline* and

*block*

). Block diagrams use Mermaid or ASCII art.

---

**Not implemented.** No such algorithm exists in the firmware: `foc::PositionAlgorithm` offers `pid`, `cascadeP`, `lqr`, `lqi` and `twoDof`, and nothing else is selectable over CAN or the CLI. This chapter is design groundwork for a law that has not been built, kept for the theory rather than as a description of shipped behaviour. Read it as a proposal, not as a specification.

---

## Overview

Friction compensation is not a selectable algorithm — it is an independently enabled feedforward that stacks on top of any speed or position feedback controller. Viscous friction ( $B_f \cdot \omega_m$ ) is estimated by the mechanical RLS and compensated implicitly by the feedback controllers. Coulomb and Stribeck friction are nonlinear: at low speeds the motor sticks until the drive overcomes the static breakaway torque, then slips. This creates position hunting at rest and velocity ripple during slow moves — the classic precision servo failure mode that no linear feedback controller can fully eliminate.

---

## Prerequisites

| Symbol          | Meaning                           | Unit                        |
|-----------------|-----------------------------------|-----------------------------|
| $T_c$           | Coulomb (kinetic) friction torque | $\text{N} \cdot \text{m}$   |
| $T_s$           | Static (breakaway) torque         | $\text{N} \cdot \text{m}$   |
| $\omega_{st}$   | Stribeck velocity                 | $\text{rad/s}$              |
| $K_t$           | Torque constant                   | $\text{N} \cdot \text{m/A}$ |
| $\omega_{dead}$ | Dead-zone smoothing width         | $\text{rad/s}$              |

Parameters  $T_c$ ,  $T_s$ ,  $\omega_{st}$  are identified by a dedicated friction sweep — separate from the mechanical RLS that estimates only the linear viscous coefficient  $B_f$ .

## Mathematical Foundation

### Friction Model (Coulomb + Stribeck)

$$T_f(\omega_m) = \left[ T_c + (T_s - T_c) e^{-(\omega_m/\omega_{st})^2} \right] \cdot \text{sgn}(\omega_m)$$

where: -  $T_c$ : Coulomb (kinetic) friction torque — the flat plateau at moderate speed -  $T_s$ : static (breakaway) torque,  $T_s > T_c$  — the peak at near-zero speed -  $\omega_{st}$ : Stribeck velocity — speed at which kinetic friction reaches its minimum

### Current Feedforward

Converting friction torque to  $i_q$  and injecting before the current loop:

$$i_{ff}(\omega_m) = \frac{T_f(\omega_m)}{K_t}$$

$$i_q^{*,total} = i_q^{*,ctrl} + i_{ff}(\omega_m)$$

### Dead-Zone Smoothing

The  $\text{sgn}(\omega_m)$  discontinuity at  $\omega_m = 0$  must be replaced with a linear ramp over  $\pm\omega_{dead}$  to prevent chattering during direction reversals and holding. Typical  $\omega_{dead} = 0.05\text{--}0.2$  rad/s.

### Parameter Identification

$T_c$ ,  $T_s$ , and  $\omega_{st}$  require a dedicated friction sweep (ramp velocity at steady state while logging  $i_q^*$ ). This is separate from the mechanical RLS, which estimates only the linear viscous coefficient  $B_f$ .

**Tuning:** -  $T_c$  ( $\text{N} \cdot \text{m}$ ): read from the  $i_q^* \times K_t$  plateau at constant low speed. -  $T_s$  ( $\text{N} \cdot \text{m}$ ): read from the breakaway  $i_q^*$  at near-zero speed. -  $\omega_{st}$  ( $\text{rad/s}$ ): speed at which the  $i_q^*$  vs.  $\omega_m$  curve reaches its kinetic minimum.

## Numerical Properties

| Property       | Value                                          |
|----------------|------------------------------------------------|
| Applied to     | $i_q^*$ at speed or position controller output |
| Ops per cycle  | ~6 MACs + exp (fast approximation)             |
| Storage        | Constant (3 parameters)                        |
| Identification | Dedicated friction sweep required              |

## Limitations & Assumptions

- Over-compensation ( $T_s$  or  $T_c$  over-estimated) causes position overshoot at zero crossing. Prefer slight under-compensation and let the feedback controller handle the residual.
- Friction parameters drift with temperature and wear. Re-identification is recommended when low-speed precision degrades.
- The  $\text{sgn}(\omega_m)$  dead-zone must be smoothed to prevent chattering; too wide a dead-zone reintroduces hunting in the smoothed region.

## References

1. Armstrong-Hélouvry, B., Dupont, P. & De Wit, C.C. — “A Survey of Models, Analysis Tools and Compensation Methods for the Control of Machines with Friction”, *Automatica*, 30(7):1083–1138, 1994.
2. Krishnan, R. — *Permanent Magnet Synchronous and Brushless DC Motor Drives*, CRC Press, 2010.

| Field     | Value                                          |
|-----------|------------------------------------------------|
| Title     | Electrical Parameters Identification — R and L |
| Type      | theory                                         |
| Status    | approved                                       |
| Version   | 2.1.0                                          |
| Component | service-electrical-ident                       |
| Date      | 2026-09-21                                     |

## Overview

FOC performance depends directly on accurate stator resistance  $R_s$  and inductance  $L_s$ . These parameters are used to: 1. Design the PI current controller gains ( $K_p = L_s \omega_{bw}$ ,  $K_i = R_s \omega_{bw}$ ). 2. Implement feed-forward decoupling of the dq cross-coupling terms. 3. Estimate motor temperature from measured  $R_s$  (since  $R_s \propto T$ ).

Two complementary offline identification methods are used, one for each parameter:

- **Resistance** — DC voltage step: steady-state V/I gives  $R_s$  directly. Accurate for any motor.
- **Inductance** — HF sinusoidal injection: synchronous demodulation of the current response at the injection frequency extracts  $\text{Im}(Z)$  and thus  $L_s$ . Suitable for low-resistance motors where the time-constant method fails.

The reason two separate methods are needed is a fundamental signal-conditioning problem. At the 700 Hz injection frequency the ratio  $\omega L_s/R_s$  is 7 for the Anaheim BLY172S-24V-4000 ( $R_s = 0.405 \Omega$ ,  $L_s = 0.64 \text{ mH}$ ) and 2.4 for the Teknic M-2310P-LN-04K ( $R_s = 0.36 \Omega$ ,  $L_s = 0.20 \text{ mH}$ ), so resistance is 14 % and 38 % of the impedance.

For lower-resistance windings that share falls below 5 %, and any small phase error drowns the R signal entirely. The time-constant method (section 4 of version 1.0) likewise fails because  $\tau = L_s/R_s$  is 1.6 ms for the Anaheim and 0.56 ms for the Teknic, 16 and 5.6 samples at 10 kHz — too few for threshold-based detection through a filter of comparable length.

Each parameter is therefore extracted using the technique that makes it the dominant signal.

---

## Prerequisites

| Symbol     | Meaning                                                            | Unit     |
|------------|--------------------------------------------------------------------|----------|
| $R_s$      | Stator resistance per phase                                        | $\Omega$ |
| $L_s$      | Stator inductance ( $L_d$ , assuming surface PMSM)                 | H        |
| $V_{step}$ | DC step terminal voltage                                           | V        |
| $I_{ss}$   | Steady-state current after DC step                                 | A        |
| $f_{inj}$  | Sinusoidal injection frequency                                     | Hz       |
| $\omega$   | Angular injection frequency = $2\pi f_{inj}$                       | rad/s    |
| $V_{inj}$  | Terminal sinusoidal injection amplitude                            | V        |
| $f_s$      | Sampling frequency (10 kHz)                                        | Hz       |
| $T_s$      | Sampling period = $1/f_s$                                          | s        |
| $N$        | Goertzel block length = $N_{periods} \times N_{spp}$               | samples  |
| $N_{spp}$  | Samples per injection period = $\text{round}(f_s/f_{inj})$         | samples  |
| $k$        | Goertzel bin index = $N_{periods}$                                 | —        |
| $d$        | ADC pipeline delay                                                 | samples  |
| $a_d, b_d$ | ZOH coefficients: $a_d = e^{-R_s T_s/L_s}$ , $b_d = (1 - a_d)/R_s$ | —        |

---

## Mathematical Foundation

### 1. Terminal Excitation and Winding Factor

Both procedures apply voltage across one terminal versus the other two shorted together. For a wye machine, terminal A sees  $R_s$  in series with the other two in parallel; for delta it sees two windings in parallel:

$$R_{terminal} = \begin{cases} \frac{3}{2}R_s & \text{wye} \\ \frac{1}{2}R_s & \text{delta} \end{cases}, \quad L_{terminal} = \begin{cases} \frac{3}{2}L_s & \text{wye} \\ \frac{1}{2}L_s & \text{delta} \end{cases}$$

The same topology factor applies to both parameters. The applied voltage is the *differential* duty  $(D_{test} - D_{neutral}) \cdot V_{dc}$  for the DC step, and for the sinusoidal injection the Clarke inverse of  $\{v, 0\}$  maps to three-phase duties  $\{v, -v/2, -v/2\}$  (phases relative to midpoint), which produces a terminal A-to-BC amplitude of  $v \cdot 0.75 \cdot V_{dc}$ .

## 2. Resistance — DC Voltage Step

A known DC voltage  $V_{step}$  is held on the winding until the current reaches steady state ( $\gg 5\tau$ ). At steady state all  $L di/dt$  terms vanish:

$$R_s = \frac{V_{step}}{I_{ss}} \cdot \frac{1}{F_{terminal}}$$

where  $F_{terminal}$  is the winding topology factor above and  $I_{ss}$  is the mean of the last 10 % of the sample buffer.

**Why this fails for L:** the DC step reveals only the time constant  $\tau = L_s/R_s$ . For the Anaheim BLY172S ( $L_s = 0.64$  mH,  $R_s = 0.405 \Omega$ )  $\tau = 1.58$  ms, which spans only 16 samples at 10 kHz, and for the Teknic M-2310P (0.20 mH,  $0.36 \Omega$ ) 5.6 samples. A 5-sample moving average and a single-sample crossing threshold cannot resolve such a narrow transient accurately. Inductance is therefore obtained by a separate frequency-domain method.

### DC Step Response

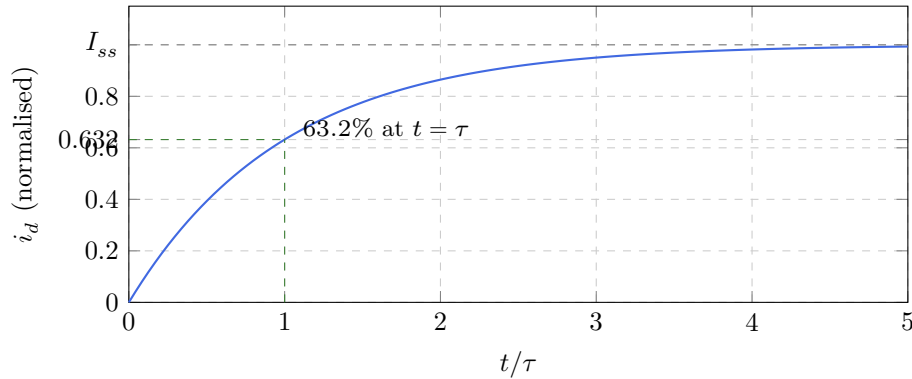


Figure 12: DC step response:  $i_d = I_{ss}(1 - e^{-t/\tau})$ .  $I_{ss} = V_{step}/R_{terminal}$ . The  $\tau$  crossing is marked but not used by the v2 estimator.

### 3. Inductance — HF Sinusoidal Injection

#### 3.1 Why Goertzel?

The inductance estimator must extract the amplitude and phase of the current response at exactly one known frequency. Several approaches were considered:

| Approach                                                                     | Why rejected                                                                                                                             |
|------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------|
| Full FFT                                                                     | Computes all bins; requires an N-sample RAM buffer (280–2000 samples = 1–8 kB); no embedded heap                                         |
| Manual I/Q accumulation<br>( $\Sigma i \cdot \sin$ , $\Sigma i \cdot \cos$ ) | Equivalent mathematically, but requires running $\sin / \cos$ generation at every ISR sample (two trig calls per sample in the hot path) |
| Cross-correlation with stored reference                                      | Requires storing the reference sinusoid — another N-sample buffer                                                                        |
| Lock-in amplifier (hardware)                                                 | Purely analog; not applicable in software                                                                                                |

The **Goertzel algorithm** is a two-pole recursive filter:

$$s[n] = x[n] + 2 \cdot \cos(\omega_0) \cdot s[n-1] - s[n-2]$$

It accumulates the single-bin DFT coefficient  $X[k] = \sum_{n=0}^{N-1} x[n] e^{-j2\pi kn/N}$  using only three state variables ( $s_1$ ,  $s_2$ , sample count). It is streaming (one sample at a time), needs no stored reference, and delivers an exact complex result when `Ready()`. For  $N = 280$  samples at 700 Hz, it uses 12 bytes of working state versus 1.1 kB for an I/Q reference buffer.

**Exact bin alignment:** the block size is chosen as  $N = N_{periods} \times N_{spp}$  and the Goertzel bin as  $k = N_{periods}$ , so exactly  $N_{periods}$  complete injection cycles fit in the window. Any mismatch between injection frequency and analysis bin causes spectral leakage that biases  $\text{Im}(Z)$ . The implementation snaps the injection to  $f_{inj}^{exact} = f_s / N_{spp}$  (nearest integer-sample-per-period frequency) to eliminate leakage entirely.

#### 3.2 Injection and Demodulation

The sinusoidal voltage  $v[n] = V_{inj} \sin(\omega n T_s)$  is injected on the alpha axis only (beta = 0), keeping net electromagnetic torque at zero so the rotor remains stationary.

Over the measurement window, the Goertzel accumulates the current response and returns

$$I_{Goertzel} = \sum_{n=0}^{N-1} i[n] e^{-j\omega n T_s}$$

For a sinusoidal current  $i[n] = I_0 \sin(\omega n T_s + \phi)$ , this gives  $I_{Goertzel} = -j(N/2) I_0 e^{-j\phi}$ .

The voltage phasor at the same bin is  $V_{Goertzel} = -j(N/2) V_{inj}$  (for a sine input). The terminal impedance is then:

$$Z_{terminal} = \frac{V_{Goertzel}}{I_{Goertzel}}$$

For an RL circuit,  $\text{Im}(Z) = \omega L_{\text{terminal}}$ , so:

$$L_s = \frac{\text{Im}(Z_{\text{terminal}})}{\omega \cdot F_{\text{terminal}}}$$

In practice, the full complex division reduces to:

$$Z_{\text{imag}} = -\frac{V_{\text{inj}} \cdot N/2 \cdot \text{Re}(I_{\text{corrected}})}{|I_{\text{corrected}}|^2}$$

where  $I_{\text{corrected}}$  is the delay-compensated Goertzel result (see section 3.3).

### 3.3 ZOH Discretisation Bias

The sampler is a zero-order-hold (ZOH), not a true continuous-time integrator. At 10 kHz sampling, the ZOH introduces a systematic bias in  $\text{Im}(Z)$ :

$$Z_{\text{ZOH}} = \frac{1 - a_d e^{-j\omega T_s}}{b_d} \implies \text{Im}(Z_{\text{ZOH}}) = \frac{a_d \sin(\omega T_s)}{b_d}$$

The fractional error in  $L_s$  relative to the continuous-time value scales approximately as  $R_s T_s / L_s$ :

| Motor                                           | $R_s T_s / L_s$ | ZOH bias in $L_s$ (estimate) | Measured in<br>SIL |
|-------------------------------------------------|-----------------|------------------------------|--------------------|
| Anaheim BLY172S-24V-4000<br>(terminal)          | 0.063           | $\approx 3 \%$               | $-6 \%$            |
| Generic (R=0.5 $\Omega$ , L=0.5 mH<br>terminal) | 0.10            | $\approx 5 \%$               |                    |
| Teknic M-2310P-LN-04K<br>(terminal)             | 0.18            | $\approx 9 \%$               | $-15.5 \%$         |

The terminal factor cancels in the ratio, so it is the per-phase  $R_s / L_s$  that sets the bias. The measured column is the software-in-the-loop identification against the exact plant, whose whole-percent duty quantises the 15 % injection and adds to the ZOH term; the estimate reads low in both cases. For the Anaheim motor the error is within calibration tolerance; for the Teknic it is not negligible. A correction using the already-measured  $R_s$  and  $b_d$  can be applied post-hoc; it is not implemented today.

### 3.4 ADC Delay Correction

The ADC delivers current sample  $i[k]$  that was measured from the voltage applied at step  $k - d$  (typically  $d = 1$  sample due to ISR pipeline latency). This introduces a phase lag of  $\omega \cdot d \cdot T_s$  on the Goertzel result. Correcting it requires a single complex rotation applied after the Goertzel completes — no per-sample cost:

$$I_{\text{corrected}} = I_{\text{Goertzel}} \cdot e^{+j\omega d T_s}$$

Applying the correction with the wrong sign doubles the delay error instead of cancelling it.

### 3.5 Coherence (fitQuality)

The ratio of signal power at  $f_{inj}$  to total current power is a dimensionless quality metric:

$$\text{fitQuality} = \frac{2 |I_{Goertzel}|^2}{N \cdot \sum_n i[n]^2} \in [0, 1]$$

This equals 1.0 for a pure sinusoid at  $f_{inj}$  and drops toward zero when the current contains large non-injection-frequency content (noise, harmonics, rotor motion). Calibration is rejected when fitQuality falls below 0.5.

### 3.6 Hardware Note — Dead-Time Voltage Error

Inverter dead-time introduces a voltage error at the switching frequency that corrupts the injection signal. At 500 ns dead-time and 20 kHz PWM the error voltage is approximately  $2 \cdot t_{dt} \cdot f_{sw} \cdot V_{dc} \approx 0.48 \text{ V}$  at  $V_{dc} = 24 \text{ V}$ . At 10 % injection ( $V_{inj} = 2.4 \text{ V}$ ) this represents 20 % distortion; at 15 % ( $V_{inj} = 3.6 \text{ V}$ ) it falls to 13 %. The minimum recommended injection voltage is **15 % of bus** to keep dead-time distortion below 15 % of the injected signal. The software model does not reproduce dead-time; simulation results will be optimistic relative to hardware.

### Signal Flow

---

## 4. Pole Pair Estimation

The number of electrical cycles per mechanical revolution equals the number of pole pairs  $p$ . During the pole-pair electrical-angle sweep, the motor is driven through exactly  $N_{rev}$  electrical revolutions over  $12 \cdot N_{rev}$  steps. The total mechanical angle  $\Delta\theta_{mech}$  is accumulated from the encoder. The pole pairs are:

$$p = \text{round}\left(\frac{2\pi N_{rev}}{\Delta\theta_{mech, total}}\right)$$

If the total mechanical rotation is below  $\pi/2$  (rotor not following the field), the result is absent.

---

## Numerical Properties

| Property      | DC Step (R)                           | HF Sinusoidal (L)                             |
|---------------|---------------------------------------|-----------------------------------------------|
| Sampling rate | 10 kHz                                | 10 kHz                                        |
| Excitation    | Differential DC step<br>(testVoltage) | Alpha-axis sine at $f_{inj}$ (default 700 Hz) |

| Property              | DC Step (R)                        | HF Sinusoidal (L)                                    |
|-----------------------|------------------------------------|------------------------------------------------------|
| Working memory        | 5-sample deque + 123-sample buffer | 3 floats (Goertzel state) + sumSquared + sampleCount |
| Settling required     | $\geq 5\tau$ before R is valid     | warmupPeriods full cycles (rotor transient decay)    |
| Result quality gate   | $I_{ss} > 0$                       | fitQuality $\geq 0.5$                                |
| Min injection voltage | —                                  | $\geq 15\%$ bus (dead-time floor on hardware)        |
| ZOH bias in L         | —                                  | $\approx R_s T_s / L_s$ fractional underestimate     |

## Limitations

- **Assumes rotor stationary.** Any rotor motion during either procedure overlays back-EMF on the measurements.
- **Assumes magnetic linearity.** Injection current must stay below the saturation onset of the winding.
- **Assumes  $L_d \approx L_q$ .** For interior PMSM a separate  $L_q$  measurement is required.
- **ZOH bias** grows with  $R_s T_s / L_s$ : about 3 % for the Anaheim BLY172S and 9 % for the Teknic M-2310P at 10 kHz.
- **Dead-time** on real hardware distorts the sinusoidal injection; models and simulation are optimistic.
- **Temperature:**  $R_s$  measurement is valid at the moment of identification only.

## References

1. Rauf, A. et al. — “Online Identification of PMSM Parameters Based on Extended Kalman Filter”, *IEEE Transactions on Industrial Electronics*, 2019.
2. Underwood, S.J. & Husain, I. — “Online Parameter Estimation and Adaptive Control of PMSM”, *IEEE Transactions on Industrial Electronics*, 2010.
3. Texas Instruments Application Report SPRABV5 — *Motor Control in Embedded Applications*, 2018.

| Field     | Value                                                       |
|-----------|-------------------------------------------------------------|
| Title     | Mechanical Parameters Identification — Friction and Inertia |
| Type      | theory                                                      |
| Status    | approved                                                    |
| Version   | 1.0.0                                                       |
| Component | service-mechanical-ident                                    |
| Date      | 2025-01-01                                                  |

## Overview

Accurate knowledge of the mechanical plant parameters — rotor moment of inertia  $J$ , viscous friction coefficient  $B$ , and Coulomb (static) friction torque  $\tau_0$  — is required to design an optimal

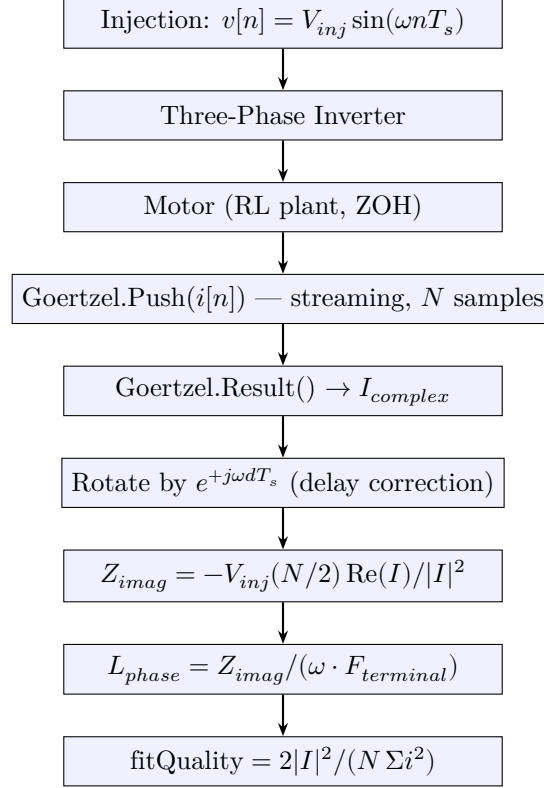


Figure 13: HF sinusoidal injection signal flow: injected voltage through inverter and RL motor plant to Goertzel demodulation and final  $L$  / quality outputs.

speed or position loop, to implement feed-forward compensation, and to estimate system health over time.

This identification procedure uses **Recursive Least Squares (RLS)** to fit the parameters of Newton’s rotational equation of motion to real-time measurements of torque (estimated from  $i_q$ ) and speed/ acceleration (derived from the encoder). RLS operates online in every control cycle, converging to the true parameters as the motor is excited with varying speed profiles.

## Prerequisites

| Symbol        | Meaning                                              | Unit                                                |
|---------------|------------------------------------------------------|-----------------------------------------------------|
| $J$           | Rotor moment of inertia                              | $\text{kg} \cdot \text{m}^2$                        |
| $B$           | Viscous friction coefficient                         | $\text{N} \cdot \text{m} \cdot \text{s}/\text{rad}$ |
| $\tau_0$      | Coulomb friction (static offset)                     | $\text{N} \cdot \text{m}$                           |
| $\tau_e$      | Electrical (motor) torque = $\frac{3}{2}p\psi_f i_q$ | $\text{N} \cdot \text{m}$                           |
| $\tau_{load}$ | External load torque (treated as disturbance)        | $\text{N} \cdot \text{m}$                           |
| $\omega$      | Rotor mechanical angular velocity                    | $\text{rad}/\text{s}$                               |
| $\alpha$      | Rotor angular acceleration = $\dot{\omega}$          | $\text{rad}/\text{s}^2$                             |
|               | Parameter vector $[J, B, \tau_0]^T$                  | mixed                                               |

---

| Symbol       | Meaning                                  | Unit  |
|--------------|------------------------------------------|-------|
|              | Regressor vector $[\alpha, \omega, 1]^T$ | mixed |
| $\lambda$    | RLS forgetting factor                    | —     |
| $\mathbf{P}$ | Error covariance matrix (3×3)            | —     |
| $\mathbf{K}$ | Kalman gain vector (3×1)                 | —     |

---

## Mathematical Foundation

### 1. Mechanical Equation of Motion

The PMSM rotor dynamics are described by Newton's second law for rotation. Assuming viscous friction and a constant Coulomb friction offset:

$$\tau_e = J\dot{\omega} + B\omega + \tau_0 + \tau_{load}$$

Ignoring unmeasured load torque (treated as noise by the estimator), this becomes a **linear regression**:

$$\tau_e = \mathbf{\Phi}^T \mathbf{\theta}$$

where:

$$\mathbf{\Phi} = \begin{pmatrix} \dot{\omega} \\ \omega \\ 1 \end{pmatrix}, \quad \mathbf{\theta} = \begin{pmatrix} J \\ B \\ \tau_0 \end{pmatrix}$$

The regression form makes the parameter vector  $\mathbf{\theta}$  directly estimable from measured input-output data  $(\mathbf{\Phi}, \tau_e)$  using least squares.

### 2. Batch Least Squares (Background)

If  $N$  measurements are collected:

$$\mathbf{y} = \mathbf{\Phi} \mathbf{\theta} + \mathbf{e}$$

where  $\mathbf{y} \in \mathbb{R}^N$  is the torque sequence,  $\mathbf{\Phi} \in \mathbb{R}^{N \times 3}$  is the regressor matrix, and  $\mathbf{e}$  is noise. The ordinary least-squares solution is:

$$\hat{\mathbf{\theta}} = (\mathbf{\Phi}^T \mathbf{\Phi})^{-1} \mathbf{\Phi}^T \mathbf{y}$$

This requires storing all  $N$  measurements and inverting a 3×3 matrix. RLS computes the same result recursively, processing one sample at a time with  $O(n^2)$  cost where  $n = 3$  is the number of parameters.

### 3. Recursive Least Squares — Derivation

Define the weighted least-squares cost with exponential forgetting factor  $\lambda \in (0, 1]$ :

$$J_N = \sum_{k=1}^N \lambda^{N-k} (\tau_e[k] - [k]^T \hat{\gamma})^2$$

The exponential weight  $\lambda^{N-k}$  discounts older measurements, giving the estimator a finite effective memory window of  $\approx 1/(1 - \lambda)$  samples.

Applying the **matrix inversion lemma (Woodbury identity)** to the recursive update of  $\mathbf{P}_N = (\sum_{k=1}^N \lambda^{N-k} [k] [k]^T)^{-1}$  yields:

$$\mathbf{K}[n] = \frac{\mathbf{P}[n-1] [n]}{\lambda + [n]^T \mathbf{P}[n-1] [n]}$$

$$\hat{\gamma}[n] = \hat{\gamma}[n-1] + \mathbf{K}[n] (\tau_e[n] - [n]^T \hat{\gamma}[n-1])$$

$$\mathbf{P}[n] = \frac{1}{\lambda} (\mathbf{I} - \mathbf{K}[n] [n]^T) \mathbf{P}[n-1]$$

The scalar  $e[n] = \tau_e[n] - [n]^T \hat{\gamma}[n-1]$  is called the **innovation** (prediction error). The gain  $\mathbf{K}$  weights how much to move the estimate in response to the innovation.

#### Forgetting Factor

For  $\lambda = 1$  (no forgetting) the RLS is equivalent to batch LS over the entire history. For  $\lambda < 1$  older data is exponentially discounted. The effective memory window is:

$$N_{eff} = \frac{1}{1 - \lambda}$$

With  $\lambda = 0.9995$  — the value `RealTimeFrictionAndInertiaEstimator::defaultForgettingFactor` carries —  $N_{eff} = 2000$  samples, two seconds at the 1 kHz outer-loop rate. The memory has to span more than one level of the excitation: within a single plateau the intercept and the speed column are collinear, so a memory shorter than a plateau (the 200 ms that  $\lambda = 0.995$  gave against a 250 ms dwell) can never separate  $B$  from  $\tau_0$ , and the online friction estimate wandered around zero.

A forgetting factor also inflates the covariance in any direction the regressor does not excite, by  $\lambda^{-1}$  per update. A plateau still informs  $B$  and  $\tau_0$  but not  $J$ , so the online estimator takes plateau samples only while the covariance trace stays under a ceiling, and accelerating samples always; see `documentation/design/service-mechanical-ident.md` § *Persistence of Excitation*.

#### 4. Torque and Kinematics Estimation

**Torque:** The motor torque is estimated from the q-axis current:

$$\tau_e = \frac{3}{2} p \psi_f i_q = K_T i_q$$

The torque constant  $K_T = \frac{3}{2} p \psi_f$  must be known from the electrical identification stage or from the motor datasheet.

**Speed:** Measured directly from the encoder as:

$$\omega[n] = \frac{\theta_m[n] - \theta_m[n-1]}{T_s}$$

**Acceleration:** Estimated by finite difference of speed:

$$\dot{\omega}[n] = \frac{\omega[n] - \omega[n-1]}{T_s}$$

The double finite difference amplifies measurement noise. A low-pass filter (e.g. moving average) on  $\omega$  before differentiation reduces this effect at the cost of bandwidth.

##### 4.1 The online form: window averages, not samples

At the 20 kHz control rate a single sample of  $i_q$  and a second difference of the encoder describe the same instant closely enough. At the 1 kHz outer-loop rate they do not. The speed difference  $\omega[n] - \omega[n-1]$  of two window-mean speeds is a triangular average of the acceleration centred one outer sample *earlier* than the instantaneous current  $i_q[n]$  it used to be regressed against. With the speed loop's transients decaying at  $\sigma \approx 400$  rad/s that misalignment scales the identified inertia by about  $e^{-\sigma T} \approx 0.67$  — the 26–33 % low reading the SIL first measured.

The online estimator therefore regresses quantities the control interrupt integrates over each outer period  $T$ . With  $\bar{\omega}_k = (\theta_k - \theta_{k-1})/T$  the window-mean speed and  $\bar{i}_{q,k}$  the window-mean current (trapezoidal over the interrupt's samples), the momentum balance integrated across two windows gives, to second order,

$$K_T \frac{\bar{i}_{q,k} + \bar{i}_{q,k-1}}{2} = J \frac{\bar{\omega}_k - \bar{\omega}_{k-1}}{T} + B \frac{\bar{\omega}_k + \bar{\omega}_{k-1}}{2} + \tau_0$$

Both sides now describe the same span, and the switching ripple is averaged out rather than sampled. Two details keep the window honest. The angle is latched *by the interrupt* at the window boundary: the outer loop can start a tick late, and differencing the angle it finds then measured 19- or 21-tick spans as if they were 20, an acceleration noise of some 2000 rad/s<sup>2</sup> that biased  $J$  toward zero exactly as the misalignment did. And the regression runs in milli-newton-metres against micro-unit  $J$  and  $B$ , so its three columns are of order one on these rotors.

## 5. Convergence Criteria

The offline identification has converged to reliable estimates when:

1. **Innovation is small:**  $|e[n]| = |\tau_e[n] - \hat{\tau}[n]| < \epsilon_e$

This indicates the model fits the data well (low residual error).

2. **Gain trace is small:**  $\text{tr}(\mathbf{K}[n]) < \epsilon_K$

This indicates the covariance has collapsed (the estimator is no longer updating significantly), i.e. the estimate has stabilised.

The two thresholds in use:  $\epsilon_e = 10^{-4}$  and  $\epsilon_K = 10^{-2}$ .

The gain trace criterion is a proxy for  $\text{tr}(\mathbf{P}[n])$  shrinking, which proves that  $\hat{\tau}[n]$  has converged to a minimum-variance estimate given the data.

Both procedures also require the observations to have included enough accelerating samples and to span more than one speed (a quarter of the fastest seen), which is what identifies  $J$  and separates  $B$  from  $\tau_0$  respectively.

The online estimator does not use the instantaneous innovation: 0.1 mN · m sits under the torque ripple any real drive has, so it was met by chance. It publishes once it has taken a thousand observations and its exponentially averaged residual power is under 1 % of the averaged output power — the fit explains the torque it sees.

## 6. Observability Requirement

RLS identifies  $J$ ,  $B$ , and  $\tau_0$  simultaneously only if the regressor matrix is **persistently exciting** of order 3 — i.e. the columns  $[\dot{\omega}, \omega, 1]$  are linearly independent over the observation window. In practice this requires:

- $\dot{\omega} \neq 0$ : the motor must accelerate and decelerate (constant-speed operation cannot identify  $J$ ).
- $\omega$  must vary over a sufficient range to separate  $B\omega$  from  $\tau_0$ .
- Both positive and negative torques should appear if possible (else  $\tau_0$  is poorly identified).

A typical identification profile: triangular speed reference — ramp up, hold briefly, ramp down — repeated  $\geq 2$  times.

---

## Block Diagrams

### RLS Signal Flow

### Effective Memory Window (Forgetting Factor)

---

## Numerical Properties

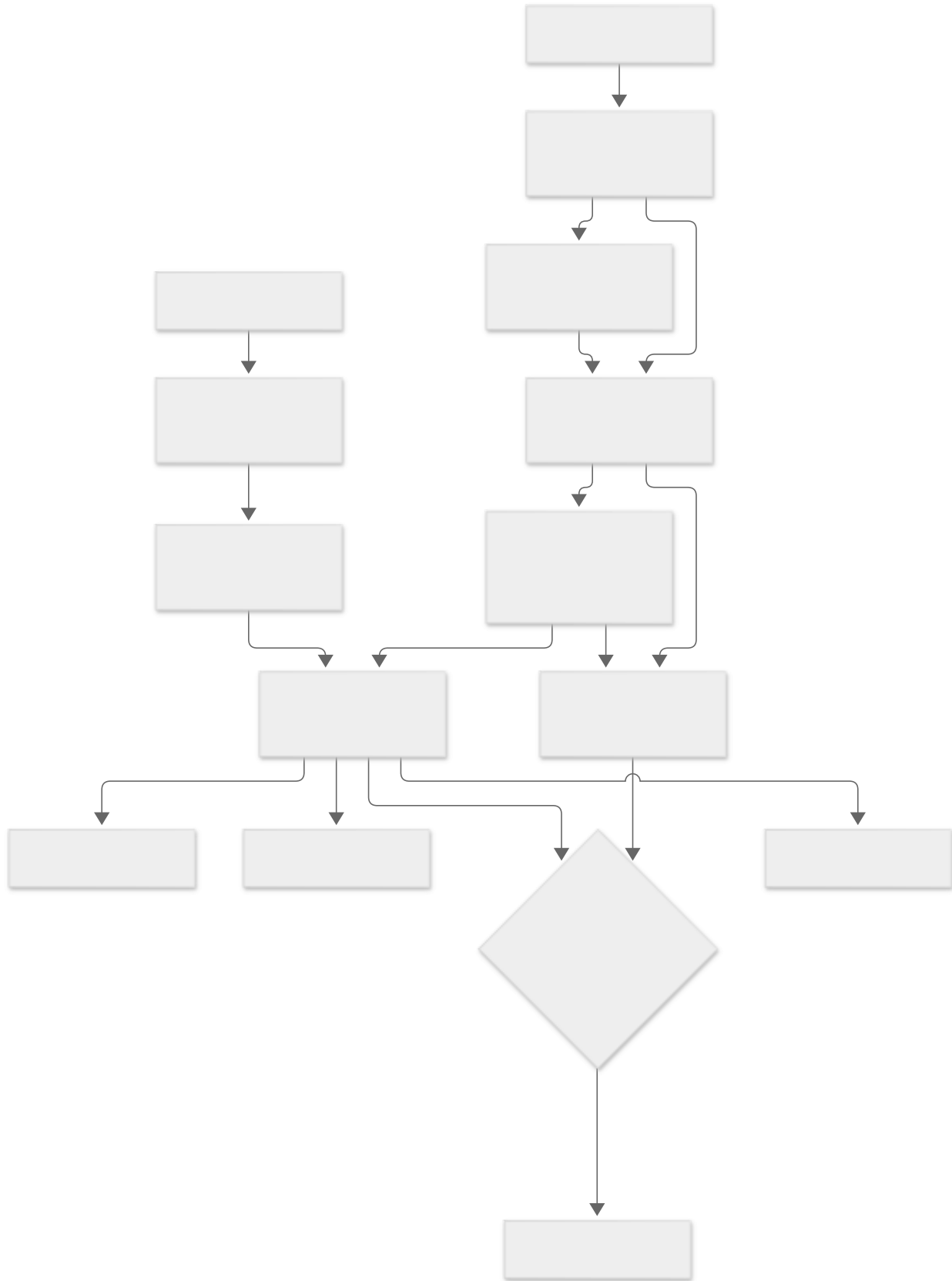


Figure 14: Block diagram

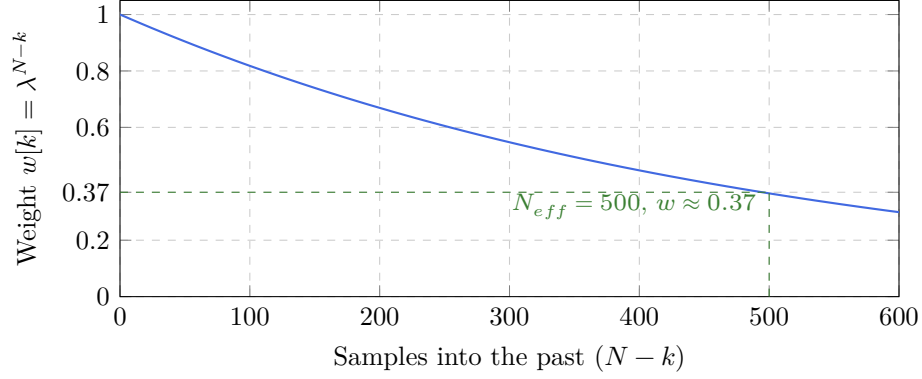


Figure 15: Exponential weight decay with  $\lambda = 0.998$ . Data older than  $N_{eff} = 1/(1 - \lambda) = 500$  samples has weight  $\leq e^{-1} \approx 0.37$ .

| Property                       | Value / Condition                                                                     |
|--------------------------------|---------------------------------------------------------------------------------------|
| Parameter vector size          | 3: $[J, B, \tau_0]^T$                                                                 |
| Covariance matrix              | $3 \times 3$ symmetric positive definite                                              |
| Forgetting factor (online)     | $\lambda = 0.9995$                                                                    |
| Effective window (online)      | $N_{eff} = 1/(1 - \lambda) = 2000$ outer-loop samples                                 |
| Convergence check $\epsilon_e$ | $10^{-4}$ (innovation threshold)                                                      |
| Convergence check $\epsilon_K$ | $10^{-2}$ (gain trace threshold)                                                      |
| RLS per-step cost              | $O(n^2) = O(9)$ — 9 multiply-add ops for $n = 3$                                      |
| Numerical stability            | $\mathbf{P}$ can become indefinite; use regularisation if $\lambda < 1$ for long runs |
| Initialisation                 | $\mathbf{P}_0 = \alpha_0 \mathbf{I}, \hat{\mathbf{0}} = \mathbf{0}$                   |

## Sensitivity Analysis

| Source of Error                | Effect                                                                  |
|--------------------------------|-------------------------------------------------------------------------|
| $K_T$ error                    | All parameters scaled proportionally ( $\tau_e$ is biased)              |
| Encoder quantisation           | Noise on $\alpha$ ; use long averaging window                           |
| Constant-speed operation       | $J$ unidentifiable ( $\dot{\omega} = 0$ makes regressor rank-deficient) |
| Very short identification time | $\mathbf{P}$ not collapsed; noisy estimates                             |
| $\lambda$ too large            | Slow convergence to parameter changes                                   |
| $\lambda$ too small            | Noisy, unstable estimates at steady state                               |

## Worked Example

Motor:  $J = 2 \times 10^{-4} \text{ kg} \cdot \text{m}^2$ ,  $B = 5 \times 10^{-4} \text{ N} \cdot \text{m} \cdot \text{s}/\text{rad}$ ,  $\tau_0 = 0.01 \text{ N} \cdot \text{m}$ ,  $K_T = 0.3 \text{ N} \cdot \text{m}/\text{A}$ ,  $f_s = 1 \text{ kHz}$ .

**Regressor at one sample** with  $\dot{\omega} = 50 \text{ rad/s}^2$ ,  $\omega = 100 \text{ rad/s}$ ,  $i_q = 2 \text{ A}$ :

$$= \begin{pmatrix} 50 \\ 100 \\ 1 \end{pmatrix}, \quad \tau_e = K_T i_q = 0.3 \times 2 = 0.6 \text{ N} \cdot \text{m}$$

**True output:**

$$T_{true} = 50 \times 2 \times 10^{-4} + 100 \times 5 \times 10^{-4} + 0.01 = 0.01 + 0.05 + 0.01 = 0.07 \text{ N} \cdot \text{m}$$

The difference  $0.6 - 0.07 = 0.53 \text{ N} \cdot \text{m}$  is the load torque  $\tau_{load}$ , which becomes the innovation residual from the estimator perspective — it will be treated as noise and averaged out over  $N_{eff}$  samples.

---

## Limitations & Assumptions

- **Assumes:**  $\tau_{load}$  is zero mean over the effective window (i.e. no sustained external load that would be attributed to  $\tau_0$  or  $J$ ).
- **Assumes:** Motor torque constant  $K_T$  is known a priori (from electrical identification).
- **Assumes:** The mechanical system is linear in  $J$ ,  $B$ ,  $\tau_0$  — nonlinear effects (e.g. position-dependent cogging, Stribeck friction) are not modelled.
- **Does not handle:** Rapidly varying  $J$  (e.g. moving load). The  $N_{eff} = 200$  sample window means changes faster than 200 ms update time are tracked with lag.
- **Does not handle:** Identification in the presence of external vibration or impact loads.

## References

1. Ljung, L. — *System Identification: Theory for the User*, 2nd ed., Prentice Hall, 1999.
2. Åström, K.J. & Wittenmark, B. — *Adaptive Control*, 2nd ed., Addison-Wesley, 1995.
3. Kim, J.-H. et al. — “Online Parameter Identification of PMSM Using Recursive Least-Squares Algorithm”, *IEEE International Electric Machines and Drives Conference*, 2007.

| Field     | Value                                            |
|-----------|--------------------------------------------------|
| Title     | Motor Alignment — Encoder Zero-Point Calibration |
| Type      | theory                                           |
| Status    | approved                                         |
| Version   | 1.0.0                                            |
| Component | service-alignment                                |
| Date      | 2025-01-01                                       |

## Overview

Motor alignment establishes the correspondence between the incremental encoder position and the electrical rotor angle  $\theta_e$ . An incremental encoder measures mechanical shaft rotation from a relative zero point that is arbitrary at power-on. FOC requires the absolute electrical angle to compute the

Clarke–Park transform correctly. Without calibration, the angle error  $\Delta\theta$  between the assumed and actual rotor position causes cross-coupling between the d and q axes, degrading torque production and potentially causing instability.

The alignment procedure forces the rotor to a known electrical position by applying a stator field at a fixed electrical angle (typically  $\theta_e = 0$ ). The encoder reading at the settled position becomes the **alignment offset**  $\theta_{offset}$ . All subsequent angle calculations subtract this offset:

$$\theta_e = p \cdot (\theta_m - \theta_{offset})$$

## Prerequisites

| Symbol            | Meaning                                                     | Unit                |
|-------------------|-------------------------------------------------------------|---------------------|
| $p$               | Number of pole pairs                                        | —                   |
| $\theta_m$        | Mechanical angle from encoder (raw counts / $2\pi$ or rad)  | rad                 |
| $\theta_e$        | Electrical angle used by FOC                                | rad                 |
| $\theta_{offset}$ | Encoder reading when rotor is at $\theta_e = 0$             | rad                 |
| $\delta$          | Misalignment angle = $\theta_e - \theta_r$ (field vs rotor) | rad                 |
| $K_T$             | Motor torque constant ( $\tau = K_T \cdot i_q$ )            | N · m/A             |
| $J$               | Rotor moment of inertia                                     | kg · m <sup>2</sup> |
| $B$               | Viscous friction coefficient                                | N · m · s/rad       |
| $N_{steps}$       | Number of alignment steps per revolution                    | —                   |

## Mathematical Foundation

### 1. Physical Principle — Reluctance Torque at Start

A PMSM permanent magnet rotor experiences a torque whenever the stator magnetic field is not aligned with the rotor d-axis. Applying a constant stator current at electrical angle  $\theta_e^{field}$  produces a stator MMF (magnetomotive force) at that electrical angle in the air gap. The interaction with the rotor's permanent magnet flux creates a synchronising torque:

$$\tau_{sync} = -K_T \cdot I_d^{field} \cdot \sin(\theta_e^{field} - \theta_r) = -K_{mag} \cdot \sin(\delta)$$

where  $\delta = \theta_e^{field} - \theta_r$  is the angle between the stator field and the rotor d-axis (the north pole of the permanent magnet), and  $K_{mag} > 0$  is a motor-dependent constant.

This is the classical restoring torque of a synchronous machine:  $\tau = -K_{mag} \sin(\delta)$ .

**Equilibrium:**  $\tau_{sync} = 0$  when  $\delta = 0$  or  $\delta = \pi$ . The stable equilibrium is  $\delta = 0$ , i.e. the rotor d-axis aligns with the applied stator field. The  $\delta = \pi$  equilibrium is unstable.

## 2. Rotor Settling Dynamics

The rotor motion during alignment is governed by the Newton-Euler equation:

$$J\ddot{\theta}_r + B\dot{\theta}_r = \tau_{sync} = -K_{mag} \sin(\theta_e^{field} - \theta_r)$$

Linearising around the equilibrium  $\theta_r \rightarrow \theta_e^{field}$  (small  $\delta$ ,  $\sin \delta \approx \delta$ ):

$$J\ddot{\delta} + B\dot{\delta} + K_{mag}\delta = 0$$

This is a damped second-order system with:

$$\omega_n = \sqrt{\frac{K_{mag}}{J}}, \quad \zeta = \frac{B}{2\sqrt{K_{mag}J}}$$

Typical PMSM rotor assemblies are underdamped ( $\zeta < 1$ ), so the rotor overshoots and oscillates before settling. The free response is:

$$\delta(t) = \delta_0 \cdot e^{-\zeta\omega_n t} \left[ \cos(\omega_d t) + \frac{\zeta}{\sqrt{1-\zeta^2}} \sin(\omega_d t) \right], \quad \omega_d = \omega_n \sqrt{1-\zeta^2}$$

See: `documentation/theory/images/alignment_settling.svg` (generated by `documentation/-tools/generate_plots.gp`)

## 3. Single-Step Alignment

The implementation applies a single fixed stator field at  $\theta_e^{field} = 0$ . The rotor d-axis is attracted to this angle and settles there from any starting position that is not at the unstable equilibrium ( $\delta = \pi$ ). In practice, static friction and motor winding asymmetry prevent the rotor from remaining at the unstable equilibrium, so the procedure reliably converges for surface-mounted PMSM and BLDC motors. For interior PMSM motors with significant magnetic saliency, a rotor that is already near the unstable equilibrium and is held by static friction may require the `settledThreshold` to be tightened and the `testVoltage` to be increased to ensure movement.

The field is maintained continuously at a constant duty cycle until the rotor is detected as settled. No stepping or sweeping is performed.

## 4. Settlement Detection

After applying the field at  $\theta_e = 0$ , the controller waits for the rotor to settle. Settlement is detected by a **consecutive-sample position-change check**:

“The rotor is considered settled when the absolute change in encoder position between successive samples is smaller than  $\epsilon_{threshold}$  for  $M$  consecutive samples.”

Formally, let  $\theta_m[n]$  be the encoder reading at sample  $n$ . Settlement is declared when:

$$|\theta_m[n] - \theta_m[n-1]| < \epsilon_{threshold}$$

holds for  $M = \text{settledCount}$  consecutive samples. If the condition is not met, the counter resets. The threshold  $\epsilon_{threshold}$  must be larger than the encoder quantisation noise but smaller than the expected motion during a half-period of the damped oscillation.

## 5. Alignment Offset Calculation

Once the motor has settled with the stator field at  $\theta_e^{field} = 0$ , the encoder reading  $\theta_m^{settled}$  corresponds to the rotor d-axis being at mechanical angle  $\theta_m^{settled}$ . This reading is recorded as the alignment offset:

$$\theta_{offset} = \theta_m^{settled}$$

During normal FOC operation the corrected electrical angle is:

$$\theta_e = p \cdot (\theta_m - \theta_{offset}) \pmod{2\pi}$$

This ensures  $\theta_e = 0$  when the rotor is at its original settled position.

## 6. Torque Production with Angle Error

If the alignment offset is incorrect by  $\Delta\theta$  (in mechanical radians), then the electrical angle error is  $p \cdot \Delta\theta$ . The effective torque is reduced:

$$\tau_{actual} = K_T \cdot i_q \cdot \cos(p \cdot \Delta\theta)$$

For example, a  $10^\circ$  mechanical offset with  $p = 4$  gives a  $40^\circ$  electrical error, reducing torque to  $\cos(40^\circ) \approx 77\%$  of its potential. The d-axis also receives unwanted excitation, potentially demagnetising the permanent magnets under sustained high current.

---

## Block Diagrams

### Alignment State Machine

### Settling Signal

---

## Numerical Properties

---

| Property            | Value / Condition                                                     |
|---------------------|-----------------------------------------------------------------------|
| Field application   | Single fixed stator field at $\theta_e^{field} = 0$ (§3); no stepping |
| Detection threshold | Position change $< \epsilon_{threshold}$ per check                    |

---

---

| Property            | Value / Condition                                                                                             |
|---------------------|---------------------------------------------------------------------------------------------------------------|
| Consecutive checks  | $M$ samples below threshold before declaring settled                                                          |
| Offset precision    | Limited by encoder resolution (e.g. $\pm 0.5$ count)                                                          |
| Torque cost         | The d-axis field produces the synchronising torque of §2; no $i_q$ is commanded                               |
| Worst-case duration | <code>AlignmentConfig::timeout</code> (default 2 s), or <code>maxSamples</code> samples if that elapses first |

---



---

## Limitations & Assumptions

- **Assumes:** The stator field is strong enough to overcome static friction and drag the rotor. For very low currents, the rotor may not follow.
- **Assumes:** The rotor reaches the stable equilibrium ( $\delta = 0$ ), not the unstable one ( $\delta = \pi$ ). Large initial misalignment or very low damping can cause incorrect convergence.
- **Assumes:** Negligible cogging torque relative to  $K_{mag} \cdot I_{align}$ . High cogging torque motors may require a higher alignment current.
- **Does not handle:** Repeated alignment without re-energising (thermal state must be considered).
- **Does not handle:** Distribution of alignment offset across multiple electrical periods for motors where  $p > 1$  and the mechanical step may span more than one electrical revolution.

## References

1. Mohan, N. — *Advanced Electric Drives: Analysis, Control and Modeling using MATLAB/Simulink*, Wiley, 2014.
2. Schroedl, M. — “Sensorless control of AC machines at low speed and standstill based on the INFORM method”, *Industry Applications Conference*, 1996.

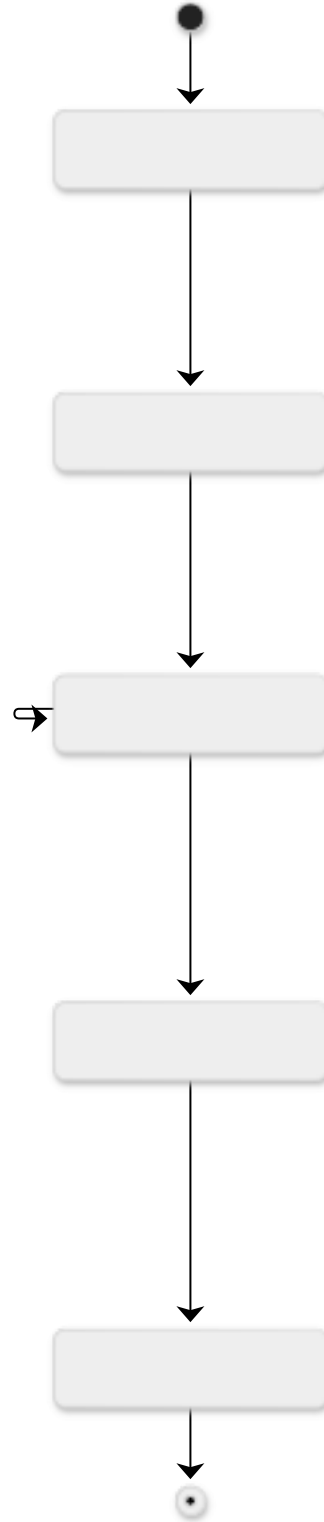


Figure 16: State diagram

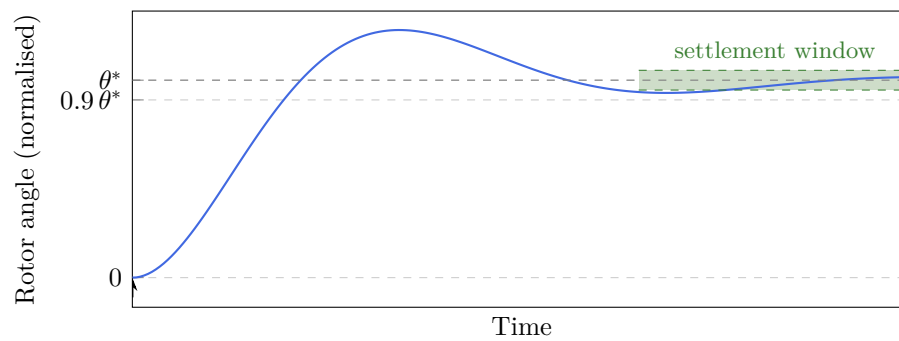


Figure 17: Under-damped settling toward target angle  $\theta^*$ . The rotor is declared settled when it stays within  $\pm 5\%$  of  $\theta^*$  for  $M$  consecutive checks.

## Part II — Architecture

---

| Field     | Value                     |
|-----------|---------------------------|
| Title     | e-foc System Architecture |
| Type      | architecture              |
| Status    | draft                     |
| Version   | 0.1.0                     |
| Component | system                    |
| Date      | 2026-04-07                |

---

**Note — Architecture-level document:** This document describes *what the system is and why it is structured this way*. Code must follow the architecture, not the opposite. Logical component and interface names are the vocabulary of architecture and belong here. What must be avoided: source file paths, implementation class names (e.g. `FooImpl`), and code-level algorithmic details.

**Diagrams:** All visuals must be either a Mermaid fenced code block or ASCII art inline in the document. External image references (`![alt](/home/runner/work/e-foc/e-foc/documentation/architecture/path)`) are **not allowed**.

Sequence diagrams, block diagrams, component diagrams, and state machines (Mermaid syntax) are allowed and encouraged.

---

## Assumptions & Constraints

- **Constraint:** No dynamic memory allocation on the embedded target. All objects are stack- or statically-allocated for deterministic, bounded memory footprint.
- **Constraint:** The core FOC loop executes in an interrupt service routine at 20 kHz. The complete control cycle must not exceed 75 percent of the available CPU time per control period ( $\leq 4500$  cycles at 120 MHz / 20 kHz) to guarantee real-time deadlines.
- **Constraint:** No recursion in control-loop or ISR-reachable paths. Stack usage must be statically predictable.
- **Constraint:** No C++ exceptions. Error signalling uses status return values, `std::optional`, or asynchronous callbacks.
- **Constraint:** The system targets 32-bit ARM Cortex-M microcontrollers (STM32 and TI Tiva families). Host builds are supported for unit testing and simulation only.
- **Assumption:** A single motor is controlled per instantiation. Multi-motor configurations require multiple independent instantiations.
- **Assumption:** The motor is a BLDC or PMSM (surface or interior permanent magnet) with a sinusoidal back-EMF profile suitable for FOC.
- **Assumption:** Rotor position is measured by a quadrature encoder. Hall-sensor support is defined in the driver interface but is a secondary configuration.
- **Assumption:** All motor electrical and mechanical parameters (resistance, inductance, pole pairs, inertia, friction) can be identified at startup or loaded from non-volatile storage.
- **Constraint:** The build system produces a single unified binary (`e_foc.sync_foc_sensored.main`) that contains all three control modes. The active mode can be selected at runtime via CAN command (`SelectControlMode`) and persisted to NVM so it survives a power cycle. The

CLI command `active_mode` reports the currently active mode. No mode switch is permitted while the motor is enabled.

---

## System Overview

e-foc is a Field-Oriented Control (FOC) firmware for BLDC and PMSM motors. Its purpose is to transform three-phase stator currents into a rotating reference frame so that torque and flux can be regulated independently with standard linear PIDs, then synthesise the three-phase PWM voltages via Space Vector Modulation.

The system is structured in four tiers:

1. **Infrastructure tier** — external repositories providing general-purpose utilities, numerical algorithms, and vendor HAL drivers. These are consumed as submodules and are never modified by this project.
2. **Core / Library tier** — the FOC algorithm library (`core/foc/`), service library (`core/services/`), and the Platform Abstraction Layer interface (`core/platform_abstraction/`). These are pure, portable libraries with no target-specific knowledge.
3. **Targets tier** — target-specific code that is NOT reusable across all environments: platform implementations (concrete `PlatformFactory` for TI, ST, and Host) and application entry points (`hardware_test`, `sync_foc_sensored`). Lives under `targets/`.
4. **Tools tier** — host-only desktop tools (simulator, CAN commander) that run on development machines only. Lives under `tools/`.

The simulator implements the same PAL contracts as real hardware. Control logic never needs to distinguish between running on a microcontroller or on the host; this enables full-fidelity closed-loop validation without hardware.

---

## Component Decomposition

### 1. FOC Core

The heart of the system. Decomposed into sub-layers following a strict separation of vocabulary, generic numerics, transform algorithms, orchestration, and wiring. Dependencies flow in one direction only — from wiring down towards vocabulary; no lower layer may depend on a higher one:

---

| Sub-component | Responsibility                                                                                                                                                                                                                                                               |
|---------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Interface     | Define the FOC vocabulary (units, phase and frame signal types) and the abstract contracts for control modes (Torque, Speed, Position) and execution (low-priority interrupt, controllable). No algorithms. Hardware ports live in the platform abstraction layer, not here. |
| Math          | Generic numerical helpers that are not specific to field-oriented control — fast trigonometry by lookup table, angle wrap-around. Header-only, dependency-free.                                                                                                              |

---

## Sub-component Responsibility

---

|                 |                                                                                                                                                                                             |
|-----------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Transforms      | The field-oriented control mathematics proper: the Clarke and Park transforms in both directions, and Space Vector Modulation.                                                              |
| Loop algorithms | Interchangeable control laws for each loop, under <code>current_loop/</code> , <code>speed_loop/</code> and <code>position_loop/</code> , together with their plant models and gain design. |
| Selection       | The <code>ControllerSelector</code> template that stores the active algorithm of a loop in a fixed-size <code>std::variant</code> and dispatches to it without allocation.                  |
| Cascade         | Orchestration of the control cascade — the per-mode control loops that combine the current, speed and position algorithms.                                                                  |
| Instantiation   | Wrapping that combines a control-mode cascade with the execution runner to produce a ready-to-use FOC controller.                                                                           |

---

Because Interfaces, Math, and Transforms carry no dependency on the control cascade, the identification, alignment, and simulation components consume them directly without pulling in the control loops.

The three control modes are available in the same binary and are selected at runtime through `ControlModeStateMachine`. Only one mode is active at a time; switching is only permitted from `Idle` or `Ready` state and persists the new choice to NVM via `defaultControlMode`:

- **Torque control** — innermost loop: regulates phase currents to produce a commanded torque (specified as Id/Iq current setpoints).
- **Speed control** — outer loop on top of torque control: a PID regulates rotor angular velocity by commanding Id/Iq setpoints to the inner loop. The outer loop runs at a lower priority interrupt (typically 1 kHz) while the inner loop still runs at 20 kHz.
- **Position control** — outermost cascade: a PID regulates rotor position by commanding a speed setpoint to the speed loop. Builds on speed control.

## Interrupt Priority Plan

On Cortex-M, equal priority means no preemption: a handler at the same level as the FOC ISR cannot be interrupted by it, so the FOC loop's worst-case latency is the duration of the longest other handler at or above its level. Leaving every source at the default therefore leaves the 20 kHz deadline hostage to a CAN receive or a UART DMA completion.

The TI target assigns priorities explicitly in `application::InterruptPriorities`:

---

| Source              | Priority       | Rationale                                                                                                        |
|---------------------|----------------|------------------------------------------------------------------------------------------------------------------|
| Phase-current ADC   | <b>highest</b> | Carries the 20 kHz control deadline; must preempt everything else                                                |
| PWM fault           | <b>highest</b> | Hardware overcurrent/overvoltage trip; nothing may delay the response                                            |
| CAN                 | <b>normal</b>  | Command and telemetry traffic; may be preempted by the control loop                                              |
| UART                | <b>low</b>     | Terminal and tracing; the least time-critical source on the board                                                |
| PendSV (outer loop) | <b>lowest</b>  | Set by <code>hal::cortex::LowPriorityInterrupt</code> ; the 1 kHz outer loop must never preempt the current loop |

---

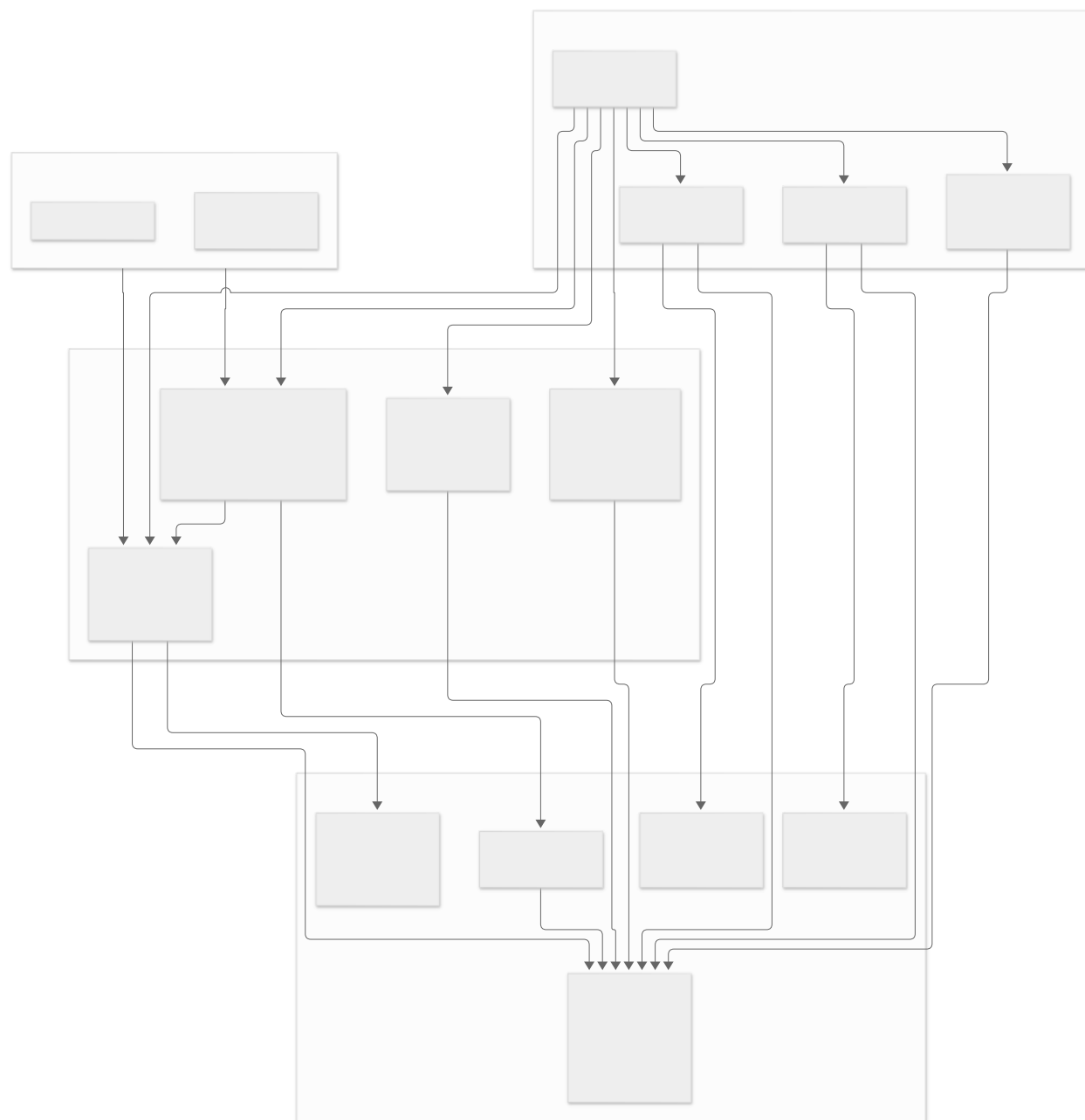


Figure 18: Block diagram

The outer loop runs on PendSV at the lowest configurable priority. It carries a 20 000-cycle budget against the inner loop's 4 500, so running it above the current loop would insert up to a full outer-loop execution of duty-update jitter once per millisecond.

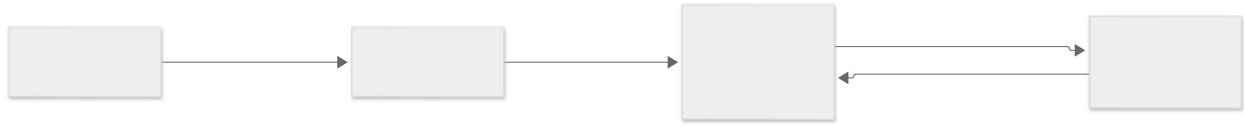


Figure 19: Block diagram

The **Runner** is the only component that interacts with the PAL inverter and encoder at interrupt time. It registers the ADC-sampling callback and drives the `Calculate()` dispatch into the active control-mode implementation.

The dispatch carries no validation of the sample. Nothing in the 20 kHz path inspects the phase currents for plausibility or for finiteness: at 4500 cycles a frame the budget is spent on the control law, and a failed shunt or ADC channel is the hardware's fault detection to catch, not the control loop's. `foc::ToDutyCycles` clamps each modulator output into  $[0, 1]$  and converts to percent, and does nothing else.

`foc::IsFiniteValue` survives for the places where validation is affordable and worth having — `CurrentPlantModel::IsUsable` when the plant is configured, and the plausibility band the mechanical estimator applies before publishing. Both run outside the hot path. It tests the float's exponent bits rather than calling `std::isnan`, which `-ffinite-math-only` licenses the compiler to fold to a constant; the embedded builds keep `-fno-finite-math-only`, but the check does not depend on it.

## 2. Services

Higher-level, non-real-time services that support commissioning and runtime operation. Each service is independently usable:

| Service                          | Responsibility                                                                                                                                              |
|----------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Alignment                        | Forces a known electrical angle on the rotor at startup so the FOC reference frame is correctly initialised before normal operation.                        |
| Electrical System Identification | Estimates phase resistance, d/q inductances, and pole pairs by injecting test signals and measuring the response.                                           |
| Mechanical System Identification | Estimates rotor inertia and viscous friction coefficient from closed-loop speed response data.                                                              |
| Non-Volatile Memory (NVM)        | Persists calibration data (R, L, pole pairs, encoder offset, PID gains) and configuration (including <code>defaultControlMode</code> ) across power cycles. |

| Service      | Responsibility                                                                                                                                                                                                                                                                                               |
|--------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| CLI          | A terminal-based command interface for triggering services, querying state, and setting parameters from a serial console.                                                                                                                                                                                    |
| Control Mode | Owns the three <code>FocStateMachine</code> instances (torque, speed, position) as a <code>std::variant</code> ; only one is live at a time. Exposes <code>Select(mode, onDone)</code> which enforces the motor-stopped precondition, persists the selection to NVM, then switches the active state machine. |

### 3. CAN Service

The CAN service layer (`core/can/`) provides the FOC motor CAN category as a server/client pair and the bridge that connects the server to the `ControlModeStateMachine`. It is split into two halves:

**Server half** (runs on the embedded target): - `FocMotorCategoryServer` — a `CanCategoryServer` subclass. Decodes incoming CAN frames into typed observer callbacks. Encodes typed results back into response frames. Wire scaling (`int16` ↔ physical unit) is contained entirely here. - `FocMotorCanBridge` — implements the server observer interface. Translates CAN motor commands (`Start`, `Stop`, `SelectControlMode`, `SetTorqueSetpoint`, `SetSpeedSetpoint`, `SetPositionSetpoint`, `ClearFault`, `EmergencyStop`) into the corresponding `ControlModeStateMachine` calls. It also implements the PID bandwidth, identification, telemetry, and encoder-resolution commands (`OnIdentifyElectrical`, `OnIdentifyMechanical`, `OnRequestTelemetry`, `OnSetEncoderResolution`, `OnConfigureTelemetryRate`), each reporting success or a specific `FocMotorCategoryError` on the response frame.

**Client half** (runs on the desktop host — CAN Commander, SIL tests): - `FocMotorCategoryClient` — a `CanCategoryClient` subclass. Encodes typed setpoints into CAN frames and decodes response frames into typed callbacks. - `FocMotorCanClient` — user-facing API that composes `CanFrameTransport`, `CanProtocolClient`, and `FocMotorCategoryClient` internally. Takes a `hal::Can&` and a node ID; exposes `Start`, `Stop`, `SetTorque`, `SetSpeed`, and `SetPosition` methods.

Tracing is applied at every layer boundary via the `can-lite` tracing decorators (`TracingCan`, `TracingCanProtocolServerObserver`, `TracingCanProtocolClientObserver`).

Services communicate via asynchronous callbacks (zero-allocation closures from `embedded-infra-lib`), not return values. This allows long-running operations (identification, alignment) to yield the CPU and complete asynchronously without blocking.

### 3. Platform Abstraction Layer

The PAL provides a single platform-facing abstraction that groups creation and access to the hardware services needed by the control system:

| Peripheral                                  | Abstraction                                                   |
|---------------------------------------------|---------------------------------------------------------------|
| Three-phase PWM + ADC triggered measurement | Power-stage drive and synchronised current-sampling interface |
| Quadrature encoder                          | Rotor-position sensing and calibration interface              |

| Peripheral             | Abstraction                                                     |
|------------------------|-----------------------------------------------------------------|
| Low-priority interrupt | Deferred scheduling interface for the speed/position outer loop |
| CAN bus                | CAN 2.0B communication interface                                |
| Performance timer      | Cycle/timestamp measurement interface for profiling             |
| Serial terminal        | Diagnostic trace and command-line interaction interface         |

Concrete implementations exist for: - **TI Tiva (EK-TM4C1294XL, EK-TM4C123GXL)**: platform-specific peripheral adapters under `targets/platform_implementations/ti/`. - **ST STM32 (STM32F407G-DISC1, NUCLEO-H563ZI)**: platform-specific peripheral adapters under `targets/platform_implementations/st/`. - **Host / Simulator**: a software-backed platform implementation that emulates the motor-control I/O needed to run the closed-loop algorithm on a development machine, located under `targets/platform_implementations/host/`.

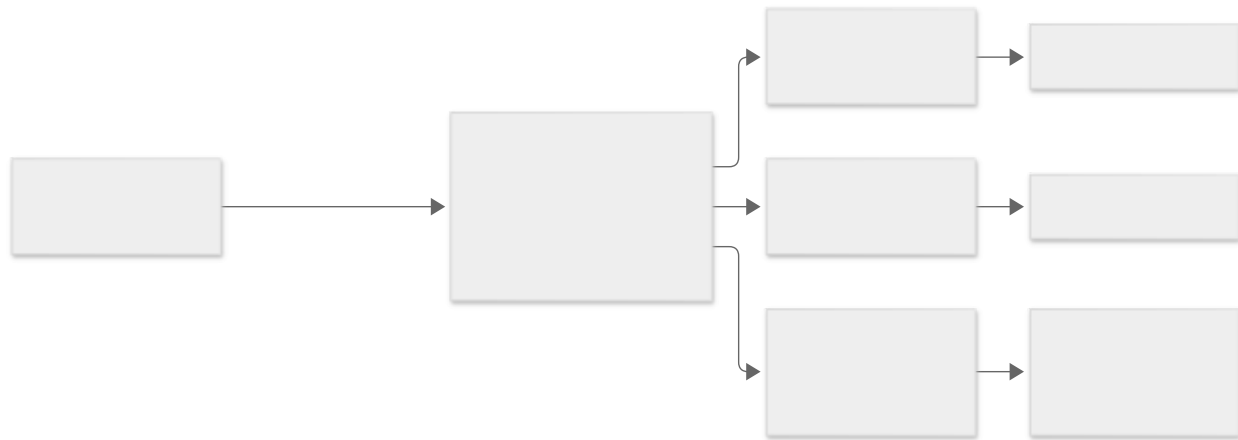


Figure 20: Block diagram

#### 4. Application / Instantiation (was §3)

The wiring layer. Assembles the concrete PAL implementation, the selected FOC mode(s), and the services into a runnable system. This is the only layer that is aware of the specific combination in use — all other layers depend only on abstractions.

The **FocStateMachine** is the central lifecycle authority. It owns the full motor commissioning and operation lifecycle: it enforces a formal five-state machine (**Idle** → **Calibrating** → **Ready** ⇌ **Enabled**, **Fault**), orchestrates the sequential calibration chain (electrical identification, alignment, and mechanical identification for speed/position modes), and responds to hardware fault notifications by immediately stopping the inverter and entering the **Fault** state. Only after **FocStateMachine** has reached **Ready** can the motor be enabled.

In CLI mode, **ControlModeStateMachine** registers the lifecycle commands `calibrate`, `enable`, `disable`, `clear_fault`, `clear_cal`, `apply_estimates`, and `active_mode` on the terminal. These commands delegate to the currently active state machine. The **TerminalFocBaseInteractor** (and its control-mode subclasses) register PID tuning and setpoint commands on the same terminal.

## 5. Tools (was §4)

Host-only tools that do not run on the embedded target:

| Tool            | Responsibility                                                                                                                                                                                   |
|-----------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Simulator       | Closed-loop software simulation: real FOC control code drives a physics-based PMSM model (Euler integration of the dq electrical equations). Used for validating control loops without hardware. |
| CAN Com-mander  | Desktop application for sending CAN commands and logging motor telemetry.                                                                                                                        |
| Hardware Bridge | Desktop client that bridges the target's terminal (serial) and CAN bus to the host; the hardware-in-the-loop runner reaches the board through it.                                                |

## 6. Infrastructure (was §5)

External repositories consumed as Git submodules. This project does not modify them.

| Submodule                            | Purpose                                                                                                                                                                                                                                                                                                                              |
|--------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>infra/embedded-lib</code>      | Heap-free C++ infrastructure: bounded containers ( <code>BoundedVector</code> , <code>BoundedString</code> , <code>BoundedDeque</code> ), <code>infra::Function&lt;&gt;</code> (zero-allocation closures), <code>infra::Observer/Subject</code> (type-safe observer pattern), timers, memory utilities, and build toolchain helpers. |
| <code>infra/numerical-toolbox</code> | Numerical algorithms for control: incremental PID controllers with anti-windup, digital filters (FIR, IIR, Kalman), recursive least-squares estimators, and compiler-optimisation helpers ( <code>OPTIMIZE_FOR_SPEED</code> ).                                                                                                       |
| <code>infra/can-lite</code>          | Lightweight CAN 2.0B protocol stack: client-server model, category-based message dispatch, ISO-TP segmentation. Zero heap allocation.                                                                                                                                                                                                |

### Upstream dependencies of the position state feedback laws

Because submodules are never patched locally, an upstream defect has to be carried here as a workaround until the fix lands. Two such defects shaped the position loop and are now resolved on `numerical-toolbox` main, which is what the submodule tracks:

| Upstream                                                                                        | Why it mattered                                                                                                                                                                                |
|-------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| PR <b>#288</b> / <b>#289</b> — scale-invariant DARE convergence tolerance                       | Without it the position LQR and LQI designs do not converge at all. The submodule must stay at or past this commit.                                                                            |
| Issue <b>#291</b> — <code>IntegralStateFeedbackLqi</code> had no non-aborting construction path | Its constructor called <code>really_assert(converged)</code> , so a non-convergent solve aborted the firmware. <code>TryCreate</code> now exists and every state feedback design here uses it. |

Should a comparable defect appear again, record it in this table with the workaround and the condition for retiring it, so a workaround does not quietly become permanent.

## 7. Vendor HAL (was §6)

Vendor-provided hardware abstraction libraries consumed as Git submodules. They supply the low-level peripheral register access and interrupt management that the PAL concrete implementations use.

| Directory                 | Vendor / Board                      |
|---------------------------|-------------------------------------|
| <code>infra/hal/ti</code> | Texas Instruments Tiva (Cortex-M4F) |
| <code>infra/hal/st</code> | STMicroelectronics STM32            |

These are never used directly by the FOC core or services — only by the PAL concrete implementations.

## Interfaces & Contracts

### Provided Interfaces (exported by this system)

| Interface                | Directory             | Purpose                                                                                                      | Invariants                                                                                                                                           |
|--------------------------|-----------------------|--------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>FocTorque</code>   | <code>torque</code>   | Provides torque control mode — accepts Id/Iq current setpoints, yields PWM duty cycles each FOC cycle        | Must be called from the PWM/ADC interrupt context only. <code>Calculate()</code> must return within the worst-case cycle budget.                     |
| <code>FocSpeed</code>    | <code>speed</code>    | Provides speed control mode — accepts an angular-velocity setpoint in rad/s, cascades into the torque loop   | Outer loop runs at a separate lower-priority interrupt. <code>OuterLoopFrequency()</code> must be queried by the caller to configure that interrupt. |
| <code>FocPosition</code> | <code>position</code> | Provides position control mode — accepts a rotor angle setpoint in radians, cascades into speed, then torque | Requires speed loop to be configured and running.                                                                                                    |
| <code>ControlLib</code>  | <code>lib</code>      | Provides Start/Stop lifecycle for a <code>FocController</code>                                               | <code>Start()</code> arms the interrupt-driven loop. <code>Stop()</code> disarms it and leaves the motor coasting.                                   |

### Required Interfaces (consumed from the PAL)

| Interface               | Directory           | Purpose                                                                          | Invariants                                                                                                                         |
|-------------------------|---------------------|----------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------|
| <code>ThreePhase</code> | <code>bridge</code> | ADC phase-current sampling and applies PWM duty cycles to the three-phase bridge | <code>PhaseCurrentsReady()</code> installs the callback invoked by the ADC interrupt. Must be called before <code>Start()</code> . |
| <code>Encoder</code>    | <code>rotor</code>  | Reads rotor mechanical angle and supports zero-offset calibration                | Read must be non-blocking and complete in $\leq$ a few cycles.                                                                     |

| Interface               | Direction | Purpose                                                                                          | Invariants                                                                                                                                                                                                                                              |
|-------------------------|-----------|--------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| LowPriorityFocInterrupt | Handler   | periodic execution of the speed and position outer loops at a lower rate than the FOC inner loop | <b>Register()</b> installs the callback. <b>Trigger()</b> is called from the FOC inner loop at the configured prescale ratio. <b>Unregister()</b> detaches it: after it returns, neither an already queued nor a future trigger may invoke the handler. |
| NonVolatileMemory       | Interface | Stores and retrieves calibration and configuration data                                          | All operations are asynchronous and invoke a callback on completion.                                                                                                                                                                                    |

## SOLID Principles Applied

The interface design reflects explicit SOLID choices:

- **Single Responsibility:** Each interface (**FocTorque**, **FocSpeed**, **FocPosition**, **ThreePhaseInverter**, **Encoder**, **NonVolatileMemory**, ...) is narrowly scoped to one concern. No interface carries unrelated responsibilities.
- **Open/Closed:** New control modes are added by implementing a new derived control interface, not by modifying existing implementations. The PAL is similarly extended by creating a new platform factory implementation for a new board without touching the core.
- **Liskov Substitution:** Every FOC controller instantiation (torque, speed, position) is fully substitutable wherever the base control interface is expected. The simulator's motor model is fully substitutable for the real PAL hardware.
- **Interface Segregation:** **FocBase** carries only the minimal common contract. Callers that need only torque control depend on **FocTorque**, not on **FocSpeed** or **FocPosition**. The PAL is similarly split — callers that only need an encoder do not depend on the PWM interface.
- **Dependency Inversion:** The FOC core and all services depend exclusively on abstract interfaces. Concrete implementations are injected at application wiring time via the platform factory and constructor arguments. No global variables; no direct peripheral access from the control layer.

## Data Flow

The primary data flow during closed-loop motor control is the 20 kHz FOC interrupt cycle:

The commissioning data flow (run once at startup or on command):

## Observer and Callback Patterns

The system uses two complementary asynchronous communication patterns, both supplied by `infra/embedded-infra-lib`:

- **Zero-allocation closures** (from `embedded-infra-lib`) — used for one-shot or single-subscriber callbacks. All service operations (NVM, identification, alignment) deliver their results through this mechanism.

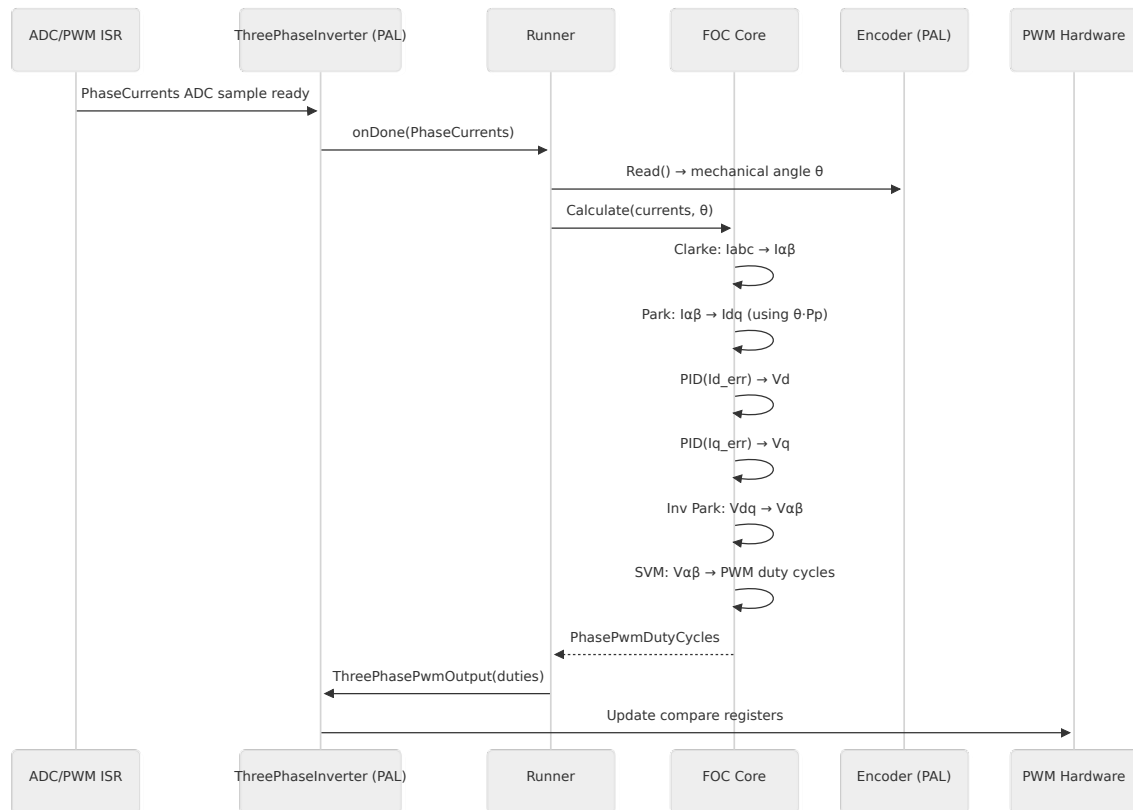


Figure 21: Sequence diagram

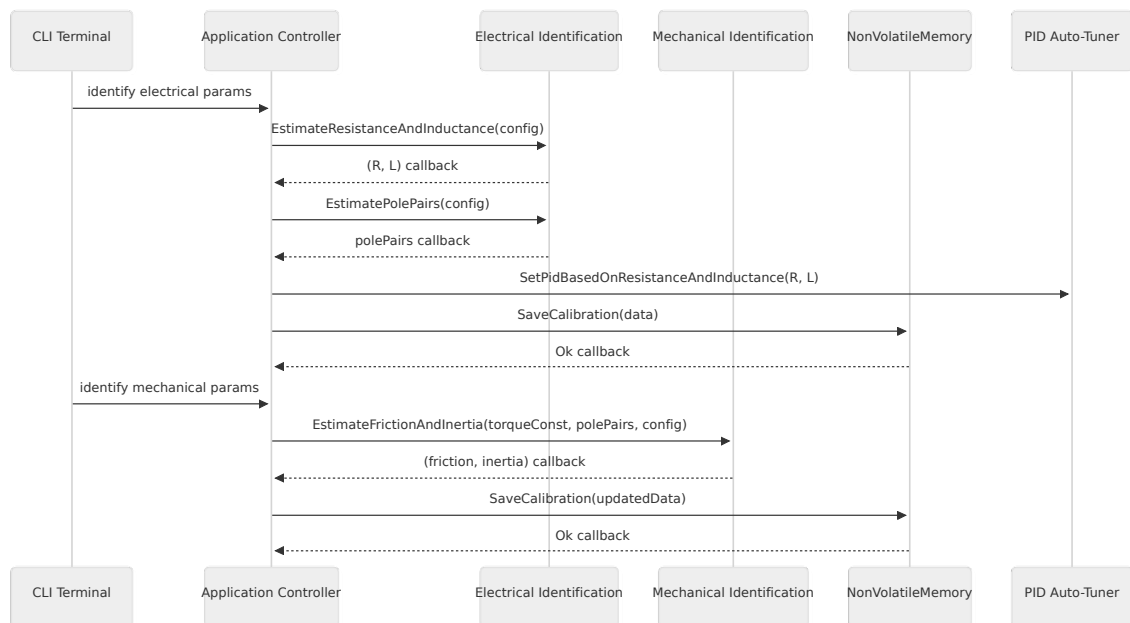


Figure 22: Sequence diagram

- **Observer/Subject pattern** (from `embedded-infra-lib`) — a type-safe, heap-free mechanism where a subject notifies multiple observers. The simulator’s motor model is a subject; views and loggers attach as observers. No raw event buses or dynamic dispatch lists are used.

These patterns eliminate polling, decouple producers from consumers, and allow the application to remain single-threaded and non-blocking.

---

## Cross-Cutting Concerns

| Concern               | Policy / Approach                                                                                                                                                                                                                                                        |
|-----------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Memory                | Absolutely no heap on the embedded target. All objects are statically or stack-allocated. Bounded containers from <code>embedded-infra-lib</code> replace STL heap-based containers. Host-side tools and tests may use the standard heap freely.                         |
| Real-time determinism | The FOC <code>Calculate()</code> path contains no virtual dispatch, no blocking calls, and no unpredictable branches. The outer loops run at lower-priority interrupts on a deterministic prescale ratio.                                                                |
| Unit safety           | All physical quantities use typed unit aliases ( <code>Ampere</code> , <code>Radians</code> , <code>Volts</code> , <code>RadiansPerSecond</code> , ...) from <code>embedded-infra-lib</code> . Raw floating-point with no unit context is not used for motor quantities. |
| Compiler optimisation | Embedded builds compile all of e-foc with one option set: the configuration’s level plus <code>-ffast-math -fno-finite-math-only</code> . No per-function pragmas; they block inlining. Debug builds use <code>-Og</code> for debuggability.                             |
| Error handling        | No C++ exceptions. Synchronous errors use <code>std::optional</code> (absent = error). Asynchronous errors are delivered as typed status codes in callbacks.                                                                                                             |
| Portability           | The control core has no knowledge of the underlying MCU. Portability to a new board is achieved by implementing a platform factory for that board and providing the PAL concrete implementations.                                                                        |
| Testability           | All modules depend on abstract interfaces, enabling host-build unit tests with mock or stub implementations. The simulator provides closed-loop integration testing without hardware.                                                                                    |

---



---

## Open Questions & Decisions

| Question /<br># Decision                                            | Status   | Options Considered                                                                                                              | Rationale                                                                                                                                                                                                                                                                                                                                             |
|---------------------------------------------------------------------|----------|---------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1 Rename <code>source/hardware/</code> and extract platform targets | resolved | Keep current name vs rename to PAL vs full extraction                                                                           | Renamed to <code>core/platform_abstraction/</code> (interface + adapters only). Platform implementations (TI, ST, Host) and application targets moved to top-level <code>targets/</code> . Host-only tools moved to top-level <code>tools/</code> . Core tier is now purely libraries. <code>source/</code> directory renamed to <code>core/</code> . |
| 2 Multi-motor support                                               | open     | Single instantiation per motor vs shared PAL with multiple FOC controllers                                                      | Current architecture supports one motor per binary. A shared PAL with multiple <code>FocController</code> instances is architecturally feasible but not yet required.                                                                                                                                                                                 |
| 3 Field-weakening                                                   | open     | Extend <code>FocTorque</code> interface with flux-weakening setpoint vs separate <code>FocTorqueFieldWeakening</code> interface | Required for operation above base speed. Separate interface preferred (ISP) but not yet scoped.                                                                                                                                                                                                                                                       |

## Integration Testing

Integration testing drives the **real firmware** as a black box over the product's own CAN command set. Nothing inside the firmware is mocked or reached into; the suite starts a target, sends commands, and asserts on telemetry and traces.

Two targets implement scenarios, selected by tag:

1. **Software-in-the-loop** — the firmware built for an emulated Cortex-M4, running under an emulator against a simulated three-phase motor the firmware itself owns. Only the motor, the power stage and the board protection are simulated, because there is no hardware. The motor is described per scenario and handed to the firmware at boot, and non-volatile memory is pre-seeded so a scenario starts from a chosen calibration and configuration.
2. **Hardware-in-the-loop** — the same firmware on a real board, reached through the hardware bridge over serial and CAN.

Design detail: `documentation/design/integration-testing.md` for the shared structure, and `documentation/design/software-in-the-loop.md` for the simulated target.

## Integration Boundaries

| Boundary              | Software-in-the-loop                    | Hardware-in-the-loop    |
|-----------------------|-----------------------------------------|-------------------------|
| Motor and power stage | Simulated plant, described per scenario | Real motor and inverter |

---

| Boundary                                                    | Software-in-the-loop                                   | Hardware-in-the-loop       |
|-------------------------------------------------------------|--------------------------------------------------------|----------------------------|
| Board protection                                            | Simulated comparators, armed with the inverter         | Real comparators           |
| Non-volatile memory                                         | Host-written image, pre-seeded or deliberately damaged | Real device                |
| State machine, cascades, calibration, NVM stack, CAN server | Real firmware                                          | Real firmware              |
| Transport                                                   | CAN frames as text on the emulated UART                | Bridge over serial and CAN |

---

## Requirements Traceability

Each scenario carries the requirement identifiers it verifies as tags, alongside the tag naming the target that implements it. The traceability matrix is generated from those tags at release time by reading the feature directory, so the mapping lives with the scenarios rather than in a table here that would drift away from them.

Requirements with no scenario are verified by unit tests under the owning library, or are structural constraints discharged by design review; the generated matrix reports both.

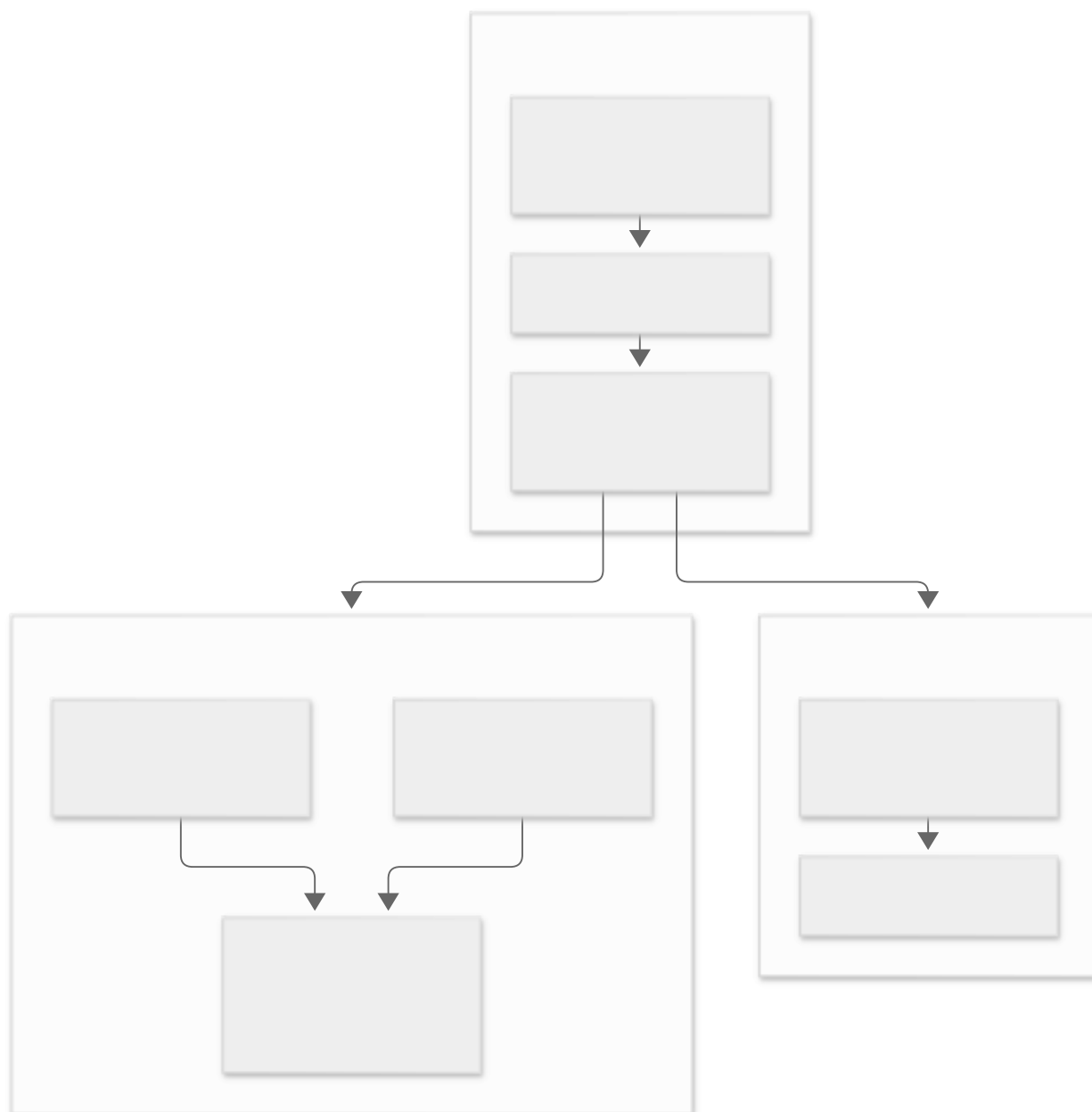


Figure 23: Block diagram

## Part III — Design

---

| Field     | Value                       |
|-----------|-----------------------------|
| Title     | FOC Mathematical Transforms |
| Type      | design                      |
| Status    | draft                       |
| Version   | 0.1.0                       |
| Component | foc-transforms              |
| Date      | 2026-04-07                  |

---

**IMPORTANT — Implementation-blind document:** This document describes *behavior, structure, and responsibilities* WITHOUT referencing code. **No code blocks using programming languages (C++, C, Python, CMake, shell, etc.) are allowed.** Use Mermaid diagrams to express behavior instead. Prose descriptions of algorithms are encouraged; source-level details are not.

**Diagrams:** All visuals must be either a Mermaid fenced code block (````mermaid`) or ASCII art inline in the document. External image references using Markdown image syntax are **not allowed**.

---

## Responsibilities

**Is responsible for:** - Converting three-phase stator currents to a two-phase stationary frame (Clarke transform) - Converting between the stationary  $\alpha\beta$  frame and the rotor-synchronous dq frame (Park transform) - Providing a composite Clarke  $\rightarrow$  Park (and inverse) operation that evaluates trigonometric functions exactly once per call - Converting a two-component voltage vector in the  $\alpha\beta$  frame to three-phase PWM duty cycles (Space Vector Modulation) - Supplying fast sine and cosine approximations via a pre-computed lookup table for use in the FOC hot path

**Is NOT responsible for:** - Reading or writing hardware peripherals (ADC, PWM, encoder) - Managing PID controllers or setpoints - Determining rotor position — electrical angle must be supplied by the caller - Any control strategy decisions (torque, speed, or position setpoints) - Scaling voltage vectors to physical units — inputs and outputs are normalised fractions relative to the DC bus

---

## Component Details

### Clarke Transform — Three-Phase to Stationary $\alpha\beta$ Frame

The Clarke transform is a geometric projection that reduces a balanced three-phase system to an equivalent two-axis representation. It requires no knowledge of rotor position.

**Forward transform** (currents 3-phase  $\rightarrow \alpha\beta$ ):

The amplitude-invariant Clarke transform uses all three measured phases, so it stays correct when the currents are not perfectly balanced:

$$I_\alpha = (2/3) \cdot (I_a - (I_b + I_c)/2)$$

$$I_\beta = (I_b - I_c) / \sqrt{3}$$

When the system *is* balanced ( $I_a + I_b + I_c = 0$ ) this reduces to the familiar two-sensor form  $I_\alpha = I_a$ ,  $I_\beta = (I_a + 2 \cdot I_b) / \sqrt{3}$ . The implementation uses the three-phase form unconditionally.

All three phase currents are measured. `AdcPhaseCurrentMeasurement` converts three ADC samples and delivers them as a `PhaseCurrents` triple, so no phase is reconstructed from the balance constraint.

**Inverse transform** (voltages  $\alpha\beta \rightarrow 3$ -phase):

Reconstructs the three-phase voltage references from  $V_\alpha$  and  $V_\beta$ , using the same geometric relationships in reverse. The output of the inverse Clarke is fed into the Space Vector Modulator.

**Invariants:** - No balance assumption is required on the input; a nonzero common-mode current appears only in the zero-sequence component, which the two-axis output discards. - The transform is purely linear and involves no state — every call is independent.

### Park Transform — Stationary $\alpha\beta$ to Rotor-Synchronous dq Frame

The Park transform rotates the stationary  $\alpha\beta$  frame to align with the rotor magnetic field. The result is a coordinate frame that is stationary relative to the rotor, so DC steady-state values represent constant torque and flux components.

**Forward transform** ( $\alpha\beta \rightarrow dq$ ):

$$I_d = I_\alpha \cdot \cos(\theta_e) + I_\beta \cdot \sin(\theta_e)$$

$$I_q = -I_\alpha \cdot \sin(\theta_e) + I_\beta \cdot \cos(\theta_e)$$

The d-axis (direct) current component is aligned with the rotor flux. For a surface-permanent-magnet synchronous motor (SPMSM), the d-axis setpoint is held at zero to maximise torque per ampere. The q-axis (quadrature) current is proportional to electromagnetic torque.

**Inverse transform** ( $dq \rightarrow \alpha\beta$ ):

$$V_\alpha = V_d \cdot \cos(\theta_e) - V_q \cdot \sin(\theta_e)$$

$$V_\beta = V_d \cdot \sin(\theta_e) + V_q \cdot \cos(\theta_e)$$

The inverse Park transform is applied after the PID controllers produce voltage demands  $V_d$  and  $V_q$ , converting them back to the stationary frame for SVM processing.

**Electrical angle:**

The electrical angle  $\theta_e$  is obtained by multiplying the mechanical rotor angle  $\theta_m$  by the motor's pole-pair count  $P$ :

$$\theta_e = \theta_m \cdot P$$

The caller is responsible for supplying the correct  $\theta_e$ .

**Invariants:** - The same  $\theta_e$  (and therefore the same cos/sin pair) must be used for both the forward Park in the current-sensing path and the inverse Park in the voltage-output path within a single control cycle. - The transform is stateless — no memory of previous cycles.

## Composite ClarkePark — Chained Transform with Single Trigonometric Evaluation

The ClarkePark composite combines the forward Clarke and forward Park transforms (or their inverses) into a single operation. The motivation is efficiency:  $\cos(\theta_e)$  and  $\sin(\theta_e)$  are computed exactly once and reused for both the Park rotation in the forward direction and the inverse Park rotation.

The forward composite takes  $(I_a, I_b, \theta_e)$  and produces  $(I_d, I_q)$  directly.

The inverse composite takes  $(V_d, V_q, \theta_e)$ , performs inverse Park to  $(V_\alpha, V_\beta)$ , then inverse Clarke to  $(V_a, V_b, V_c)$ , producing all three phase voltages in one call.

This is the form used by all inner-control-loop implementations in this project.

## Space Vector Modulation — $\alpha\beta$ Voltage Vector to PWM Duty Cycles

Space Vector Modulation (SVM) maps a two-component voltage reference vector in the  $\alpha\beta$  plane to three symmetrical PWM duty cycles. The implementation is *sector-free*: it reaches the same duty cycles as classical sector-and-dwell-time SVM by injecting a common-mode offset, which is algebraically equivalent and has no branch on sector index.

### Inverse Clarke:

The reference is first expanded to three phase voltages,  $v_A = V_\alpha$ ,  $v_B = -V_\alpha/2 + (\sqrt{3}/2) \cdot V_\beta$ ,  $v_C = -V_\alpha/2 - (\sqrt{3}/2) \cdot V_\beta$ .

### Common-mode (min-max) injection:

The offset  $v_{\text{Common}} = -(\max(v_A, v_B, v_C) + \min(v_A, v_B, v_C)) / 2$  is added to all three phases. Because it is identical on every phase it cancels in every line-to-line voltage, so it changes no motor current while centring the waveform. This is what recovers the linear range that classical SVM obtains from symmetric null-vector splitting: a factor  $2/\sqrt{3} \approx 1.155$  more than sinusoidal PWM reaches on the same bus. In the per-unit convention used here — inputs relative to  $V_{\text{dc}}/\sqrt{3}$  — that extended range *is* the unit circle, which is why the modulation limiter clamps on  $|V| \leq 1$  rather than on  $2/\sqrt{3}$ .

### Duty cycle output:

Each offset phase voltage is scaled by  $1/\sqrt{3}$ , biased by 0.5 to centre it in the PWM period, and clamped:  $d = \text{clamp}(v/\sqrt{3} + 0.5, 0, 1)$ . A reference of unit magnitude therefore spans exactly  $[0, 1]$ .

**Constraints:** - Input  $V_\alpha, V_\beta$  are per-unit relative to  $V_{\text{dc}}/\sqrt{3}$  (dimensionless, inside the unit circle for linear operation). - Output duty cycles are in the range  $[0.0, 1.0]$ ; values outside this range are clamped, which is how over-modulation is handled. - Being branch-free in the sector, the output is continuous across every sector boundary by construction. - SVM does not accept angles directly — it operates only on the  $(V_\alpha, V_\beta)$  pair.

## Fast Trigonometry — Lookup-Table Sine and Cosine

Direct hardware transcendental functions (`sin`, `cos`) introduce variable latency and are unsuitable for a 20 kHz ISR. A 512-entry pre-computed lookup table (LUT) covering one full period  $[0, 2\pi)$  provides both sine and cosine with bounded, cycle-predictable evaluation time.



Figure 24: Block diagram

**LUT characteristics:** - 512 entries, each a 32-bit floating-point value - Stored in ROM (read-only, `constexpr`) at a 16-byte memory alignment boundary - Total ROM footprint:  $512 \times 4 \text{ bytes} = 2048 \text{ bytes}$  (2 KB) - Covers exactly one full period

**Range handling:** the angle is *not* normalised to  $[0, 2\pi)$  before indexing. It is scaled by  $512/2\pi$ , truncated toward negative infinity, and the resulting index is masked with `size - 1`, so the wrap is a bitwise AND on the index rather than a modulo on the angle. This is why there is no `fmod` in the hot path. The consequence to know is the domain limit it leaves: the intermediate is an `int32_t`, so the result is meaningless for angles beyond roughly  $2.6 \times 10^9$  radians. Electrical angle reaches the transforms already wrapped, so this bound is not approached in the control loop.

**Cosine** reuses the sine table — there is no separate cosine table — but it is *not* an index offset. It adds  $\pi/2$  to the angle and performs the whole lookup again, so a cosine costs a full sine plus one addition, and evaluating both for the same angle does the range reduction twice.

**Interpolation:** Linear interpolation between adjacent LUT entries provides accuracy sufficient for FOC applications; the quantisation error is below the noise floor of 12-bit ADC current sensing.

## Interfaces

### Provided

| Interface               | Purpose                                                     | Contract                                                                                                                                |
|-------------------------|-------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------|
| Clarke —<br>Forward     | Converts (Ia, Ib, Ic) to (I $\alpha$ , I $\beta$ )          | All three phase currents are measured; no phase is reconstructed. The amplitude-invariant form remains valid for unbalanced phases.     |
| Clarke —<br>Inverse     | Converts (V $\alpha$ , V $\beta$ ) to (Va, Vb, Vc)          | Produces all three phase voltages. The sum of outputs is zero.                                                                          |
| Park —<br>Forward       | Converts (I $\alpha$ , I $\beta$ , $\theta_e$ ) to (Id, Iq) | Caller supplies pre-computed or LUT-evaluated $\cos(\theta_e)$ and $\sin(\theta_e)$ . Stateless.                                        |
| Park —<br>Inverse       | Converts (Vd, Vq, $\theta_e$ ) to (V $\alpha$ , V $\beta$ ) | Uses the same $\theta_e$ as the forward Park in the same control cycle.                                                                 |
| ClarkePark —<br>Forward | Converts (Ia, Ib, $\theta_e$ ) to (Id, Iq) in one call      | Computes cos/sin once; result is identical to chaining Clarke then Park separately.                                                     |
| ClarkePark —<br>Inverse | Converts (Vd, Vq, $\theta_e$ ) to (Va, Vb, Vc) in one call  | Chains inverse Park $\rightarrow$ inverse Clarke; computes cos/sin once. Output is three phase voltages, not (V $\alpha$ , V $\beta$ ). |

| Interface            | Purpose                                              | Contract                                                                                                                                                   |
|----------------------|------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------|
| SpaceVectorModulator | Generates $(V_\alpha, V_\beta)$ to three duty cycles | Inputs must be per-unit relative to $V_{dc}/\sqrt{3}$ . Outputs are always in $[0.0, 1.0]$ . Common-mode injection is internal; there is no sector branch. |
| FastTrigonometry     | Returns an approximation of $\sin(\theta)$           | $\theta$ is normalised to $[0, 2\pi)$ internally. ROM LUT; no floating-point transcendental at runtime.                                                    |
| FastTrigonometry     | Returns an approximation of $\cos(\theta)$           | Derived from the sine LUT via a quarter-period offset. Same ROM, no additional storage.                                                                    |

## Required

| Interface | Purpose                                                                        | Contract |
|-----------|--------------------------------------------------------------------------------|----------|
| None      | All transforms are pure mathematical algorithms with no external dependencies. | —        |

## Data Model

| Entity              | Field               | Type / Unit           | Range               | Notes                                                                 |
|---------------------|---------------------|-----------------------|---------------------|-----------------------------------------------------------------------|
| Three-phase current | Ia, Ib              | Ampere (float)        | $\pm$ rated current | Ic derived; not stored                                                |
| Stationary frame    | $I_\alpha, I_\beta$ | Ampere (float)        | $\pm$ rated current | Output of Clarke forward                                              |
| Synchronous frame   | Id, Iq              | Ampere (float)        | $\pm$ rated current | Output of Park forward                                                |
| Voltage demand      | Vd, Vq              | Dimensionless (float) | $[-1, +1]$          | Normalised to DC bus                                                  |
| Stationary voltage  | $V_\alpha, V_\beta$ | Dimensionless (float) | $[-1, +1]$          | Output of inverse Park / input to SVM                                 |
| Phase duty cycles   | Da, Db, Dc          | Dimensionless (float) | $[0.0, 1.0]$        | Clamped; 0 = always low, 1 = always high                              |
| Electrical angle    | $\theta_e$          | Radians (float)       | $[0, 2\pi)$         | $\theta_m \times \text{pole\_pairs}$ ; normalised before LUT indexing |
| LUT                 | Sine table          | float[512]            | $[-1.0, +1.0]$      | constexpr; 2 KB ROM; 16-byte aligned                                  |

## Block Diagram



Figure 25: Block diagram

## Constraints & Limitations

| Constraint                      | Value / Description                                                                                                         |
|---------------------------------|-----------------------------------------------------------------------------------------------------------------------------|
| Three-sensor current topology   | All three phase currents are measured directly. The amplitude-invariant Clarke form is valid regardless of phase balance.   |
| LUT ROM footprint               | 2 048 bytes (2 KB) of read-only memory.                                                                                     |
| LUT accuracy                    | Approximation error is bounded and below the noise floor of 12-bit ADC current sensing.                                     |
| SVM input normalisation         | $V\alpha$ , $V\beta$ must be dimensionless fractions in $[-1, +1]$ relative to the DC bus, not physical volt values.        |
| SVM output saturation           | Duty cycles are clamped to $[0.0, 1.0]$ . Over-modulation is silently saturated, not flagged.                               |
| Electrical angle responsibility | The caller is responsible for computing $\theta_e = \theta_m \times \text{pole\_pairs}$ before invoking any Park operation. |
| Stateless transforms            | No transform retains state between calls. Every call is independent.                                                        |
| Hot-path constraint             | All transforms must execute within the cycle budget of the 20 kHz FOC interrupt.                                            |

| Field     | Value              |
|-----------|--------------------|
| Title     | FOC Torque Control |
| Type      | design             |
| Status    | draft              |
| Version   | 0.1.0              |
| Component | foc-torque         |
| Date      | 2026-04-07         |

**IMPORTANT — Implementation-blind document:** This document describes *behavior, structure, and responsibilities* WITHOUT referencing code. **No code blocks using programming languages (C++, C, Python, CMake, shell, etc.) are allowed.** Use Mermaid diagrams to express behavior instead. Prose descriptions of algorithms are encouraged; source-level details are not.

**Diagrams:** All visuals must be either a Mermaid fenced code block (````mermaid`) or ASCII art inline in the document. External image references using Markdown image syntax are **not allowed**.

## Responsibilities

**Is responsible for:** - Executing the innermost FOC current control loop at 20 kHz in interrupt context - Converting three-phase stator currents to the rotor-synchronous dq frame via Clarke and Park transforms - Regulating the d-axis current ( $I_d$ ) to its setpoint using a dedicated PID controller - Regulating the q-axis current ( $I_q$ ) to its setpoint using a dedicated PID controller - Converting the resulting voltage demands back to three-phase PWM duty cycles via inverse Park and SVM - Accepting a combined ( $I_d$ ,  $I_q$ ) setpoint from an outer control loop or directly from an application - Providing Enable and Disable transitions that safely reset internal PID state

**Is NOT responsible for:** - Reading phase current values from the ADC hardware — currents are passed in as arguments - Committing duty cycles to the PWM hardware — duty cycles are returned to the Runner - Measuring or tracking rotor position — the mechanical angle is passed in as an argument - Outer-loop speed or position control - Flux weakening ( $I_d \neq 0$  operation is structurally possible but not yet supported)

---

## Component Details

### Innermost Control Loop — 20 kHz ISR Execution

The torque control loop is the innermost layer of the FOC cascade. It is invoked once per PWM switching period (default 20 kHz) by the hardware interrupt handler that drives the three-phase inverter. The entire loop — from phase current input to duty cycle output — must complete within the interrupt budget.

The eleven-step sequence executed each cycle is:

1. Receive phase currents ( $I_a$ ,  $I_b$ ,  $I_c$ ) as the **PhaseCurrents** argument.
2. Read the rotor mechanical angle  $\theta_m$  from the **Radians** argument (supplied by the Runner from the encoder).
3. Compute the electrical angle:  $\theta_e = \theta_m \times \text{pole\_pairs}$ .
4. Evaluate  $\cos(\theta_e)$  and  $\sin(\theta_e)$  via the fast trigonometric LUT — one evaluation, results reused.
5. Clarke forward:  $(I_a, I_b) \rightarrow (I_\alpha, I_\beta)$ ;  $I_c$  is derived from the balanced-phase constraint.
6. Park forward:  $(I_\alpha, I_\beta, \theta_e) \rightarrow (I_d, I_q)$ .
7. d-axis PID:  $\text{error} = I_{d\_setpoint} - I_d \rightarrow \text{output} = V_d$ .
8. q-axis PID:  $\text{error} = I_{q\_setpoint} - I_q \rightarrow \text{output} = V_q$ .
9. Inverse Park:  $(V_d, V_q, \theta_e) \rightarrow (V_\alpha, V_\beta)$ .
10. Space vector modulation:  $(V_\alpha, V_\beta) \rightarrow (\text{duty\_a}, \text{duty\_b}, \text{duty\_c})$ .
11. Return **PhasePwmDutyCycles** to the Runner for immediate PWM register update.

### d-axis and q-axis PID Controllers

Two independent incremental PID controllers regulate the direct (d) and quadrature (q) current components independently. Both PIDs share the same structural properties:

- **Output clamping:** symmetric to  $[-1, +1]$ , representing a normalised fraction of the DC bus voltage.

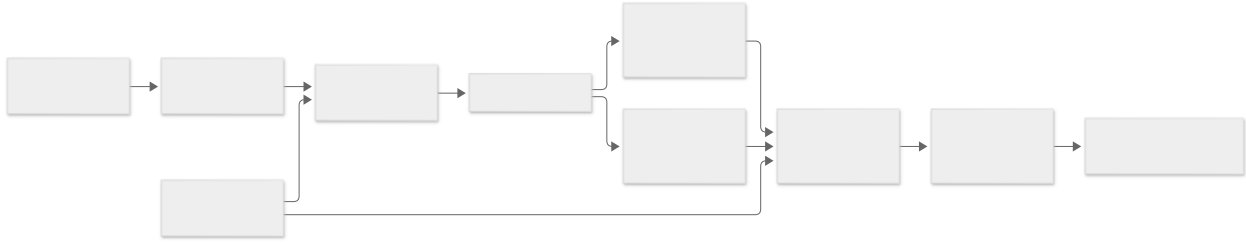


Figure 26: Block diagram

- **Anti-windup:** the integrator is clamped whenever the output reaches the saturation boundary, preventing integrator wind-up during transients.
- **Gain derivation:** the current regulator derives its own gains from the motor model (phase resistance, inductance) and the requested closed-loop bandwidth, normalised by the DC bus voltage. Tuning is therefore expressed as a single bandwidth in radians per second rather than raw controller gains.
- **Incrementalism:** the PID uses the incremental (velocity) form, accumulating output from step to step, which naturally limits abrupt changes at Enable transitions.

The d-axis setpoint is normally 0 A for a surface PMSM ( $I_d = 0$  gives maximum torque per ampere). Setting  $I_d$ \_setpoint to a non-zero negative value for flux weakening is structurally possible but the operational consequences are outside the scope of this document.

## Enable and Disable

**Enable:** arms both the d-axis and the q-axis PID controllers and resets their internal integrator state to zero. The most recently stored setpoints ( $I_d$ \_setpoint,  $I_q$ \_setpoint) are re-applied immediately — there is no rollback to zero. This allows seamless re-engagement after a brief disable event without a step change in the setpoint.

**Disable:** disarms both PIDs. While disarmed, the `Calculate()` method returns zero duty cycles regardless of the current error. PID state is frozen and will be fully reset on the next Enable call.

The two states are the only run-time states of this component:

## Current Envelope

The speed and position cascades bound the current reference their outer loop produces. Torque mode has no outer loop — the reference arrives from the operator, over CAN or the CLI — so the cascade applies the bound itself, at `SetPoint`, before the reference reaches the current loop.

The bound is on the **magnitude of the current vector**, not on either axis separately: the plant constraint is  $\sqrt{I_d^2 + I_q^2} \leq I_{max}$ , and a per-axis clamp would admit  $\sqrt{2}$  times the limit on the diagonal. A setpoint outside the envelope is scaled onto it, preserving the commanded ratio between  $I_d$  and  $I_q$  rather than clipping one axis and rotating the vector.

The envelope comes from the inverter's `MaxCurrentSupported()`, the same source the speed and position cascades take theirs from, so the three modes cannot disagree about the hardware limit.

The ( $I_d$ ,  $I_q$ ) pair is written outside the interrupt and read by it. It is handed over through a lock-free triple buffer — the same mechanism that carries estimator windows the other way — so

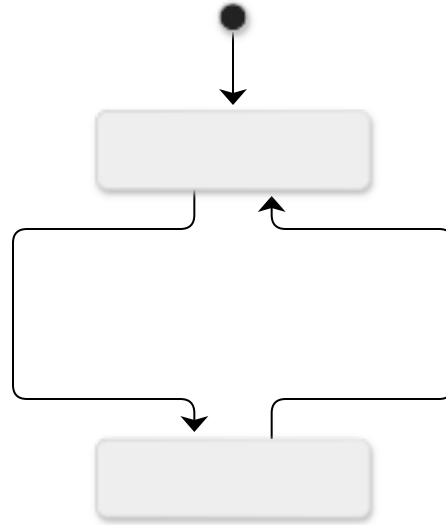


Figure 27: State diagram

the interrupt always reads a pair published together, never a new Id with the previous Iq.

## Pole-Pair Configuration

The pole-pair count is an integer property that translates the mechanical rotor angle supplied by the encoder into the electrical angle required by the Park transform. It must be configured before the first `Calculate()` call. Changing it while enabled produces undefined control behaviour and must be avoided.

## Interfaces

### Provided

| Interface       | Purpose                                                                                                         | Contract                                                                                                                                           |
|-----------------|-----------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------|
| Configure       | Supplies the motor model: resistance, inductance, flux linkage, bus voltage, sampling frequency and pole pairs. | Must be called before the first <code>Calculate()</code> . Must not be changed while Enabled.                                                      |
| Enable          | Arms both PID controllers and resets their integrator state.                                                    | Safe to call repeatedly. PIDs start from a clean state each time. Last setpoints are preserved.                                                    |
| Disable         | Disarms both PID controllers and forces zero duty cycle output.                                                 | Safe to call from any context. <code>Calculate()</code> returns zero while disabled.                                                               |
| SetCurrentLimit | Provides the current loop closed-loop bandwidth.                                                                | Gains are derived from the motor model and normalised internally. Takes effect on the next <code>Calculate()</code> call.                          |
| SetPoint        | Sets the (Id, Iq) current setpoint in Ampere.                                                                   | Scaled onto the current envelope on the way in. New setpoint is used on the next <code>Calculate()</code> invocation. Can be called while Enabled. |

| Interface Purpose                                                                 | Contract                                                                                       |
|-----------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------|
| <b>Calculate</b> Executes the full 11-step FOC torque loop for one control cycle. | Must be called at 20 kHz from ISR context. Returns <b>PhasePwmDutyCycles</b> . Must not block. |

## Required

| Interface Purpose                                                                                                                      | Contract |
|----------------------------------------------------------------------------------------------------------------------------------------|----------|
| None    The inverter and encoder are managed externally by the Runner; they are not accessed directly by the torque control component. | —        |

## Data Model

| Entity            | Field        | Type / Unit           | Range               | Notes                                                                 |
|-------------------|--------------|-----------------------|---------------------|-----------------------------------------------------------------------|
| Phase currents    | Ia, Ib, Ic   | Ampere (float)        | $\pm$ motor rated A | Ic derived; passed in from ADC callback                               |
| Synchronous frame | Id, Iq       | Ampere (float)        | $\pm$ motor rated A | Regulated quantities                                                  |
| Current setpoint  | Id_sp, Iq_sp | Ampere (float)        | $\pm$ motor rated A | Id_sp = 0 for SPMSM MTPA; Iq_sp from application                      |
| Voltage demands   | Vd, Vq       | Dimensionless (float) | $[-1, +1]$          | PID output; normalised to DC bus                                      |
| Duty cycles       | Da, Db, Dc   | Dimensionless (float) | $[0.0, 1.0]$        | Returned to Runner for PWM register write                             |
| Mechanical angle  | $\theta_m$   | Radians (float)       | $[0, 2\pi)$         | Supplied by Runner; pole-pairs multiply to $\theta_e$                 |
| Electrical angle  | $\theta_e$   | Radians (float)       | $[0, 2\pi)$         | $\theta_m \times \text{pole\_pairs}$ ; used for Park and inverse Park |
| Pole pairs        | P            | Integer (unsigned)    | $\geq 1$            | Motor property; configured once at startup                            |

## Constraints & Limitations

| Constraint        | Value / Description                                                                                                      |
|-------------------|--------------------------------------------------------------------------------------------------------------------------|
| Control loop rate | 20 kHz — must be called once per PWM switching period from the FOC interrupt.                                            |
| Cycle budget      | $\leq 4500$ cycles (75% of the 6000-cycle control period at 120 MHz / 20 kHz) for the full <b>Calculate()</b> execution. |

| Constraint                      | Value / Description                                                                                                              |
|---------------------------------|----------------------------------------------------------------------------------------------------------------------------------|
| No virtual dispatch in hot path | <code>Calculate()</code> must not incur virtual function call overhead.                                                          |
| Output duty cycle format        | Normalised floating-point [0.0, 1.0]; conversion to PWM timer counts is the Runner's responsibility.                             |
| Electrical angle wrapping       | Handled by the LUT normalisation within <code>FastTrigonometry</code> — no explicit modulo operation required in loop.           |
| Gain derivation dependency      | Gain derivation requires the motor model — including the DC bus voltage — to be supplied through <code>Configure()</code> first. |
| No flux weakening               | <code>Id_setpoint</code> $\neq 0$ is structurally accepted but operational flux-weakening strategy is not yet defined.           |
| Current envelope                | Vector magnitude bounded to the inverter's <code>MaxCurrentSupported()</code> at <code>SetPoint</code> ; it is not a regulator.  |
| PID state at Enable             | Integrators are always zeroed on Enable, regardless of previous state.                                                           |

## Controller Alternatives

The dual-PID current controller described above is one of several selectable strategies for the current loop. Alternative controllers — Decoupled PID, Deadbeat, and Sliding-mode — are described in `documentation/theory/advanced-controllers.md` (`current-loop-decoupled-pid.md`, `current-loop-deadbeat.md`, `current-loop-sliding-mode.md`). Plant model foundations are in `documentation/theory/foc-plant-models.md`.

The mechanism by which a controller strategy is selected at runtime, stored without heap allocation, dispatched in the ISR via type-aware visit, and persisted to NVM is described in `documentation/design/controller-selection.md`.

| Field     | Value             |
|-----------|-------------------|
| Title     | FOC Speed Control |
| Type      | design            |
| Status    | draft             |
| Version   | 0.1.0             |
| Component | foc-speed         |
| Date      | 2026-04-07        |

**IMPORTANT — Implementation-blind document:** This document describes *behavior, structure, and responsibilities* WITHOUT referencing code. **No code blocks using programming languages (C++, C, Python, CMake, shell, etc.) are allowed.** Use Mermaid diagrams to express behavior instead. Prose descriptions of algorithms are encouraged; source-level details are not.

**Diagrams:** All visuals must be either a Mermaid fenced code block (````mermaid`) or ASCII art inline in the document. External image references using Markdown image syntax are **not allowed**.

## Responsibilities

**Is responsible for:** - Executing the two-level FOC cascade: an outer speed PID loop (1 kHz by default) and an inner current control loop (20 kHz) - Estimating rotor angular velocity from successive mechanical angle samples using finite differences with wrap-around compensation - Regulating rotor speed to the commanded setpoint in radians per second - Producing the  $I_q$  (quadrature current) setpoint that drives the inner torque loop, bounded by a configurable current limit - Registering an outer-loop callback with the `LowPriorityInterrupt` interface so that the outer loop fires at the correct prescale ratio - Providing `OuterLoopFrequency()` so that the application can schedule the `LowPriorityInterrupt` at the correct rate - Arming and resetting all three PID controllers (speed, d-axis, q-axis) on Enable / Disable

**Is NOT responsible for:** - Reading phase currents or encoder position directly from hardware — these are supplied by the Runner - Writing duty cycles to PWM hardware — duty cycles are returned to the Runner - Position control or any loop beyond the speed layer - Flux weakening ( $I_d$  is always commanded to 0 A)

---

## Component Details

### Two-Loop Cascade Architecture

Speed control builds on the torque control loop by adding a supervisory outer loop. The two loops operate at different rates and in different interrupt contexts:

| Loop       | Rate   | Context                  | Input                            | Output                  |
|------------|--------|--------------------------|----------------------------------|-------------------------|
| Inner loop | 20 kHz | High-priority FOC<br>ISR | Phase currents, $\theta_m$       | PWM duty cycles         |
| Outer loop | 1 kHz  | Low-priority interrupt   | $\theta_m$ from last inner cycle | $I_q$ setpoint (Ampere) |

The inner loop is identical to the torque control loop described in the FOC Torque Control design document. The outer loop adds the speed PID and the speed estimation step.

### Outer Loop — Speed Estimation and Speed PID

The outer loop is triggered by the `LowPriorityInterrupt` at a prescale ratio determined by the ratio of the inner-loop frequency to the desired outer-loop frequency (e.g., 20 kHz / 1 kHz = 20 inner cycles per outer cycle).

#### Speed estimation by finite difference:

The angular velocity is estimated from the difference between the current mechanical angle sample and the previous angle sample, divided by the outer-loop period  $\Delta t$ :

$$\omega = \Delta\theta / \Delta t$$

Because the encoder angle wraps around at  $2\pi$ , the raw angle difference may be discontinuous at the wrap boundary. Wrap-around compensation applies a correction: if the computed  $\Delta\theta$  is greater than  $+\pi$ , it is shifted by  $-2\pi$ ; if it is less than  $-\pi$ , it is shifted by  $+2\pi$ . This produces a signed

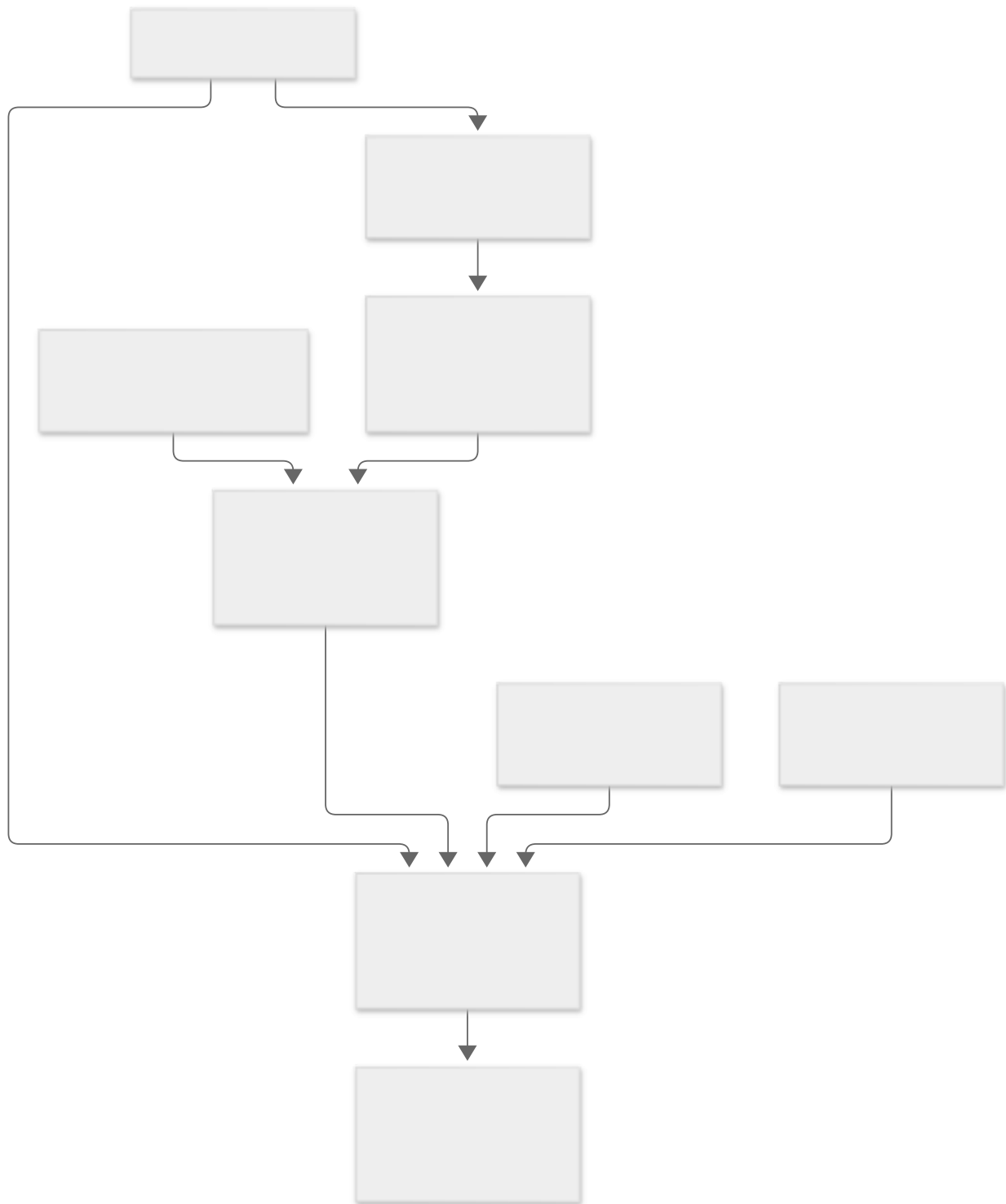


Figure 28: Block diagram

angular velocity in the range  $(-\pi/\Delta t, +\pi/\Delta t)$  radians per second for any physical speed below half the Nyquist rate of the sampling interval.

### Speed PID:

The speed PID receives the speed error (setpoint minus estimate) and produces the  $I_q$  setpoint. Its output is clamped to  $\pm \text{maxCurrent}$  (in Ampere), which directly limits the peak torque the motor can apply while tracking the speed command. This clamp is the primary over-current protection for the speed-control mode.

The d-axis setpoint is 0 A, except while an online electrical estimator is attached: the cascade then adds a  $\pm 2.5\%$  of  $\text{maxCurrent}$ , 10 Hz square wave so the winding resistance stays observable (see `service-electrical-ident.md`, `Excitation`).

### LowPriorityInterrupt — Outer Loop Scheduling

The `LowPriorityInterrupt` is an abstract scheduling interface. The speed control component registers its outer-loop handler with this interface at construction time. The inner-loop ISR counts cycles and triggers the `LowPriorityInterrupt` at the appropriate prescale ratio, causing the outer-loop handler to execute without RTOS involvement.

The registered handler captures the cascade instance, so its lifetime is bound to that instance. The cascade calls `Unregister()` from its destructor; after `Unregister()` returns, an already queued trigger must be discarded rather than dispatched, so a mode switch that destroys the cascade cannot leave a handler pointing at freed storage.

`OuterLoopFrequency()` returns the configured outer-loop rate so the application layer can assert or configure the LPI trigger frequency correctly.

### Enable and Disable

**Enable:** resets and arms all three PID controllers (speed, d-axis, q-axis). The last speed setpoint is preserved so the motor resumes commanding toward the previous target.

**Disable:** disarms all three PIDs. `Calculate()` returns zero duty cycles. The outer loop continues to receive its LPI callbacks but produces no  $I_q$  output while disarmed.

---

## Interfaces

### Provided

| Interface | Purpose                                                                                | Contract                                                                                      |
|-----------|----------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------|
| Configure | Supplies the motor model, including the pole-pair count used for the electrical angle. | Must be called before the first <code>Calculate()</code> . Must not be changed while Enabled. |
| Enable    | Arms all three PIDs and resets their integrator state.                                 | Safe to call repeatedly. Speed setpoint is preserved.                                         |

| Interface                    | Purpose                                                                   | Contract                                                                                                                                                                                                                                                  |
|------------------------------|---------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Disable                      | Disarms all PIDs and forces zero duty cycle output.                       | Safe to call from any context.                                                                                                                                                                                                                            |
| SetCurrentSettings           | Sets the current loop closed-loop bandwidth.                              | Gains are derived from the motor model and normalised internally. Takes effect on the next <b>Calculate()</b> .                                                                                                                                           |
| SetSpeedSettings             | Sets the speed loop closed-loop bandwidth.                                | Gains are derived from the mechanical model supplied through <b>ConfigureMechanics()</b> . Output is clamped to $\pm \text{maxCurrent}$ .                                                                                                                 |
| SetPoint                     | Sets the speed setpoint in radians per second.                            | Written atomically; used on the next outer-loop cycle.                                                                                                                                                                                                    |
| Calculate                    | Executes the inner 20 kHz FOC torque loop for one cycle.                  | Called from the FOC ISR; returns <b>PhasePwmDutyCycles</b> . Must not block.                                                                                                                                                                              |
| OuterLoopFrequency           | Returns the configured outer-loop frequency in Hz.                        | Pure query; no side effects. Can be called before Enable.                                                                                                                                                                                                 |
| SetOnlineMechanicalEstimator | Attaches the mechanical parameter estimator (optional).                   | If attached, the estimator is updated automatically on each outer-loop cycle. No separate estimator Enable/Disable interface is defined here; detach or omit the estimator to stop updates.                                                               |
| SetOnlineElectricalEstimator | Attaches the electrical parameter estimator (optional).                   | If attached, the estimator is updated each outer-loop cycle from window averages the inner loop accumulates, and a d-axis square wave is added to the current reference so the resistance is observable. Assumes non-salient motor ( $L_d \approx L_q$ ). |
| ObserveMotor                 | Reports the window speed, window-mean q-axis current and speed reference. | Written by the control interrupts, read from the event loop; feeds telemetry (REQ-INT-014) and the encoder plausibility monitor (REQ-SM-028).                                                                                                             |

## Required

| Interface            | Purpose                                                                        | Contract                                                                                                                                                             |
|----------------------|--------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| LowPriorityInterrupt | Provides the callback registration point for the outer-loop speed PID handler. | The inner-loop ISR triggers the LPI at the correct prescale ratio. The outer-loop handler must execute within its own interrupt context, not the 20 kHz ISR context. |

## Data Model

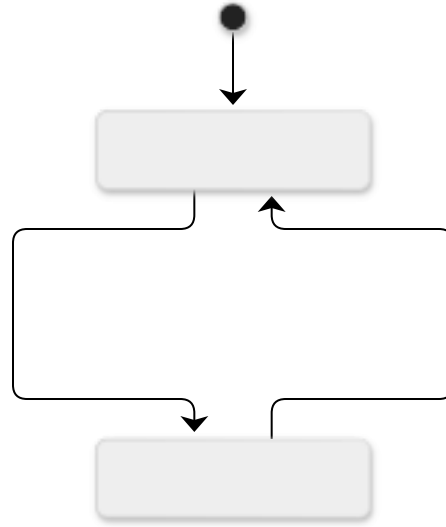


Figure 29: State diagram

| Entity            | Field              | Type / Unit              | Range                                              | Notes                                                    |
|-------------------|--------------------|--------------------------|----------------------------------------------------|----------------------------------------------------------|
| Speed setpoint    | $\omega_{sp}$      | RadiansPerSecond (float) | $\pm$ mechanical max                               | Written by application; applied on next outer-loop tick  |
| Estimated speed   | $\omega$           | RadiansPerSecond (float) | computed from $\Delta\theta/\Delta t$              | Computed each outer-loop cycle                           |
| Speed PID output  | $Iq_{sp}$          | Ampere (float)           | $\pm$ maxCurrent                                   | Written to inner loop; clamped by speed PID output limit |
| d-axis setpoint   | $Id_{sp}$          | Ampere (float)           | 0 A; $\pm 2.5\%$ $I_{max}$ at 10 Hz with estimator | SPMSM maximum torque per ampere                          |
| Previous angle    | $\theta_{m\_prev}$ | Radians (float)          | $[0, 2\pi)$                                        | Saved each outer cycle for finite-difference estimator   |
| Outer loop period | $\Delta t$         | Seconds (float)          | $1 / \text{outer\_frequency}$                      | Constant after construction                              |
| Pole pairs        | P                  | Integer (unsigned)       | $\geq 1$                                           | Motor property                                           |
| Max current       | maxCurrent         | Ampere (float)           | $> 0$                                              | Upper bound on $Iq$ setpoint from speed PID              |

## Sequence Diagrams

### Single Outer-Loop Cycle (1 kHz LPI Event)

#### Online Estimator Update Within Outer-Loop Cycle

When online estimators are attached and enabled, each 1 kHz outer-loop cycle also feeds observations to them after the speed PID step:

**Vd\_reconstructed** is the d-axis voltage back-computed from the last normalized Vd output of the inner loop and the DC bus voltage.

---

| Constraint         | Value / Description                                                                                                                 |
|--------------------|-------------------------------------------------------------------------------------------------------------------------------------|
| LPI scheduling     | The outer loop does not use an RTOS. The inner loop ISR triggers the LPI — no timer or task involved.                               |
| Cycle budget       | Inner loop <code>Calculate()</code> must complete in $\leq 4500$ cycles (75% of the 6000-cycle control period at 120 MHz / 20 kHz). |
| Setpoint atomicity | The Iq setpoint written by the outer loop must be read atomically by the inner loop on 32-bit ARM targets.                          |

---



---

## Controller Alternatives

The speed PID described above is one of several selectable strategies for the outer speed loop. Alternative controllers — LQI, ADRC, and Two-DOF — are described in `documentation/theory/advanced-controllers.md` (`speed-loop-lqi.md`, `speed-loop-adrc.md`, `speed-loop-two-dof.md`). Plant model foundations are in `documentation/theory/foc-plant-models.md`.

The runtime selection mechanism is described in `documentation/design/controller-selection.md`.

| Field     | Value                |
|-----------|----------------------|
| Title     | FOC Position Control |
| Type      | design               |
| Status    | draft                |
| Version   | 0.1.0                |
| Component | foc-position         |
| Date      | 2026-04-07           |

---

**IMPORTANT — Implementation-blind document:** This document describes *behavior, structure, and responsibilities* WITHOUT referencing code. **No code blocks using programming languages (C++, C, Python, CMake, shell, etc.) are allowed.** Use Mermaid diagrams to express behavior instead. Prose descriptions of algorithms are encouraged; source-level details are not.

**Diagrams:** All visuals must be either a Mermaid fenced code block (````mermaid`) or ASCII art inline in the document. External image references using Markdown image syntax are **not allowed**.

---

## Responsibilities

**Is responsible for:**

- Executing the FOC cascade: an outermost position loop and a middle speed loop (both at 1 kHz within the same LPI callback), and the innermost current control loop (20 kHz)
- Regulating rotor mechanical position to a commanded setpoint in radians
- Dispatching to the selected position algorithm and routing its output to either the speed loop or, for the state feedback laws, straight to the current loop
- Estimating rotor angular velocity from consecutive mechanical

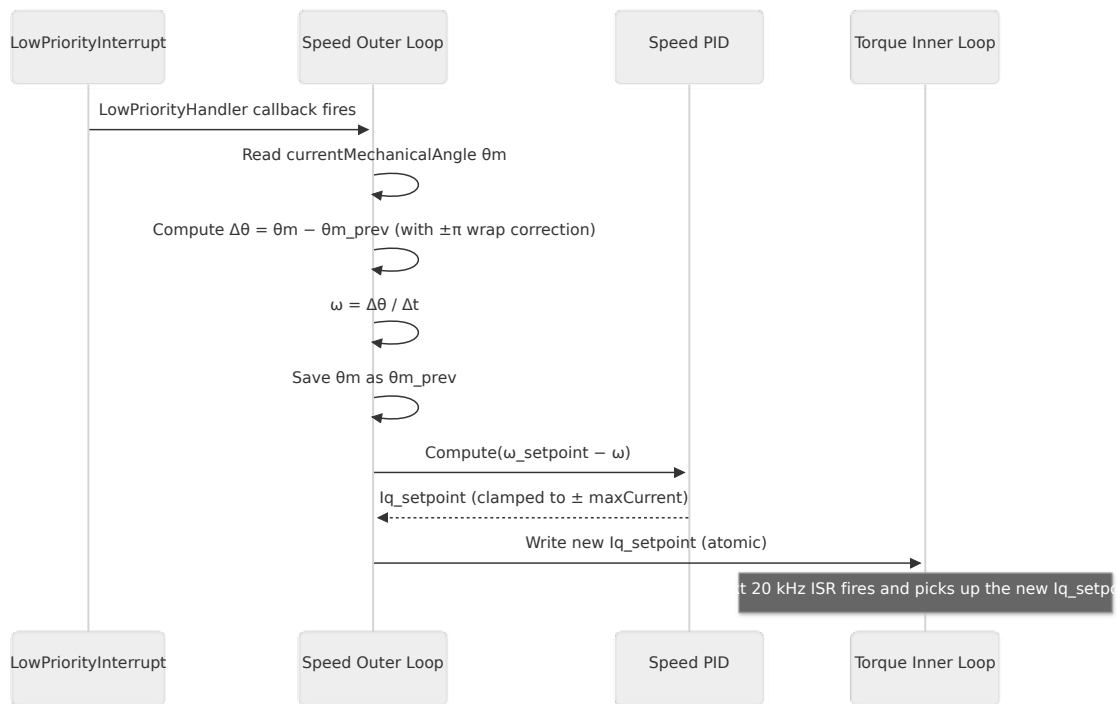


Figure 30: Sequence diagram

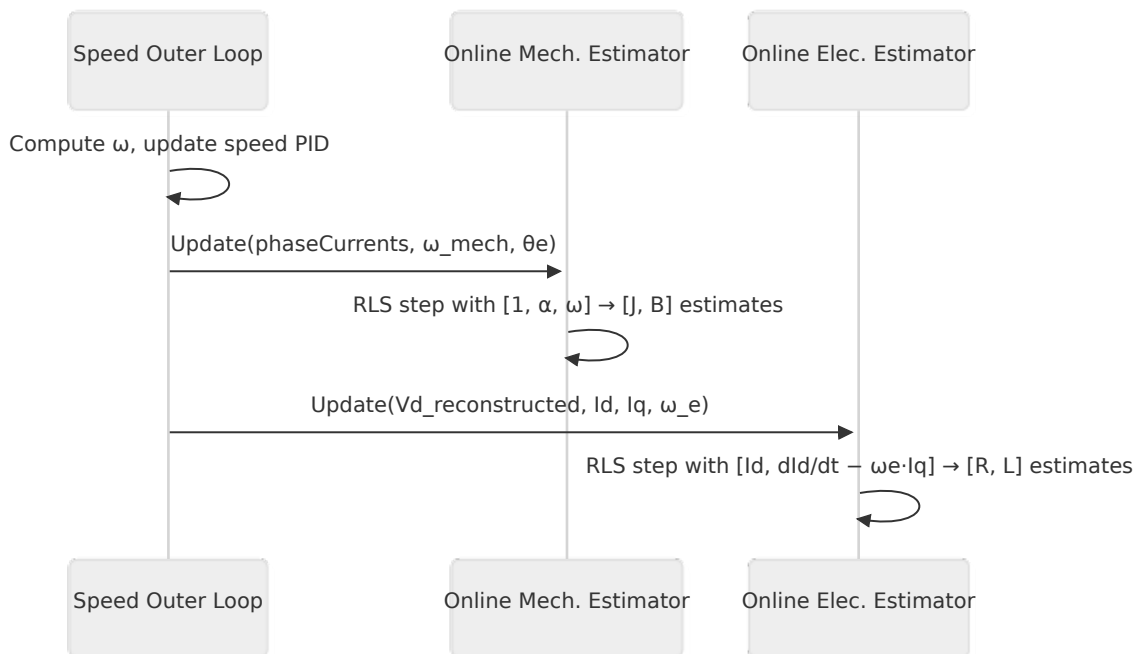


Figure 31: Sequence diagram

angle samples (same wrap-around compensation as speed control) for the middle loop - Resetting the position, speed and current controller state on Enable / Disable - Accepting a position setpoint in mechanical radians and propagating all derived setpoints through the cascade

**Is NOT responsible for:** - Reading phase currents or encoder position directly from hardware — these are supplied by the Runner - Writing duty cycles to the PWM hardware — duty cycles are returned to the Runner - Multi-turn position tracking — the setpoint is a single-turn mechanical angle, and the loop always takes the shorter way round to it - Flux weakening (Id is always commanded to 0 A)

## Component Details

### Three-Loop Cascade Architecture

Position control extends the speed control design by adding a third outermost loop. All three loops share the same `LowPriorityInterrupt` trigger; the outer two loops execute sequentially within a single LPI callback, while the innermost loop executes in the high-priority 20 kHz ISR.

| Loop        | Rate   | Context                | Input                              | Output                 |
|-------------|--------|------------------------|------------------------------------|------------------------|
| Inner loop  | 20 kHz | High-priority FOC ISR  | Phase currents, $\theta_m$         | PWM duty cycles        |
| Middle loop | 1 kHz  | Low-priority interrupt | $\theta_m$ from last inner cycle   | Iq setpoint (Ampere)   |
| Outer loop  | 1 kHz  | Low-priority interrupt | $\theta_m$ (position), position SP | Speed setpoint (rad/s) |

The position loop fires **before** the speed loop within the same LPI callback. This ensures the speed setpoint written by the position PID is immediately consumed by the speed PID in the same interrupt invocation, keeping both derived setpoints in step.

### Outer Loop — Selectable Position Law

The position error is always the wrapped difference between the setpoint and the current mechanical angle  $\theta_m$ , folded into  $[-\pi, \pi]$  so a move across the encoder seam takes the shorter way round rather than most of a turn backwards.

`PositionControllerSelector` holds every position algorithm in a `std::variant` and dispatches through `std::visit`, exactly as the current and speed loops do. Each algorithm declares the kind of output it produces:

- **Speed reference** (PID, Cascade P, Two-DOF): the value is written as the speed setpoint and the speed loop runs immediately afterwards in the same callback, as it always has.
- **Current reference** (LQR, LQI): the speed loop is skipped entirely and the value becomes the q-axis current setpoint directly. These laws carry  $\omega$  in their own state vector and already regulate it, so running them on top of a speed loop would stack two regulators on the same state.

Figure 32: Block diagram

The speed-reference laws saturate at one loop bandwidth of speed per  $\pi$  radians of error. The current-reference laws clamp to  $\pm \text{maxCurrent}$ , and LQI stops accumulating its integral state whenever that clamp is active.

See `documentation/design/controller-selection.md` for the algorithm set, the numerical conditioning the state feedback designs require, and the rules governing when a selection or a retune is refused.

## Middle Loop — Speed Estimation and Speed PID

The speed estimation and speed PID in the middle loop are functionally identical to those described in the FOC Speed Control design document. The speed PID output is clamped to  $\pm \text{maxCurrent}$  (Ampere) and written as the  $I_q$  setpoint for the inner current loop.

When a speed-reference position law is active, the speed setpoint consumed by the speed PID is the output of the position law from the same LPI invocation, not a value set by the application. When a current-reference law is active the speed loop does not run at all.

## Inner Loop — Torque and Current Control

The inner loop is functionally identical to the FOC Torque Control design document: Clarke, Park, d-axis PID, q-axis PID, inverse Park, and SVM. It runs independently at 20 kHz and reads its  $I_q$  setpoint from the value most recently written by the middle loop.

The d-axis setpoint is 0 A, except while an online electrical estimator is attached: the cascade then adds a  $\pm 2.5\%$  of  $\text{maxCurrent}$ , 10 Hz square wave so the winding resistance stays observable (see `service-electrical-ident.md`, Excitation).

## Enable and Disable

**Enable:** resets and arms all four PID controllers in cascade order (position, speed, d-axis, q-axis). The last position setpoint is preserved.

**Disable:** disarms all four PIDs. `Calculate()` returns zero duty cycles until re-enabled.

## LPI Callback Execution Order Within One 1 kHz Tick

Within a single LPI callback, the position and speed loops execute in strict order:

1. **Position law fires first:** reads  $\theta_m$ , computes the wrapped position error, produces either a speed setpoint or an  $I_q$  setpoint.
2. **Speed PID fires second, if the position law produced a speed setpoint:** reads the freshly updated setpoint, computes speed error using  $\Delta\theta/\Delta t$ , produces and clamps the  $I_q$  setpoint. A current-reference law writes the  $I_q$  setpoint itself and this step is skipped.
3. Callback returns. On the next 20 kHz ISR cycle, the inner loop picks up the updated  $I_q$  setpoint.

This ordering guarantees that the  $I_q$  setpoint used by the inner loop is always the one derived in the most recent outer-loop tick.

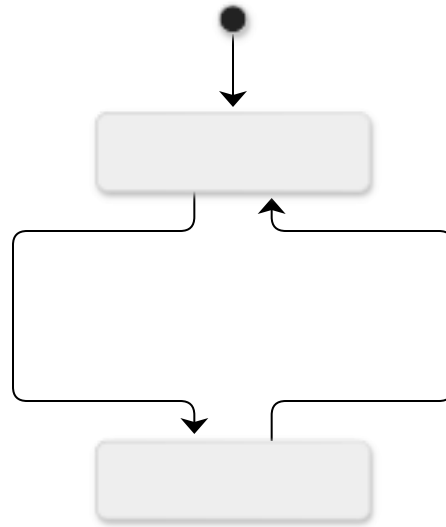


Figure 33: State diagram

## Interfaces

### Provided

| Interface               | Purpose                                                  | Contract                                                                                                                                                                                                    |
|-------------------------|----------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Configure               | Supplies the motor model, including the pole-pair count. | Must be called before the first <b>Calculate()</b> . Must not be changed while Enabled.                                                                                                                     |
| Enable                  | Resets the position law and arms the loops beneath it.   | Safe to call repeatedly. Position setpoint is preserved.                                                                                                                                                    |
| Disable                 | Disarms all PIDs and forces zero duty cycle output.      | Safe to call from any context.                                                                                                                                                                              |
| SetCurrentSettings      | Settings the current loop closed-loop bandwidth.         | Gains are derived from the motor model and normalised internally.                                                                                                                                           |
| SetSpeedSettings        | Settings the speed loop closed-loop bandwidth.           | Speed loop output clamped to $\pm \text{maxCurrent}$ .                                                                                                                                                      |
| SetPositionSettings     | Settings position loop bandwidth and cost weights.       | Returns <b>SelectResult</b> . Refused with <b>busy</b> while enabled; refused with <b>invalidParameters</b> if the active law cannot be redesigned for the new tunings, leaving the last accepted set live. |
| SelectPositionAlgorithm | Selects the position algorithm.                          | Refused with <b>busy</b> while enabled, and with <b>invalidParameters</b> when the design does not converge; the previously active algorithm stays live.                                                    |

| Interface | Purpose                                                  | Contract                                                                     |
|-----------|----------------------------------------------------------|------------------------------------------------------------------------------|
| SetPoint  | Sets the target position in mechanical radians.          | Written atomically; used on the next outer-loop cycle.                       |
| Calculate | Executes the inner 20 kHz FOC torque loop for one cycle. | Called from the FOC ISR; returns <b>PhasePwmDutyCycles</b> . Must not block. |

## Required

| Interface          | Purpose                                                                                 | Contract                                                                                                                            |
|--------------------|-----------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------|
| LowPriorityProvide | Provides the callback registration point for the combined position + speed outer loops. | The inner-loop ISR triggers the LPI at the 1 kHz prescale ratio. Both outer-loop stages execute sequentially in one LPI invocation. |

## Data Model

| Entity              | Field            | Type / Unit              | Range                                              | Notes                                                                                |
|---------------------|------------------|--------------------------|----------------------------------------------------|--------------------------------------------------------------------------------------|
| Position setpoint   | $\theta\_sp$     | Radians (float)          | Application-defined                                | Not wrapped — application must supply single-turn-compatible target                  |
| Actual position     | $\theta_m$       | Radians (float)          | $[0, 2\pi)$                                        | Mechanical angle from encoder                                                        |
| Position PID output | $\omega\_sp$     | RadiansPerSecond (float) | $[-bw \cdot \pi, +bw \cdot \pi]$<br>rad/s          | Clamped speed setpoint; saturation = one speed-loop bandwidth per $\pi$ rad of error |
| Estimated speed     | $\omega$         | RadiansPerSecond (float) | Computed from $\Delta\theta/\Delta t$              | Finite difference with $\pm\pi$ wrap correction                                      |
| Speed PID output    | $Iq\_sp$         | Ampere (float)           | $\pm \text{maxCurrent}$                            | Written to inner loop                                                                |
| d-axis setpoint     | $Id\_sp$         | Ampere (float)           | 0 A; $\pm 2.5\%$ $I_{max}$ at 10 Hz with estimator | SPMSM maximum torque per ampere                                                      |
| Previous angle      | $\theta_m\_prev$ | Radians (float)          | $[0, 2\pi)$                                        | Saved each outer cycle for speed estimator                                           |
| Outer loop period   | $\Delta t$       | Seconds (float)          | 1 / $outer\_frequency$                             | Constant after construction                                                          |
| Pole pairs          | P                | Integer (unsigned)       | $\geq 1$                                           | Motor property                                                                       |

| Entity      | Field           | Type / Unit              | Range                             | Notes                                                         |
|-------------|-----------------|--------------------------|-----------------------------------|---------------------------------------------------------------|
| Max current | maxCurrent      | Ampere (float)           | $> 0$                             | Upper bound on Iq setpoint from speed PID                     |
| Speed clamp | $\pm$ bandwidth | RadiansPerSecond (float) | derived from speed-loop bandwidth | Position PID output saturation; determines approach speed cap |

## Sequence Diagrams

### Single 1 kHz LPI Tick — Position and Speed Outer Loops

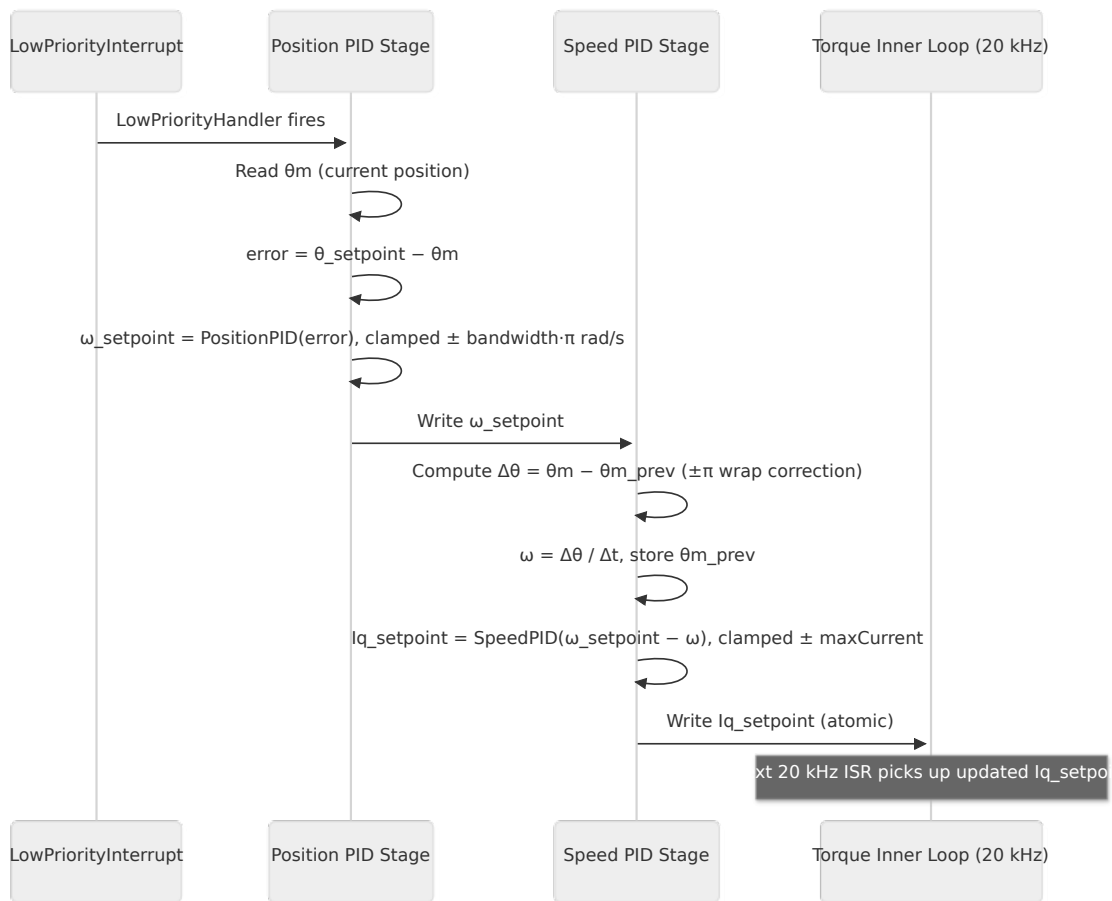


Figure 34: Sequence diagram



Figure 35: Block diagram

## Block Diagram

## Constraints & Limitations

| Constraint                      | Value / Description                                                                                                                 |
|---------------------------------|-------------------------------------------------------------------------------------------------------------------------------------|
| Inner loop rate                 | 20 kHz — called from the FOC ISR once per PWM period.                                                                               |
| Outer loop rate                 | 1 kHz (same for both position and speed stages). Must be an integer divisor of 20 kHz.                                              |
| Speed-reference clamp           | One loop bandwidth of speed per $\pi$ radians of error, which bounds the approach speed at any position error.                      |
| No multi-turn tracking          | Position setpoint is in single-turn radians. Multi-turn logic must be handled externally.                                           |
| Position wrap-around correction | The position error is folded into $[-\pi, \pi]$ , so a setpoint across the encoder seam moves the short way round.                  |
| LPI callback order              | The position law must always execute before the speed PID within the same LPI callback. Reversing the order yields stale setpoints. |
| No flux weakening               | $\text{Id} = 0$ is invariant. High-speed flux-weakening operation is out of scope.                                                  |
| Cycle budget (inner loop)       | <code>Calculate()</code> must complete in $\leq 4500$ cycles (75% of the 6000-cycle control period at 120 MHz / 20 kHz).            |
| Controller state at Enable      | Every integrator and observer is zeroed on Enable; the position setpoint is preserved.                                              |
| Setpoint atomicity              | Speed and Iq setpoints written by outer loops must be read atomically by downstream loops on 32-bit ARM.                            |

| Field     | Value                        |
|-----------|------------------------------|
| Title     | Runtime Controller Selection |
| Type      | design                       |
| Status    | accepted                     |
| Version   | 0.3.0                        |
| Component | controller-selection         |
| Date      | 2026-08-17                   |

**IMPORTANT — Implementation-blind document:** This document describes *behavior, structure, and responsibilities* WITHOUT referencing code. **No code blocks using programming languages (C++, C, Python, CMake, shell, etc.) are**

**allowed.** Use Mermaid diagrams to express behavior instead. Prose descriptions of algorithms are encouraged; source-level details are not.

**Diagrams:** All visuals must be either a Mermaid fenced code block (````mermaid`) or ASCII art inline in the document. External image references using Markdown image syntax are **not allowed**.

---

## Responsibilities

**Is responsible for:**

- Maintaining a collection of concrete controller instances for each of the three FOC loops (current, speed, position), all stored in fixed-size, statically-allocated storage without heap use
- Activating one controller per loop at runtime by constructing it in place from the available set
- Routing each loop's per-sample computation to the active controller with zero virtual-dispatch overhead through type-aware dispatch (variant visit)
- Enforcing that algorithm selection is only permitted while the active mode is stopped (**Idle** or **Ready**) and has no NVM work outstanding — not only while the motor is disabled
- Propagating motor model parameters from the online RLS estimators to the newly selected controller immediately after selection
- Persisting the active algorithm identifier for each loop to non-volatile memory upon each selection
- Restoring the persisted algorithm identifiers at boot and re-configuring the controllers from NVM before the first Enable command
- Exposing algorithm selection to the CLI terminal

**Is NOT responsible for:**

- Implementing the control algorithms themselves — each concrete controller is self-contained
- Executing FOC transforms (Clarke, Park, SVM) — these remain inline in the FOC loop
- Estimating motor model parameters — that is the responsibility of the RLS identification services
- Safety monitoring or fault detection — the FOC state machine owns these
- Tuning algorithm-specific parameters beyond what is derivable from motor model parameters

---

## Component Details

### Part A — Algorithm Enumeration

Each of the three FOC loops has an independent enumeration of available algorithm choices.

Each loop has its own independent algorithm enumeration. An algorithm is only valid for the loop it is listed under; there is no sharing of algorithm identifiers across loops.

**Current loop** — executes at 20 kHz in the FOC ISR:

| Algorithm     | Description                                             |
|---------------|---------------------------------------------------------|
| PID           | Incremental PI baseline                                 |
| Decoupled PID | PI plus cross-coupling and back-EMF feedforward         |
| Deadbeat      | One-step or two-step predictive voltage inversion       |
| Sliding-mode  | Robust switching control with boundary-layer saturation |

**Speed loop** — executes at 1 kHz in the low-priority handler:

---

| Algorithm | Description                                                        |
|-----------|--------------------------------------------------------------------|
| PID       | Incremental PI baseline                                            |
| LQI       | DARE-computed state-feedback with integral augmentation            |
| ADRC      | Extended-state observer + active disturbance cancellation          |
| Two-DOF   | Reference pre-filter + PI; decoupled tracking and stiffness tuning |

---

**Position loop** — executes at 1 kHz in the low-priority handler:

---

| Algorithm | Output            | Description                                                        |
|-----------|-------------------|--------------------------------------------------------------------|
| PID       | Speed reference   | Incremental PI on the wrapped position error                       |
| Cascade P | Speed reference   | Industry-standard P position $\rightarrow$ speed loop; single gain |
| LQR       | Current reference | DARE-computed state feedback on $(\theta, \omega)$                 |
| LQI       | Current reference | LQR with position integral augmentation                            |
| Two-DOF   | Speed reference   | Reference pre-filter + PI; tracking and stiffness tune apart       |

---

Each position algorithm declares whether it produces a speed reference or a current reference. A speed reference runs through the existing speed and current loops unchanged. A current reference bypasses the speed loop and feeds the current loop directly, because the state feedback laws already regulate speed as part of their own state vector; running them on top of a speed loop would stack two regulators on the same state.

The enumeration for each loop is fixed at build time. The default for all loops at first boot is PID.

---

## Part B — Heap-Free Variant Storage

All concrete controller instances for a given loop are held simultaneously in a single fixed-size discriminated union (a type-safe union that tracks which alternative is currently active). Only the active alternative is live at any time; inactive alternatives exist as constructed but dormant objects.

Switching the active algorithm constructs the new controller in place inside the union, destroying the previous one, with no dynamic memory allocation. The total storage required is the size of the largest concrete controller in the set, determined at build time.

This pattern is identical to the one already used by `ControlModeStateMachine` for switching between Torque, Speed, and Position control modes.

---

## Part C — Hot-Path Dispatch via Type-Aware Visit

The per-sample `Compute` call — which runs at 20 kHz in the current loop ISR — must not incur indirect dispatch through a virtual base-class pointer, as this prevents the compiler from inlining and optimising the critical arithmetic.

Dispatch is performed by visiting the discriminated union: the visitor examines the stored type tag and jumps directly to the concrete implementation, allowing the compiler to devirtualize the call and inline the arithmetic. The resulting machine code is equivalent to a switch table with a direct

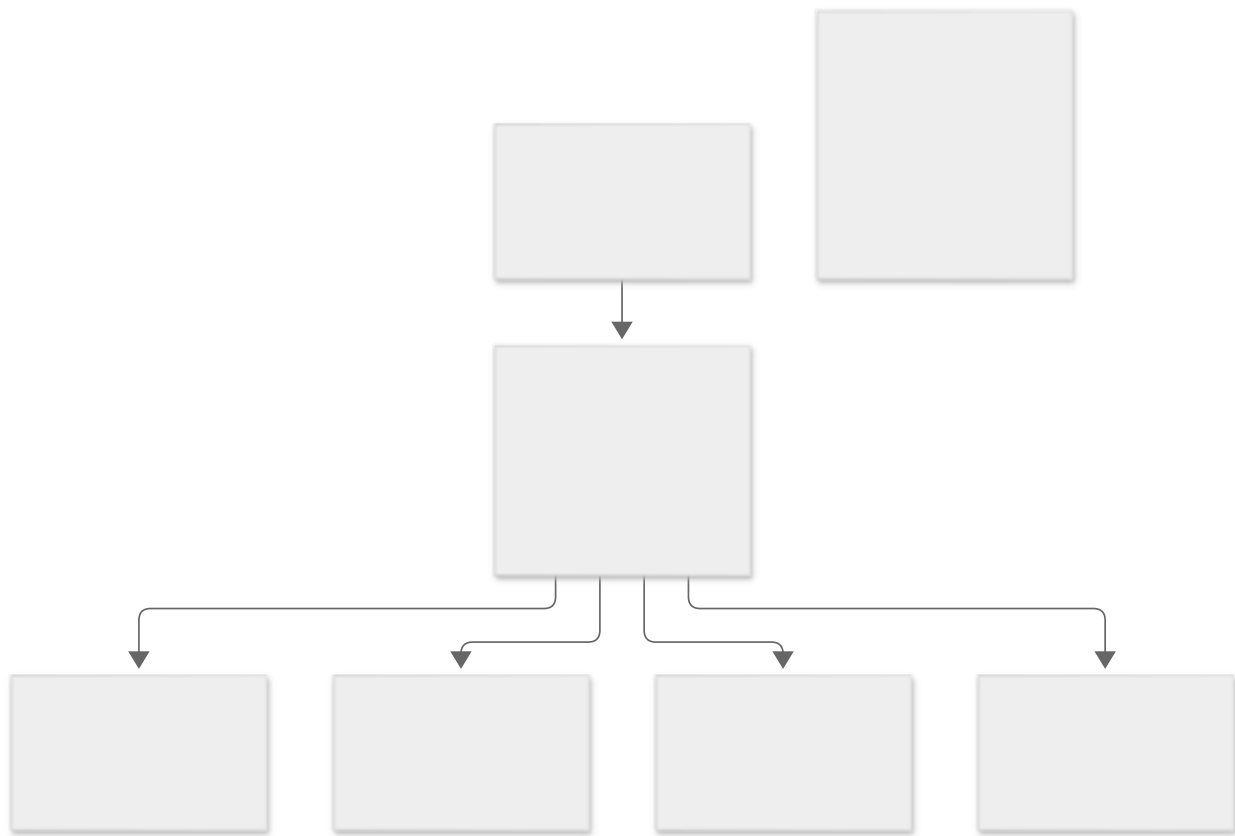


Figure 36: Block diagram

function body inlined at each case, not a vtable indirection. There is no allocation and no heap touch in this path.

The visit call is the only place where **Compute** is invoked in the ISR; all controller-specific setup, gain computation, and model updates occur off the hot path in the configuration routine.

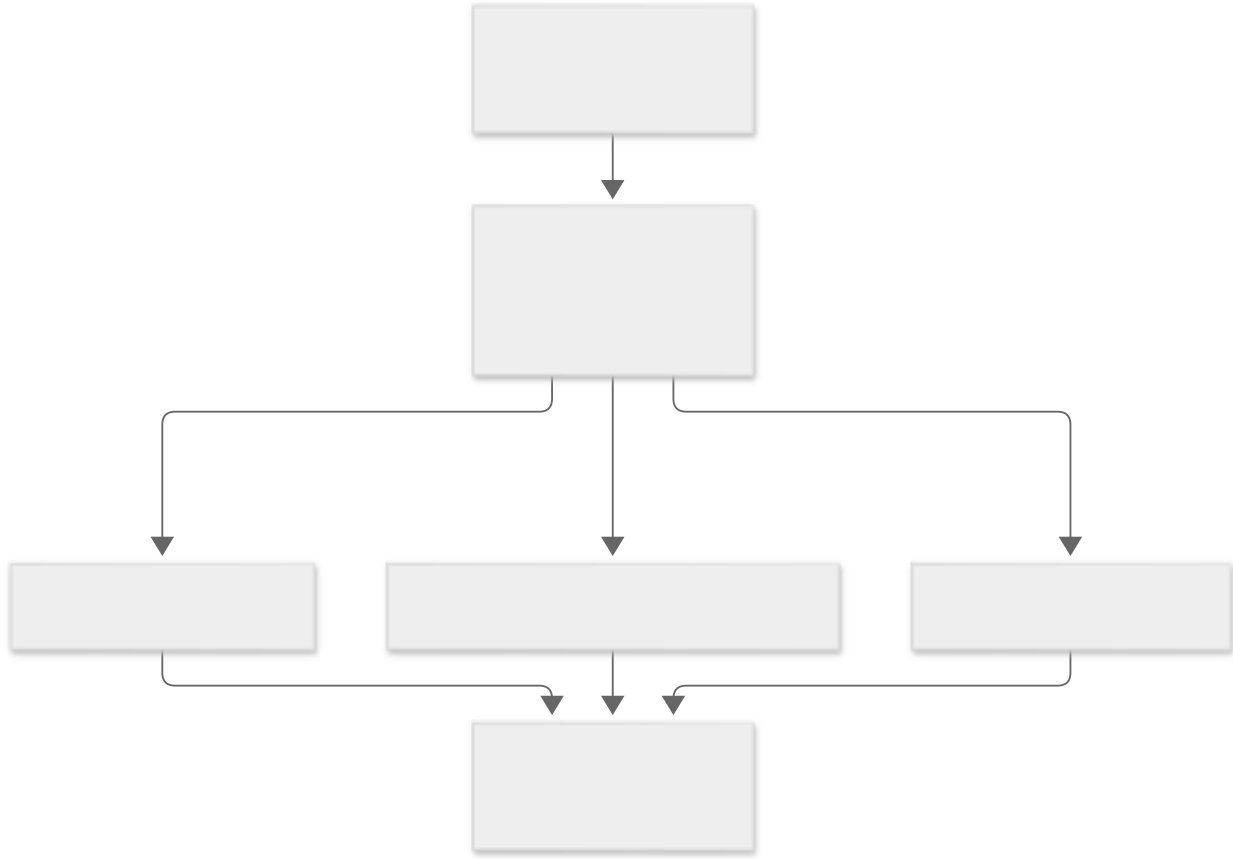


Figure 37: Block diagram

## Part D — State Gating

Algorithm selection is a configuration-time operation, not a run-time one. It is only permitted while the active mode's lifecycle state machine is **stopped** (**Idle** or **Ready**) and has no asynchronous work outstanding, including any write to the shared non-volatile memory. A request arriving while the motor is **Enabled**, **Calibrating** or in **Fault**, or while such a write is outstanding, is rejected immediately with **Busy** without altering the active algorithm or any controller state.

The guard is the same one `ControlModeStateMachine::Select` applies to mode changes (`documentation/design/state-machine.md` C2b). The cascades' own refusal while enabled remains underneath it.

On a successful selection, the sequence is: 1. Construct the new controller in place in the union (destroying the previous one). 2. Update the active algorithm identifier. 3. Configure the new

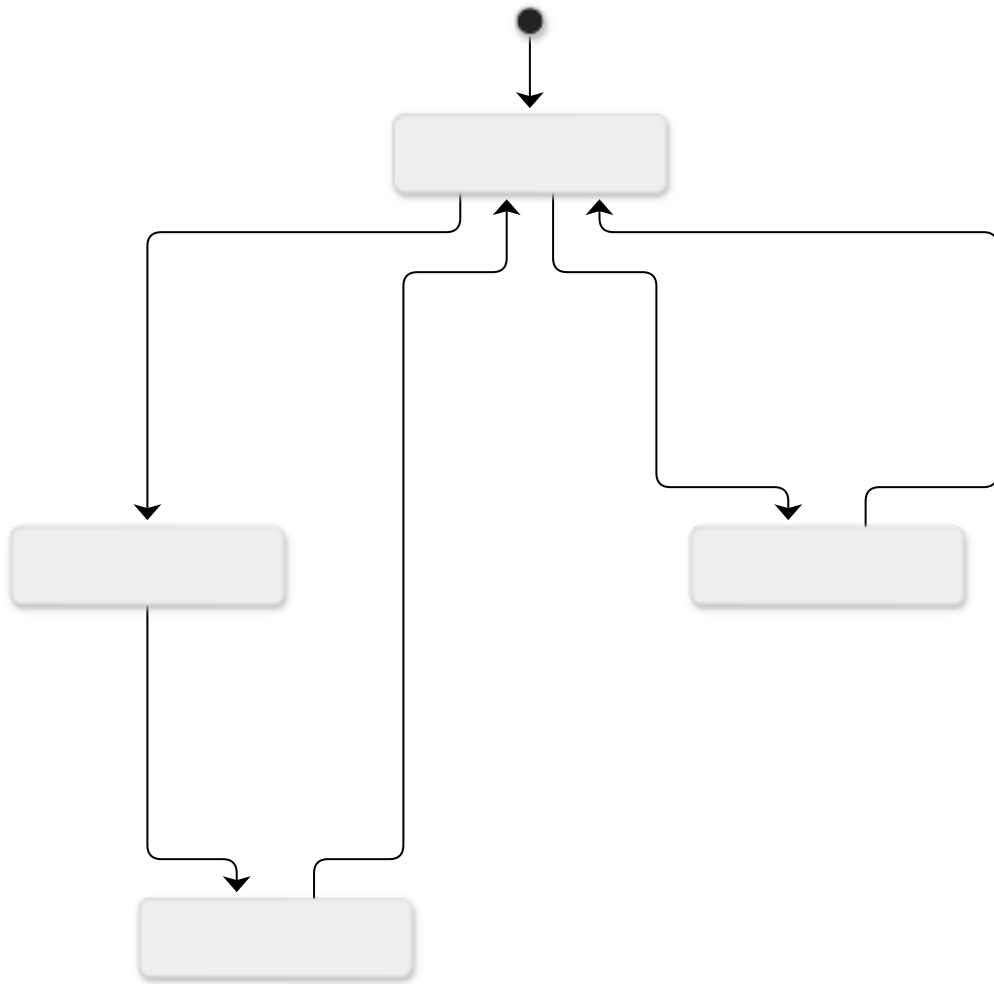


Figure 38: State diagram

controller with the current motor model parameter snapshot from RLS. 4. Reset the new controller's internal integrators and observer state to zero. 5. Persist the algorithm identifier to NVM.

Step 4 (Reset) ensures the controller starts from a clean state regardless of what state the motor or the previous controller was in.

---

## Part D2 — Design Feasibility for the Position State Feedback Laws

The position plant is a near double integrator, which makes its discrete Riccati solve far more fragile than the first-order speed plant. Solved on the raw state  $(\theta, \omega)$  at a 1 kHz outer rate it does not converge at all in single precision. Three conditioning measures make it tractable:

1. **Time-scaled state.** The design runs on  $(\theta, \omega T_s)$  rather than  $(\theta, \omega)$ , so the state matrix becomes  $A = \begin{bmatrix} 1 & 1 \\ 0 & 1 - \frac{B_f T_s}{J} \end{bmatrix}$  with every entry near unity.
2. **Normalised input.** The input matrix is fixed at  $B = [0 \ 1]^T$  by folding the actuator authority  $\frac{K_t}{J} T_s^2 I_{max}$  into a scalar applied to the resulting command. The Riccati recursion therefore never sees the motor's torque scale.
3. **Bandwidth-derived effort weight.**  $R = 1/(\omega_{bw} T_s)^2$ , with  $\omega_{bw} T_s$  clamped to  $[10^{-3}, 0.5]$ . This makes settling time scale as  $2/\omega_{bw}$  seconds with roughly 4 % overshoot across four decades of bandwidth, so **bandwidth** remains the single aggressiveness knob it is for every other loop.

The state weights are shape knobs only. `positionErrorWeight` anchors the cost at one and the remaining weights enter as ratios against it, clamped to  $[10^{-3}, 10^3]$ . Anchoring matters for more than conditioning: if the position entry of  $Q$  can reach zero the cost stops penalising the integrator state and the recursion has no stabilising solution to find.

These measures shrink but do not eliminate non-convergence. `Lqr::TryCreate` is therefore used throughout in preference to the aborting constructor, and feasibility is part of the selection rule rather than an afterthought:

- `PositionControllerTraits::IsSelectable` runs the actual Riccati solve for LQR and LQI. A selection whose design does not converge returns `invalidParameters` and **leaves the previously active algorithm running**, so the motor never ends up holding a silently dead loop.
- `SetPositionTunings` is refused outright while the motor is enabled, returning `busy`. A live motor is not a safe place to redesign the law that is holding it.
- When disabled, `SetPositionTunings` revalidates the new tunings against the active algorithm and rejects them if the design fails, keeping the last accepted gains live.

This is why `SetPositionTunings` returns `SelectResult` where the current and speed loops return `void`: those loops derive their gains algebraically and cannot fail.

`IntegralStateFeedbackLqi` is not used for the LQI position law even though it now offers a `TryCreate` factory. It augments the plant with  $C \cdot (-Ts)$ , which reintroduces the sample period into the integral row and undoes the time scaling above; at a 1 kHz outer rate that row is three orders of magnitude smaller than the rest of the matrix and the solve stops converging. The augmentation is therefore written out explicitly in the scaled coordinates, where the integral state simply

accumulates the position deviation per sample, and solved with `Lqr<float, 3, 1>::TryCreate`.

## Part E — Motor Model Parameter Flow

When an algorithm is selected, it is immediately configured using the most recent snapshot of the motor model parameters. The parameters are drawn from two sources:

- **Electrical parameters** ( $R_s$ ,  $L_s$ ,  $\psi_f$ ,  $V_{dc}$ , pole pairs): from the electrical RLS identification service, as set by the calibration sequence.
- **Mechanical parameters** ( $J$ ,  $B_f$ ,  $K_t$ ): from the mechanical RLS identification service, available after the speed/position calibration phase.

The configuration call is a one-time operation that computes all gains and pre-computed matrices (e.g., DARE solution for LQR/LQI) before the controller becomes active. The ISR then executes only the pre-computed gain application.

If the RLS estimators produce updated parameter estimates during operation (continuous identification mode), the active controllers are not automatically reconfigured. Reconfiguration with updated parameters requires a new `SelectAlgorithm` call (or an explicit `ReconfigureFromCurrentParameters` command), which requires the motor to be disabled.

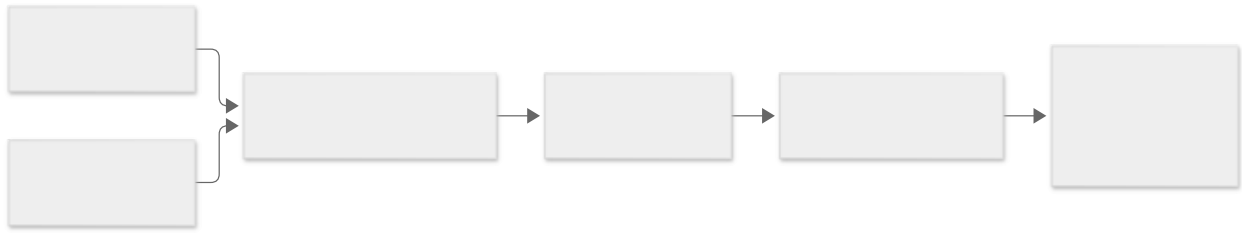


Figure 39: Block diagram

## Part F — CLI Interface

`ControlModeStateMachine` registers the selection commands on the terminal.

| Command                                            | Alias            | Effect                                    |
|----------------------------------------------------|------------------|-------------------------------------------|
| <code>select_current_algorithm &lt;alg&gt;</code>  | <code>sca</code> | Selects the current-loop algorithm        |
| <code>select_speed_algorithm &lt;alg&gt;</code>    | <code>ssa</code> | Selects the speed-loop algorithm          |
| <code>select_position_algorithm &lt;alg&gt;</code> | <code>spa</code> | Selects the position-loop algorithm       |
| <code>active_algorithms</code>                     | <code>aa</code>  | Prints the active algorithm for each loop |

Valid `<alg>` tokens are the lower-case algorithm names: `pid`, `decoupled`, `deadbeat` and `sliding` for the current loop; `pid`, `lqi`, `adrc` and `twodof` for the speed loop; and `pid`, `cascadep`, `lqr`, `lqi` and `twodof` for the position loop. Unknown tokens produce a usage error without altering state.

The commands enforce the motor-enabled guard described in Part D and report the `SelectResult` verbatim, so an operator always learns why a selection was refused.

---

## Part G — CAN Interface

Algorithm selection is **not** exposed over CAN. The `can-lite foc_motor` category defines no frame for it, and the CAN bridge does not reinterpret an existing frame to carry an algorithm identifier. Selection and query are CLI-only; see REQ-CTRL-001 and REQ-CTRL-013.

---

## Part H — Required Parameters Per Controller

Each controller algorithm has specific parameter dependencies. These dependencies determine what must be available before that algorithm can be selected, and who is responsible for providing each value.

### Parameter Sources

Parameters come from three distinct sources with different availability windows:

| Source                     | Parameters                                                        | Available after                 |
|----------------------------|-------------------------------------------------------------------|---------------------------------|
| Motor datasheet (static)   | Pole pairs $p$ , flux linkage $\psi_f$ , peak current $I_{q,max}$ | Before any run                  |
| Electrical RLS calibration | $R_s, L_s, V_{dc}$                                                | After electrical identification |
| Mechanical RLS calibration | $J, B_f, K_t = \frac{3}{2}p\psi_f$                                | After mechanical identification |

$\psi_f$  is the only datasheet value not directly observable by the RLS estimators. If unavailable from the datasheet, it can be derived from the rated torque constant:  $\psi_f = K_{t, rated} \cdot 2/(3p)$ .

$V_{dc}$  is measured dynamically during operation; a nominal value must be known at configuration time for gain normalisation.

### Per-Algorithm Parameter Requirements

#### Current loop:

| Parameter                   | PID                      | Decoupled PID            | Deadbeat                 | Sliding-mode              |
|-----------------------------|--------------------------|--------------------------|--------------------------|---------------------------|
| $R_s, L_s$ (electrical RLS) | Required                 | Required                 | Required (tight)         | Required                  |
| $\psi_f$ (datasheet)        | Not required             | Required — back-EMF FF   | Not required             | Required — back-EMF FF    |
| $V_{dc}$                    | Required — normalisation | Required — normalisation | Required — normalisation | Required — normalisation  |
| $I_{q,max}$                 | For output clamp         | For output clamp         | For clamp                | For switching gain sizing |

Deadbeat requires the tightest RLS convergence.

Sliding-mode applies the same decoupling feedforward as Decoupled PID, and for the same reason: its sliding surface is designed on the decoupled per-axis RL plant, so the cross-coupling and the back-EMF are disturbance to it. Sizing the switching gain to reject them outright is not viable — at 1000 rad/s electrical on a 0.01 Wb rotor the back-EMF alone is several times the authority the default gain commands, and raising the gain that far would put the boundary-layer ratio outside the range in which the error map contracts. Feeding them forward leaves the switching term the residue it is sized for.

### Speed loop:

| Parameter                 | PID          | LQI                   | ADRC                     | Two-DOF      |
|---------------------------|--------------|-----------------------|--------------------------|--------------|
| $J, B_f$ (mechanical RLS) | Not required | Required — DARE plant | Required — $b_0 = K_t/J$ | Not required |
| $K_t$ (derived)           | Not required | Required              | Required                 | Not required |
| $I_{q,max}$               | For clamp    | Required — R weight   | For clamp                | For clamp    |

ADRC is the most forgiving:  $b_0$  tolerates  $\pm 50\%$  error. Two-DOF and PID require no mechanical RLS and are available as soon as electrical calibration completes.

### Position loop:

| Parameter                 | PID          | Cascade P       | LQR                 | LQI                 | Two-DOF      |
|---------------------------|--------------|-----------------|---------------------|---------------------|--------------|
| $J, B_f$ (mechanical RLS) | Not required | Not required    | Required            | Required            | Not required |
| $K_t$ (derived)           | Not required | Not required    | Required            | Required            | Not required |
| $I_{q,max}$               | For clamp    | Speed-loop dep. | Required — R weight | Required — R weight | For clamp    |

### Tuning Knobs (design choices, not estimated)

Each loop exposes one tuning record shared by all of its algorithms: a single closed-loop bandwidth plus a small set of law-specific weights. The record is not estimated — it encodes the operator's performance intent, and each algorithm reads only the fields that apply to it. Gains are always derived from the bandwidth and the motor model, never supplied per axis.

### Current loop:

| Knob             | Default                 | Consumed by        |
|------------------|-------------------------|--------------------|
| Bandwidth        | $2\pi \cdot 1000$ rad/s | PID, Decoupled PID |
| Switching gain   | 0.2                     | Sliding-mode       |
| Boundary layer   | 0.5                     | Sliding-mode       |
| Two-step variant | off                     | Deadbeat           |

The sliding-mode error map inside the boundary layer contracts only while the switching gain stays below the boundary layer, so the ratio of the two is bounded by design. This is why the back-EMF is handled by feedforward rather than by a larger switching gain.

#### Speed loop:

| Knob                     | Default               | Consumed by |
|--------------------------|-----------------------|-------------|
| Bandwidth                | $2\pi \cdot 30$ rad/s | PID, ADRC   |
| Speed error weight       | 1.0                   | LQI         |
| Integral weight          | 0.1                   | LQI         |
| Observer bandwidth ratio | 5.0                   | ADRC        |
| Reference time constant  | 5.3 ms                | Two-DOF     |

#### Position loop:

| Knob                    | Default              | Consumed by              |
|-------------------------|----------------------|--------------------------|
| Bandwidth               | $2\pi \cdot 3$ rad/s | PID, Cascade P, LQR, LQI |
| Position error weight   | 1.0                  | PID, LQR, LQI            |
| Speed error weight      | 0.1                  | LQR, LQI                 |
| Integral weight         | 0.05                 | PID, LQI                 |
| Reference time constant | 53 ms                | Two-DOF                  |

Two-DOF on both outer loops derives its feedback part from the same bandwidth as the corresponding PID law and adds only the reference pre-filter time constant on top.

Of these knobs only the current loop bandwidth and the speed loop bandwidth are persisted, as part of the calibration record. The remaining weights and the position loop bandwidth are not stored and return to their defaults on boot.

#### Readiness Gate per Algorithm

The selection guard (Part D) enforces not only the stopped-and-not-busy check but also a parameter-readiness check. Attempting to select an algorithm whose required parameters are not yet available returns `InvalidParameters` rather than `ok`.

The readiness state for each parameter source is:

| Source                           | Readiness condition                                                           |
|----------------------------------|-------------------------------------------------------------------------------|
| Datasheet params ( $p, \psi_f$ ) | Set during motor configuration before calibration                             |
| Electrical RLS                   | Calibration sequence reached and completed the electrical identification step |
| Mechanical RLS                   | Calibration sequence reached and completed the mechanical identification step |

Current-loop algorithms (Decoupled PID, SMC) become selectable after electrical identification. Speed and position algorithms (LQI, ADRC, LQR) additionally require mechanical identification.

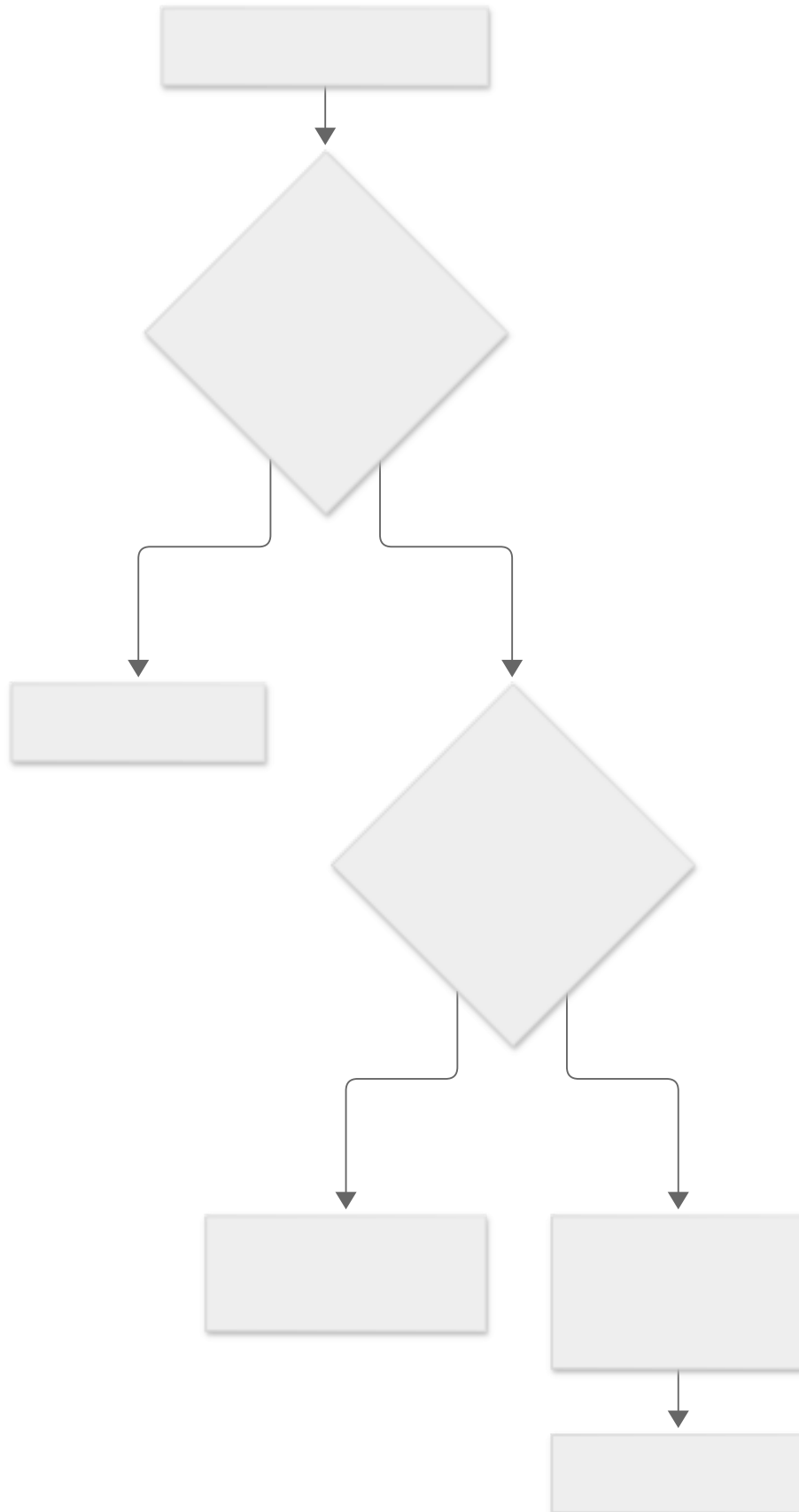


Figure 40: Block diagram

## Part I — NVM Persistence

The active algorithm identifier for each loop is stored as three additional fields in the existing `ConfigData` structure alongside `defaultControlMode`. On boot:

1. NVM is read and validated.
2. If valid, the persisted algorithm identifiers are loaded and used to configure each loop's controller using the also-persisted motor model parameters.
3. If invalid (first boot, CRC mismatch), the default (PID for all loops) is used and written.

Restoration is driven by the state machine's transition into **Ready**, not by construction of the control mode: the motor model only reaches the loops when the calibration record has been applied, which happens asynchronously after the NVM read completes. Applying the persisted identifiers any earlier would reject every model-dependent algorithm (deadbeat, decoupled PID, speed LQI, speed ADRC, position LQR, position LQI).

Two failure kinds are distinguished when a persisted identifier cannot be activated:

| Failure                                                                | Effect on <code>ConfigData</code>                             |
|------------------------------------------------------------------------|---------------------------------------------------------------|
| Byte out of enum range, or names no algorithm for that loop            | Corrected to the active algorithm — the record is meaningless |
| Valid algorithm, not selectable yet ( <code>InvalidParameters</code> ) | Preserved, and retried on the next entry to <b>Ready</b>      |

The non-volatile memory serves one write at a time and answers a concurrent one with *busy*. A selection's write that comes back busy is retried after a short delay until it lands; while it or any other write to that memory is outstanding, selections, mode changes and calibration are refused with *busy* ([documentation/design/state-machine.md](#) C2a).

The persistence ensures that an operator who selects ADRC for the speed loop does not need to repeat the selection after every power cycle. The motor state machine will transition directly to **Ready** with the previously selected algorithms active, ready to enable.

## Interfaces

### Provided

| Interface                            | Purpose                                                   | Contract                                                                                     |
|--------------------------------------|-----------------------------------------------------------|----------------------------------------------------------------------------------------------|
| <code>SelectCurrentAlgorithm</code>  | Selects the active current-loop controller by enum value  | <b>busy</b> unless stopped with no pending work (Part D). Effect is immediate if <b>ok</b> . |
| <code>SelectSpeedAlgorithm</code>    | Selects the active speed-loop controller by enum value    | As above.                                                                                    |
| <code>SelectPositionAlgorithm</code> | Selects the active position-loop controller by enum value | As above.                                                                                    |

| Interface                        | Purpose                                                                                 | Contract                                                   |
|----------------------------------|-----------------------------------------------------------------------------------------|------------------------------------------------------------|
| ActiveCurrentAlgorithm           | Returns the currently active current-loop algorithm identifier                          | Always returns a valid enum value. Never blocks.           |
| ActiveSpeedAlgorithm             | Returns the currently active speed-loop algorithm identifier                            | As above.                                                  |
| ActivePositionAlgorithm          | Returns the currently active position-loop algorithm identifier                         | As above.                                                  |
| ComputeCurrentControl            | Dispatches to the active current controller for one sample                              | Called at 20 kHz from ISR. Zero heap. No virtual dispatch. |
| ComputeSpeedControl              | Dispatches to the active speed controller for one sample                                | Called at 1 kHz from low-priority handler.                 |
| ComputePositionControl           | Dispatches to the active position controller for one sample                             | Called at 1 kHz from low-priority handler.                 |
| ReconfigureFromCurrentParameters | Re-applies the latest RLS snapshot to the active controllers without changing algorithm | Motor must be disabled.                                    |

## Required

| Interface               | Purpose                                                                              | Contract                                                                            |
|-------------------------|--------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|
| IsMotorEnabled          | Checks whether the motor state machine is in the Enabled state                       | Used to enforce the selection guard. Must be callable from off the ISR.             |
| GetElectricalParameters | Returns the current RLS estimate of $R_s$ , $L_s$ , $\psi_f$ , $V_{dc}$ , pole pairs | Called during Select and ReconfigureFromCurrentParameters.                          |
| GetMechanicalParameters | Returns the current RLS estimate of $J$ , $B_f$ , $K_t$                              | Called during Select and ReconfigureFromCurrentParameters for speed/position loops. |
| NvmWrite                | Persists the algorithm identifiers to non-volatile storage                           | Called once per successful selection. Must not block the hot path.                  |
| NvmRead                 | Reads persisted algorithm identifiers at boot                                        | Called during boot initialization before any Enable.                                |

## Data Model

| Entity            | Field | Type / Unit | Range                                    | Notes                             |
|-------------------|-------|-------------|------------------------------------------|-----------------------------------|
| CurrentAlgorithm  | enum  | uint8       | pid, decoupledPid, deadbeat, slidingMode | Selectable current-loop strategy  |
| SpeedAlgorithm    | enum  | uint8       | pid, lqi, adrc, twoDof                   | Selectable speed-loop strategy    |
| PositionAlgorithm | enum  | uint8       | pid, cascadeP, lqr, lqi, twoDof          | Selectable position-loop strategy |

| Entity               | Field                | Type /<br>Unit                                                 | Range                                               | Notes                                                             |
|----------------------|----------------------|----------------------------------------------------------------|-----------------------------------------------------|-------------------------------------------------------------------|
| PositionOutputKind   | uint8                | uint8                                                          | speedReference,<br>currentReference                 | Declares whether a position law drives the speed or current loop  |
| SelectResult         | enum                 | uint8                                                          | ok, busy,<br>invalidAlgorithm,<br>invalidParameters | Return code from<br>SelectAlgorithm                               |
| MotorModelParameters | psiF                 | Ohm<br>(float)                                                 | $> 0$                                               | From electrical RLS                                               |
| MotorModelParameters | psiF                 | Henry<br>(float)                                               | $> 0$                                               | From electrical RLS                                               |
| MotorModelParameters | psiF                 | Weber<br>(float)                                               | $> 0$                                               | From alignment calibration                                        |
| MotorModelParameters | polePairs            | uint8                                                          | $\geq 1$                                            | Motor constant                                                    |
| MotorModelParameters | voltage              | Volt<br>(float)                                                | $> 0$                                               | Measured dynamically                                              |
| MotorModelParameters | currentLoopRate      | Hertz<br>(uint32)                                              | $> 0$                                               | $1/T_s^i$ ; required to discretize the plant for Deadbeat and SMC |
| MotorModelParameters | psiF                 | $\text{kg} \cdot \text{m}^2$<br>(float)                        | $> 0$                                               | From mechanical RLS                                               |
| MotorModelParameters | psiF                 | $\text{N} \cdot \text{m} \cdot \text{s}/\text{rad}$<br>(float) | $\geq 0$                                            | From mechanical RLS                                               |
| MotorModelParameters | psiF                 | $\text{N} \cdot \text{m}/\text{A}$<br>(float)                  | $> 0$                                               | Derived from psiF and polePairs                                   |
| NvmRecord            | currentAlgorithm     | uint8                                                          | 0–255                                               | Validated on load; invalid $\rightarrow$ default (PID)            |
| NvmRecord            | speedAlgorithm       | uint8                                                          | 0–255                                               | As above                                                          |
| NvmRecord            | positionAlgorithm    | uint8                                                          | 0–255                                               | As above                                                          |
| CalibrationRecord    | currentLoopBandwidth | Bandwidth<br>(float)                                           | $\geq 0$                                            | Persisted current loop bandwidth                                  |
| CalibrationRecord    | speedLoopBandwidth   | Bandwidth<br>(float)                                           | $\geq 0$                                            | Persisted speed loop bandwidth                                    |

## State Machine

The controller selection component has no states of its own — it is stateless except for the union’s active alternative and the active enum values. Its behaviour is governed entirely by the motor state machine’s current state (stopped with no pending NVM work, vs. enabled, calibrating, faulted, or NVM-busy — Part D).

The sequence on a successful algorithm selection:

The sequence on a rejected selection (motor enabled):

Boot restoration sequence:

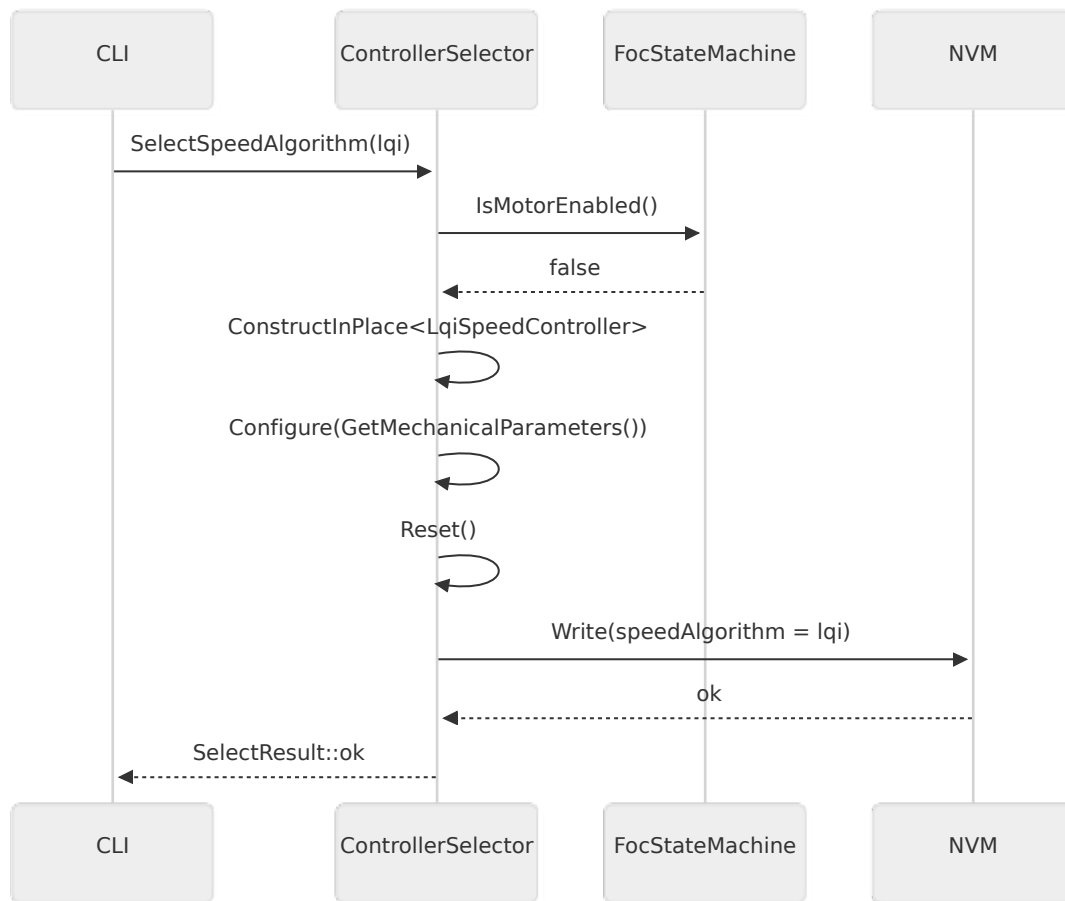


Figure 41: Sequence diagram

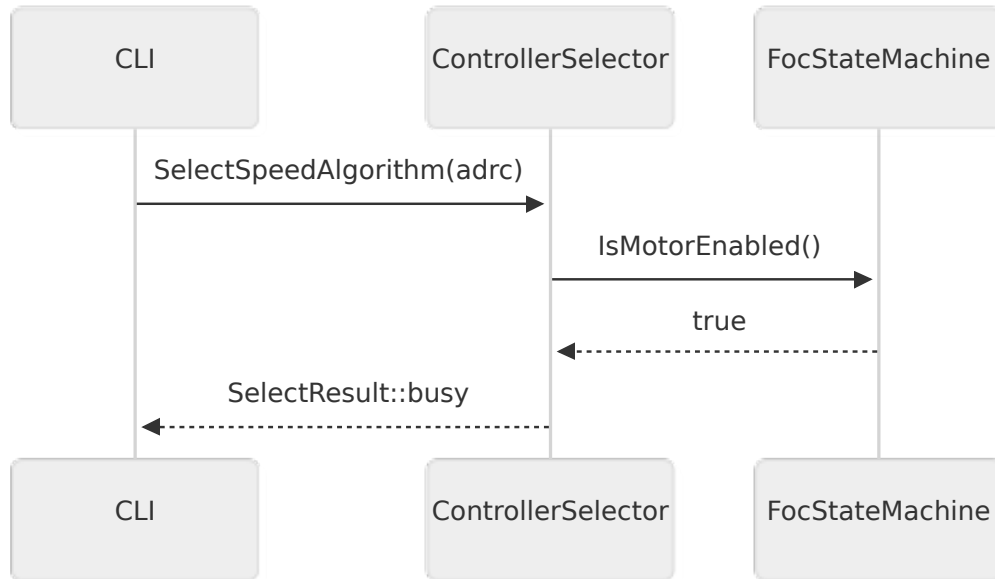


Figure 42: Sequence diagram

## Block Diagram

## Constraints & Limitations

| Constraint                   | Value / Description                                                                                                                              |
|------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------|
| Heap allocation              | Zero. All controller instances reside in fixed-size, statically-allocated storage.                                                               |
| Hot-path virtual dispatch    | None. The visit dispatch inlines directly to concrete implementations.                                                                           |
| Selection while not stopped  | Rejected (Part D). <code>SelectResult::busy</code> returned; no state change.                                                                    |
| Current loop cycle overhead  | Visit dispatch adds $\leq 10$ cycles versus direct call on the measured cycle budget.                                                            |
| Maximum number of algorithms | Fixed at build time by the union type list. Adding a new algorithm requires a rebuild.                                                           |
| NVM write latency            | NVM write occurs synchronously after selection; motor must remain disabled during the write.                                                     |
| Parameter staleness          | Controllers are configured with the RLS snapshot at selection time. They are not automatically updated if RLS estimates change during operation. |

| Constraint                     | Value / Description                                                                                                                                                                                                                                                   |
|--------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Mechanical parameters required | Speed and position algorithm configuration requires that the mechanical RLS identification has been completed. Attempting to select LQI or LQR before mechanical calibration is done produces an <b>InvalidParameters</b> result.                                     |
| DARE numerical stability       | For extremely large inertia ( $B_d \rightarrow 0$ ) or zero viscous friction ( $A_d \rightarrow 1$ ), the DARE solution may be ill-conditioned. A controllability pre-check is performed before solving; an ill-conditioned system returns <b>InvalidParameters</b> . |

## Open Questions

| # | Question                                                                                                                                                                                             | Options                                                                                    | Status |
|---|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------|--------|
| 1 | Should the CAN <b>SetControllerAlgorithm</b> frame be acknowledged synchronously (blocking until NVM write completes) or asynchronously (NVM write deferred)?                                        | Synchronous (simpler, adds latency) vs. asynchronous (lower bus latency, NVM write queued) | open   |
| 2 | Should <b>ReconfigureFromCurrentParameters</b> be exposed on the CLI, or only as a CAN command?                                                                                                      | CLI only / CAN only / both                                                                 | open   |
| 3 | When the motor transitions from Speed mode to Position mode, should the position-loop LQR gains be recomputed from the speed-mode ADRC's parameter snapshot, or should they be stored independently? | Independent storage (current design) vs. shared parameter snapshot                         | open   |

| Field     | Value                      |
|-----------|----------------------------|
| Title     | Service: FOC State Machine |
| Type      | design                     |
| Status    | draft                      |
| Version   | 0.3.0                      |
| Component | state-machine              |
| Date      | 2026-09-21                 |

**IMPORTANT — Implementation-blind document:** This document describes *behavior, structure, and responsibilities* WITHOUT referencing code. **No code blocks using programming languages (C++, C, Python, CMake, shell, etc.) are allowed.** Use Mermaid diagrams to express behavior instead. Prose descriptions of algorithms are encouraged; source-level details are not.

**Diagrams:** All visuals must be either a Mermaid fenced code block (````mermaid`) or ASCII art inline in the document. External image references (`![alt](/home/runner/work/e-foc/e-foc/documentation/design/path)`) are **not allowed**.

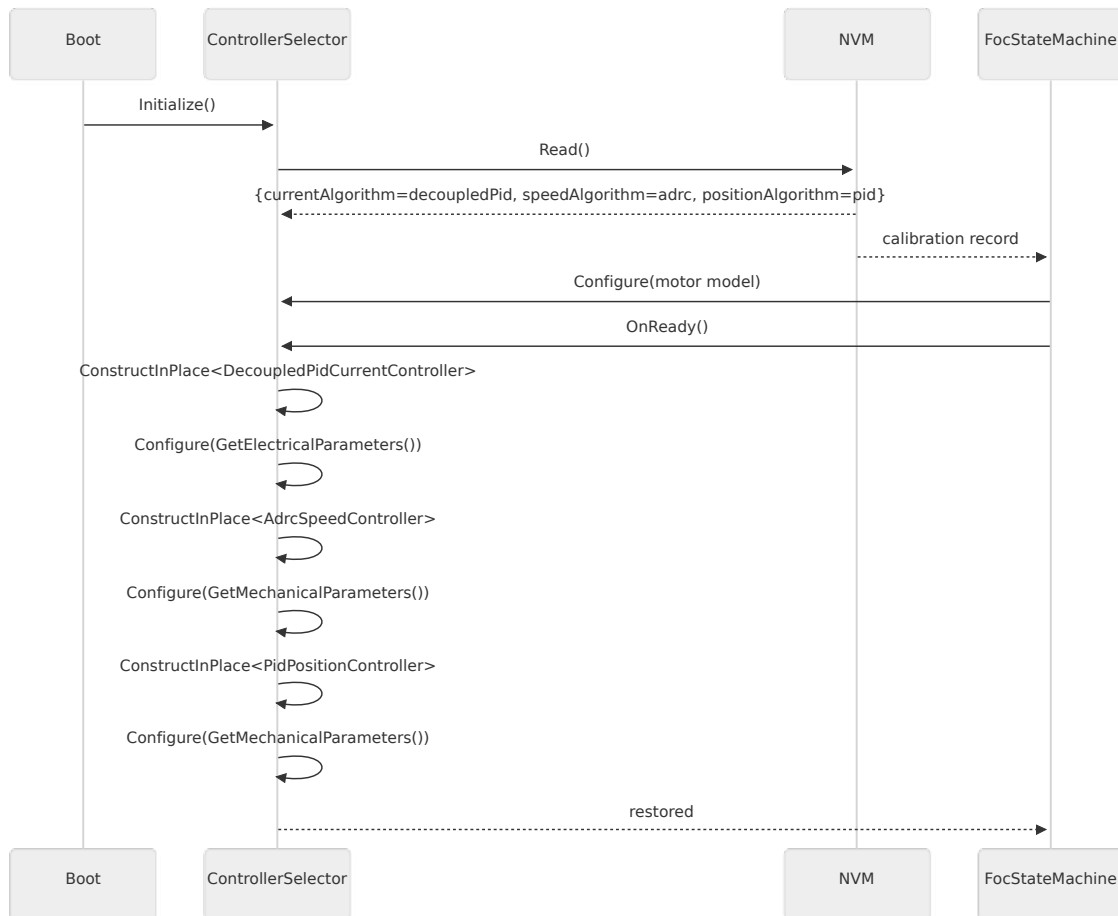


Figure 43: Sequence diagram

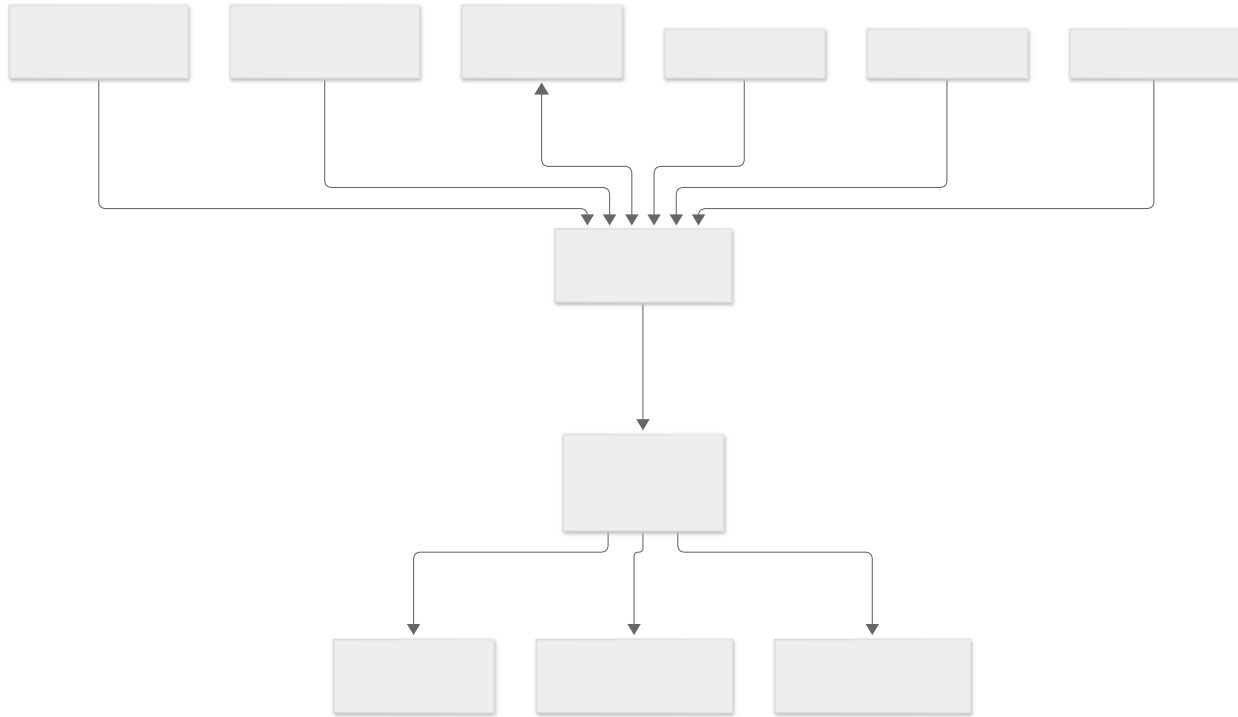


Figure 44: Block diagram

## Responsibilities

**Is responsible for:** - Owning the complete motor lifecycle: **Idle** → **Calibrating** → **Ready** ⇌ **Enabled**, and **Fault** as an escape state reachable from any active state - Enforcing all transition guards so that the FOC controller can only be enabled after a successful calibration, and calibration can only be started from **Idle** or **Ready** - Orchestrating the sequential calibration chain: pole-pair identification → resistance and inductance estimation → alignment → (mechanical parameter identification for speed/position modes) → NVM persistence - On successful boot with valid NVM data, loading electrical parameters (R, L, pole pairs) and remaining in **Idle** until a live alignment re-establishes the rotor reference; the motor cannot be enabled until **CmdReAlign()** or **CmdCalibrate()** succeeds - Registering lifecycle commands on the terminal in CLI-driven mode (**calibrate**, **align**, **enable**, **disable**, **clear\_fault**, **clear\_cal**) - Intercepting hardware fault notifications from the fault notifier: stopping the inverter in the delivering context, then transitioning to **Fault** on the event dispatcher (see Fault Safety)

**Is NOT responsible for:** - Executing FOC calculations — those remain the responsibility of the control-mode implementation (Torque, Speed, Position) - Directly driving hardware peripherals — all hardware interaction is delegated to the calibration services and the FOC controller - Storing persistent calibration data itself — that is the responsibility of the Non-Volatile Memory service - Implementing the identification algorithms — those are encapsulated in the Electrical and Mechanical Parameters Identification services

## Component Details

### Motor Lifecycle States

The state machine has five named states:

| State              | Motor condition                                                                                                                       | Allowed transitions                                                                                                                                                                                                                                                                 |
|--------------------|---------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Idle</b>        | No calibration data, or electrical parameters loaded but rotor reference not yet established; motor cannot be enabled                 | → <b>Calibrating</b> (CmdCalibrate, CmdReAlign with loaded parameters, or CmdReserveExternalCalibration), → <b>Idle</b> (CmdClearCalibration), → <b>Fault</b> (hardware fault)                                                                                                      |
| <b>Calibrating</b> | Calibration sequence in progress; motor is driven by identification services, under a provisional plant model for the mechanical step | → <b>Ready</b> (sequence complete + NVM saved, or CmdEmergencyStop with previously valid calibration), → <b>Idle</b> (record saved but incomplete for this mode, or CmdEmergencyStop without valid calibration), → <b>Fault</b> (any step or the NVM save fails, or hardware fault) |
| <b>Ready</b>       | Calibration data valid, rotor reference established; motor can be enabled                                                             | → <b>Enabled</b> (CmdEnable, only when <code>rotorReferenceValid</code> is true), → <b>Calibrating</b> (CmdCalibrate re-runs, CmdReAlign, or CmdReserveExternalCalibration), → <b>Idle</b> (CmdClearCalibration), → <b>Fault</b> (hardware fault, or the NVM invalidation failing)  |
| <b>Enabled</b>     | FOC controller active; motor under closed-loop control                                                                                | → <b>Ready</b> (CmdDisable, or CmdEmergencyStop with valid calibration), → <b>Idle</b> (CmdEmergencyStop without valid calibration), → <b>Fault</b> (hardware fault)                                                                                                                |
| <b>Fault</b>       | Safe state; inverter stopped; fault code recorded and latched                                                                         | → <b>Ready</b> (CmdClearFault with valid calibration held), → <b>Idle</b> (CmdClearFault without it); at most 3 consecutive times. A further fault re-enters <b>Fault</b> but keeps the code that first tripped the drive                                                           |

### State Diagram

#### Transition Table

The graph above is not a description of hand-written command handlers; it is the transition table the FOC state machine is built from. Every operator command, every service completion and every fault notification is an **event**, and every arrow in the diagram is a **row** of the table: source state, event, target state, an optional guard and the action that builds the target state. Rows that must be accepted in a state without leaving it (a calibration sub-step changing, a flux-linkage save completing) are declared as internal rows.

The table is a compile-time constant. Its rows are `constexpr` values whose guards and actions are captureless functions receiving the state machine as their context, so the whole table lives in flash; the machine itself only holds the current state and a bounded queue of events. That is what keeps the lifecycle within the RAM of the 32 KB targets.

The queue depth is budgeted, not rounded up: each slot costs a whole **Event**, and overflowing it is an assertion failure rather than a dropped event. The deepest run of nested dispatches is a calibration sequence whose identification services report inline — the three steps the orchestrator announces

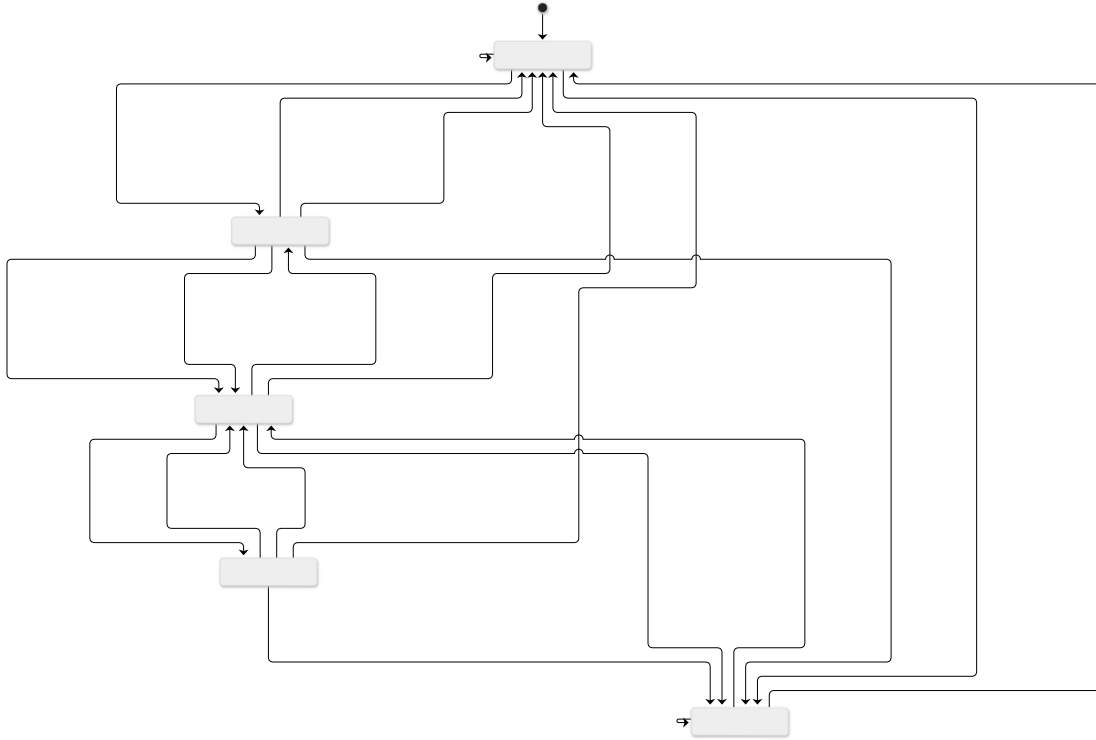


Figure 45: State diagram

plus the alignment result that follows the last one — which peaks at four. The friction-and-inertia step is not among them, because the control mode sets that step directly instead of announcing it, and the events that follow the alignment are pushed only once the queue has drained again. The remaining slots absorb a command chained from a completion callback, giving the six the machine reserves. Adding a calibration step that announces itself therefore costs a slot, and on the 32 KB targets a slot has to be paid for out of a RAM budget the linker now guards.

The table is the single source of truth for what the machine accepts:

- An event that arrives in a state with no row for it is **forbidden**: nothing changes, the command reports **rejected**, and the trace shows which event was refused in which state.
- An event whose rows exist but whose guards all refuse it is **rejected** in the same way; the guard is where conditions such as “asynchronous work is outstanding” or “the rotor reference is not established” live.
- The table is checked when the firmware is compiled, not when it runs: the machine can only be constructed from `services::Validated<FocLifecycleTable>()`, which evaluates the table with `static_assert`. Duplicate rows, rows that can never be selected because an unguarded row precedes them, states that no sequence of rows reaches from `Idle`, events that no row handles, states that no row leaves, and rows from any state that can never fire all stop the build, naming the offending row. A malformed lifecycle therefore never produces a binary.
- `FocLifecycleTable::Rules()` declares which state changes the lifecycle may make, separately from the rows: `Idle` → `Calibrating` or `Idle`; `Ready` → `Calibrating`, `Enabled` or `Idle`; `Calibrating`, `Enabled` and `Fault` → `Ready` or `Idle`; and any state → `Fault`. `Enabled` is forbidden from every state except `Ready`, and `Fault` never leads straight back to `Calibrating`. A row that makes any other transition, or an allowed transition that no row makes, is a compile

error.

Events are handled to completion. An event dispatched while another is being handled, for instance a service that completes synchronously inside the action that started it, or a fault raised while the drive is being started, is queued and handled once the current transition has been committed and announced. Actions therefore always observe a consistent state, and the target state is committed before its side effects (starting the drive, completing the pending command, notifying the ready handler) run, in that order.

Every operator command that starts asynchronous work (**Calibrate**, **ReAlign**, **ReserveExternalCalibration**, **ClearCalibration**, **SetFluxLinkage**) and **Enable** are guarded by **HasPendingAsyncWork()**: they are rejected while a command is pending, while any NVM operation is in flight, including the boot-time check and load, while an identification service is running, or while a fault latched in an interrupt still owes its transition. One request therefore owns the machine at a time. A save still outstanding after an emergency stop cannot be overlapped by the save of a new run, a clear or a flux-linkage change in progress cannot be interrupted by **Enable**, whose transition would discard the completion and leave the command pending forever, and the boot-time load cannot overwrite the record of a calibration that started before it answered. The command is completed with **rejected** and the client retries once the outstanding work has completed.

A command issued from inside a completion callback is such a queued event, and it reports **CommandResult::queued**, which is distinct from **ok**: the event was accepted for processing, the table has not yet decided it, and the caller must not read it as “applied”. Callers that translate a command outcome onward — the CAN bridge turns it into **busy** — therefore never report success for work whose outcome is still unknown. A command that carries a callback is not left dangling when the table then refuses it: **CommandRejections** observes every forbidden or rejected event and completes that command’s callback with **rejected**, whether the event was dispatched directly or from the queue.

The transition table is observable: a tracer prints every transition as **fsm: <from> --<event>--> <to>** next to the existing **[SM]** lines, and every forbidden, rejected or discarded event with its state.

## Structure

**FocStateMachineCommon** is a composition root, not the place where the lifecycle logic lives. It owns the table-driven machine and hands the rows a **LifecycleContext**: references to the collaborators that carry out the work, each with one responsibility.

---

### CollaboratorResponsibility

---

**FocLifecycleTable** **Table** **constexpr** rows and the entered hooks; the only place that knows which event is legal in which state

**CalibrationFlow** Full calibration, re-alignment and external calibration: running the orchestrator, saving the record and completing the command

**MaintenanceFlow** Moving the stored calibration and changing the flux linkage

**BootSequence** The boot-time validity check and load of the stored record

**OperationFlow** Enable and disable, faults and emergency stop, and the post-commit work of every state

**PendingCommand** The one outstanding operator command, and the result held back until the target state has been committed

---

CollaboratorResponsibility

---

|                           |                                                                                                                                                                                                       |
|---------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>CommandRejected</b>    | The server that completes a queued command's callback with <b>rejected</b> when the table refuses it, so no callback is left dangling                                                                 |
| <b>NvmActivity</b>        | The count of NVM operations whose callbacks still capture the machine; part of <b>HasPendingAsyncWork()</b> . Owned by <b>ControlModeStateMachine</b> , not by <b>FocStateMachineCommon</b> — see C2a |
| <b>CalibrationContext</b> | Calibration record in RAM and its application to the controller                                                                                                                                       |
| <b>ModeHooks</b>          | The interface through which the flows reach the control mode: the controller, its tunables and the mode-specific calibration steps                                                                    |

---

**TorqueStateMachine**, **SpeedStateMachine** and **PositionStateMachine** derive from **FocStateMachineCommon** and implement **ModeHooks**; nothing else in the lifecycle depends on the concrete control mode.

## Emergency Stop

**CmdEmergencyStop** is the unconditional safety command: it is accepted from **every** state and always returns **CommandResult::ok**. Its first action is to stop the FOC controller and therefore the PWM output, before any state evaluation takes place. The stop is not left to that entry point alone: every **EmergencyStop** row stops the controller in its own action as well, so the PWM output is cut by the table itself and an **EmergencyStop** reaching the machine by any other route cannot transition out of **Enabled** with the bridge still switching. Stopping twice is harmless because the call is idempotent. Any command callback still outstanding (a running calibration or a pending **CmdClearCalibration**) is completed with **CommandResult::abortedByFault**.

The resulting state depends on the state the command was issued from:

| Issued from                          | Resulting state | Rationale                                                                          |
|--------------------------------------|-----------------|------------------------------------------------------------------------------------|
| <b>Enabled</b> or <b>Calibrating</b> | <b>Ready</b>    | Calibration data in memory and NVM is still valid; the motor stays enableable      |
| <b>Enabled</b> or <b>Calibrating</b> | <b>Idle</b>     | No valid calibration data is held, so the motor must be calibrated before enabling |
| <b>Idle</b> , <b>Ready</b>           | unchanged       | The machine is already stopped; only the inverter stop is (re)issued               |
| <b>Fault</b>                         | <b>Fault</b>    | The fault state is safe and must be cleared explicitly with <b>CmdClearFault</b>   |

---

**CmdClearFault** returns to **Ready** when valid calibration is held and to **Idle** when it is not — it does not unconditionally return to **Idle**. Clearing releases the latch, so the following **CmdEnable** is accepted rather than refused.

Calibration data is considered valid when the applied record's **stage** is **CalibrationStage::complete**, it holds a non-zero pole-pair count and a positive phase resistance, and the active mode's own requirements are met: torque mode adds no further requirement, while speed and position modes additionally require a finite, positive rotor inertia and a finite, non-negative viscous friction (**HasValidModeSpecificCalibration**, overridden by **OuterLoopStateMachine**). This mode-aware

check is why a torque-only calibration record cannot silently be reused to reach **Ready** in speed or position mode. An aborted calibration run never commits its results — the in-progress values live in `Calibrating::pendingData` and are only copied out after the NVM save succeeds — so a previously valid calibration survives an emergency stop during a re-calibration and the machine returns to **Ready**.

Dropping to **Idle** on every emergency stop would force a full recalibration after each safety intervention, which is why the transition is conditional on calibration validity rather than unconditional.

## Calibration Sequence

When `CmdCalibrate` is issued from **Idle** or **Ready**, the state machine enters **Calibrating** and executes a sequential chain of identification steps. Each step is asynchronous: the state machine calls a service and awaits a callback before proceeding to the next step. If any step returns a failure result, the machine immediately enters **Fault**.

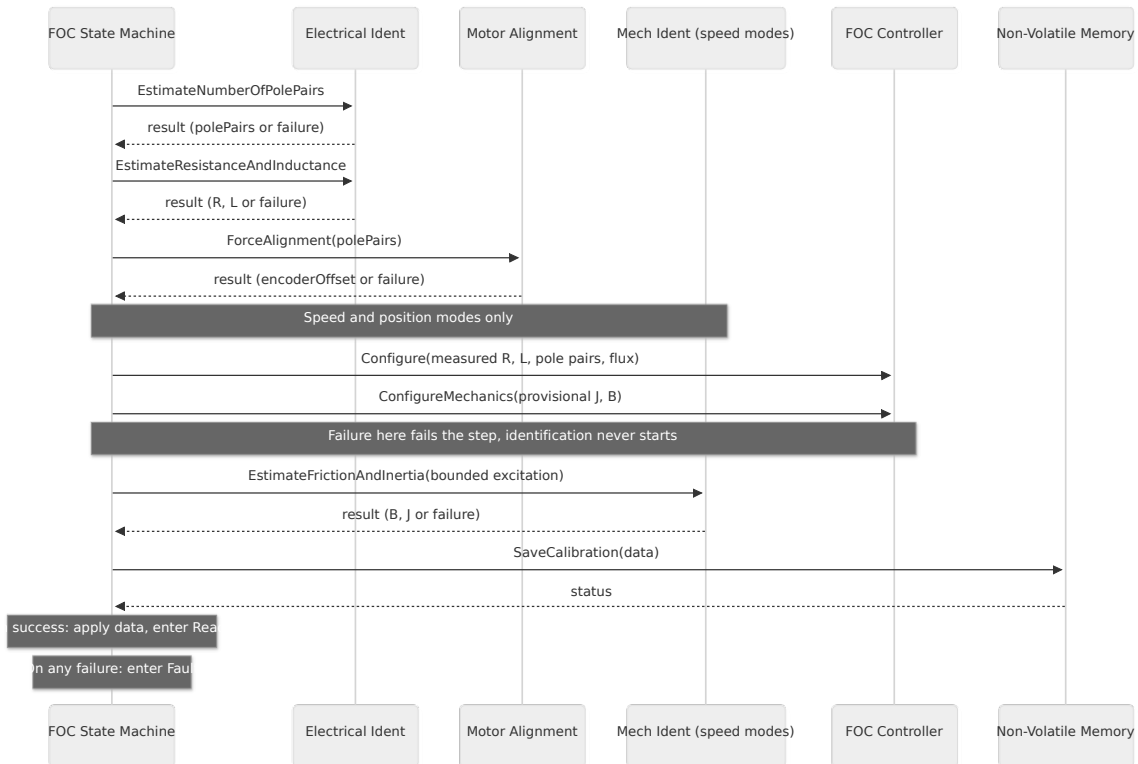


Figure 46: Sequence diagram

### Steps and data produced:

| Step                         | Service          | Data stored                                             |
|------------------------------|------------------|---------------------------------------------------------|
| 1. Pole pairs                | Electrical Ident | <code>polePairs</code>                                  |
| 2. Resistance and inductance | Electrical Ident | <code>rPhase</code> , <code>lD</code> , <code>lQ</code> |

| Step                                           | Service             | Data stored                                                                           |
|------------------------------------------------|---------------------|---------------------------------------------------------------------------------------|
| 3. Alignment                                   | Motor Alignment     | <code>encoderZeroOffset</code>                                                        |
| 4. Apply measured model + provisional plant    | FOC controller      | Nothing persisted; makes the loops able to move the rotor                             |
| 5. Mechanical parameters (speed/position only) | Mechanical Ident    | <code>inertia</code> , <code>frictionViscous</code> , <code>speedLoopBandwidth</code> |
| 6. NVM persist                                 | Non-Volatile Memory | All of the above written to EEPROM                                                    |

After saving, calibration data is applied to the FOC controller (current PID gains computed from R/L/bandwidth, velocity PID gains applied for speed modes), and the state machine transitions to **Ready**. The encoder zero offset is not written back to the encoder at this point; it is established only by the alignment step itself during the calibration sequence.

Every record handed to the NVM is traced first as `[SM] Calibration record: R=... L_mH=... p=... J_uNms2=... B_uNms=...`. Inertia and viscous friction are traced in micro-units because the tracer prints three decimals, which cannot show values around  $1e-6$  and  $1e-5$ ; the same convention applies to the `[EST]` lines and the `[SM] Applying mechanical estimates` line below.

The calibration record line is what an external observer, the software-in-the-loop suite included, compares against the plant it configured.

The torque constant every mechanical consumer uses — the identification service, the online inertia estimator and the speed-loop gain design — is not configured per target. It is derived from the record as  $K_t = \frac{3}{2}p\psi_f$  (`core/foc/math/TorqueConstant.hpp`), the factor the plant model itself applies.

A motor whose flux linkage is set with `set_flux_linkage` therefore gets a consistent torque constant without a second number to keep in step (REQ-SM-027).

## Making the loops live before mechanical identification

Mechanical identification is the only calibration step that asks the drive to *move*: it commands a speed trajectory and identifies the plant from the motion that results. Every earlier step injects current into a stationary rotor and needs no closed loop at all.

Until a model is applied, the current loop holds inert gains — it is configured from resistance, inductance and pole pairs, none of which are known before the electrical steps run — and the speed loop holds a zero current envelope and no plant, so its controller refuses to produce a reference at all. Starting identification in that state commands a speed no rotor ever reaches: the regressor carries no excitation, the run reaches its timeout, and the calibration fails on hardware while passing against any test double that publishes samples on demand.

Before starting identification the state machine therefore applies, to the live controller:

1. the **electrical model it has just measured** in this run — the pending resistance, inductance, pole-pair count and effective flux linkage, with the current-loop tunings that follow from them;

2. a **bounded provisional mechanical model** — a small placeholder inertia, zero viscous friction and a deliberately low speed-loop bandwidth. It is a stand-in that lets the speed loop act within its current envelope, not a measurement, and it is never persisted or reported as calibration.

Both are checked. If the pending electrical record is not usable, if the torque constant derived from the pending pole pairs and effective flux linkage ( $K_t = \frac{3}{2}p\psi_f$ , the same factor the plant model applies) is not positive, or if either configuration call is refused by the controller, identification is **not started**: the step fails and the machine enters **Fault**. A run that cannot move the rotor is a failed calibration, not a run that silently produces nothing.

The excitation the service is given is bounded in current, speed and time. The current envelope is the drive's own configured limit, with a small margin above it so that noise at the clamp does not read as a runaway, and the service clamps that down again to what the inverter reports it supports. The speed limit is derived from the commanded target rather than from any rated speed of the machine, because no per-drive speed rating is configured anywhere in the system; it bounds the trajectory against a runaway, not against a motor whose safe speed is below the commanded target.

### Restoring the controller after an aborted run

The provisional model must not outlive the run that needed it. Every way a calibration can end short of a committed record reaches the same restore, because they all pass through the one place that aborts a calibration run: a failed step, a refused configuration, a failed NVM save, a hardware fault, an emergency stop, and the release of the machine's external resources when it is torn down. A record that saves but is incomplete for this mode restores as well, since nothing was committed for the loops to keep.

| Situation                         | Restored state                                                                           |
|-----------------------------------|------------------------------------------------------------------------------------------|
| A valid calibration is still held | That record is re-applied: electrical model, current tunings, mechanics, estimator seeds |
| No valid calibration is held      | Nothing is re-applied; the motor stays un-enableable until a calibration succeeds        |

A committed calibration supersedes the provisional model instead of restoring over it: the record just saved is applied in full, so there is nothing left to put back.

### Validating the identified mechanics

An identified pair is accepted only when it is finite and inside the physical band the mechanical identification service and the online estimator share (see the Mechanical Parameters Identification design document). A non-finite, non-positive or out-of-band inertia or friction fails the step through the normal calibration-failure path — `CommandResult::calibrationFailed`, `Fault-Code::calibrationFailed`, `Fault` — and is never handed to the NVM service. The completeness check that stamps `CalibrationStage::complete` catches zero and non-finite values but not a merely absurd one such as an inertia of  $10^{30}$ , which is why the band is applied at the point the estimate arrives.

## External Calibration (CAN-driven)

An external client (e.g. the CAN bridge) can supply pre-measured calibration data without running the internal identification chain. This uses a two-command protocol to ensure the FSM state is correct before any inverter interaction begins:

1. **CmdReserveExternalCalibration()** — synchronous. Checks that the machine is in **Idle** or **Ready** with no pending async work, like every command that starts asynchronous work, then transitions to **Calibrating** and returns **CommandResult::ok**. Returns **CommandResult::rejected** in any other state. The **Calibrating** state prevents a second request from being accepted concurrently.
2. **CmdCompleteExternalCalibration(data, onDone)** — async. Called by the external client after its own estimation is finished. Stores **data** in the pending **Calibrating** slot and runs the alignment step, so an externally supplied record still gets a live rotor frame before it can be used. Its guard refuses the command while another command is still pending, so a second completion sent while the first is aligning is rejected instead of overwriting the pending record. If a fault occurred between the two calls, the machine is in **Fault**, where the completion command has no row and is rejected; the client observes the failure through its own estimation callback.

The two-command split ensures the FSM enters **Calibrating** before any open-loop PWM is applied, and that the state guard lives entirely inside the state machine rather than in the calling layer.

The external path stops after alignment; it does not chain into mechanical identification. **OnIdentifyElectrical** and **OnIdentifyMechanical** are separate CAN commands (REQ-INT-011), so the electrical command must not drive the mechanical estimator. The NVM save therefore stamps **stage = complete** only when the record satisfies the active mode's own validity requirements. Torque mode is satisfied by electrical parameters plus alignment and transitions to **Ready**. Speed and position still lack inertia and friction, so their record is persisted with **stage = none** and the machine returns to **Idle** holding a partial record, reported over CAN as **FocMotorState::partialCalibration**. Completing those modes requires the internal **CmdCalibrate** chain, which runs mechanical identification.

## Fault Safety

**Event-dispatcher path.** The **FaultDetected** event has a row from every state to **Fault**. Its action records the code, latches the fault controller, stops the inverter if the machine was in **Enabled** or **Calibrating**, and aborts the calibration services; only then is **Fault** committed. Any event raised while that happens, a further fault from inside the stop or a calibration completion the abort could not suppress, is queued and handled only after **Fault** has been committed, so it cannot observe the state being left. A further fault does re-enter **Fault**, but the recorded code is the one that first tripped the drive: the fault raised while shutting down is a consequence, and overwriting the root cause with it would lose the only diagnostic the operator has. The new code is recorded again once the fault has been cleared.

The clear budget is spent where the transition commits, not where the command is issued. **CmdClearFault** only reads whether a clear is still allowed; the row action is what consumes one of the three attempts and releases the latch, so a **ClearFault** that is queued and then refused does not silently cost an attempt.

**Interrupt path (board protection, CAN bus-off).** When a fault is delivered in interrupt context, the platform stops the FOC controller bridge immediately within that interrupt — before any state mutation or tracing. The `FaultDetected` event, its trace output and any pending-command completion are posted to the event dispatcher and execute on the next dispatcher turn. This ensures no multi-word state write, tracing call or non-volatile-memory access runs from an interrupt context (see REQ-SM-021).

Stopping the inverter is not on its own enough to cut the PWM output. The identification services drive the bridge through their own timers and phase-current callbacks, and one left running writes duty cycles on its next tick — `ThreePhasePwmOutput` re-arms the peripheral that `Stop()` just disabled. So the transition to `Fault` and `CmdEmergencyStop` both call `Abort()` on the electrical identification, the alignment and (in speed and position modes) the mechanical identification. `Abort()` stops injection, cancels the service’s timers, and drops the pending completion **without invoking it**: the state machine owns the outcome, and a late calibration result must not overwrite the fault that interrupted it. A service that has been aborted or has completed ignores any further phase-current sample it is handed, rather than reclaiming the inverter’s callback slot — reassigning that slot from inside its own invocation would destroy the closure being executed.

The `Runner` releases the inverter’s phase-current callback in `Disable()` for the same reason: a callback left pointing at a stopped control loop is another path back to `ThreePhasePwmOutput`.

The transition to `Enabled` commits the state **before** the FOC controller is started: the start runs as a post-commit effect of the transition, and a fault raised while it runs is queued behind it. A fault raised in the window where current first flows would otherwise observe `Ready`, skip the stop, and then be overwritten by the pending assignment to `Enabled` — leaving an energised bridge on faulted hardware with the machine reporting `Enabled`. Because the state is committed first, such a fault stops the drive.

Committing the state first is not on its own enough for a fault delivered from an interrupt, because that path dispatches `FaultDetected` on the event dispatcher, not in the interrupt: reading the lifecycle state around the start would still see `Enabled`. The latch is therefore set **in the interrupt**, and the `Enabled` entry hook consults the latch rather than the state — before `Start()`, so the cutoff the interrupt performed is not undone, and again after it, for a fault raised while `Start()` ran. Either check turns the latched fault into a `FaultDetected` dispatch, which the table queues and drains before the `Enable` dispatch returns; the whole enable sequence is therefore atomic with respect to the fault and `CmdEnable` reports `abortedByFault`.

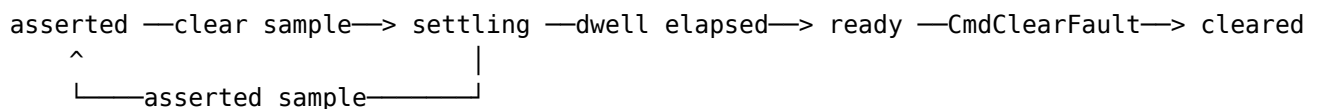
The same reasoning applies one level down, inside `Runner::Enable()`. Its steps — take the phase-current slot, enable the control law, start the inverter, set the `enabled` flag — are individually interruptible, and a `Stop()` from the faulting context between any two of them would be undone by the steps that follow it, re-arming the bridge on hardware that has just faulted, with no control loop attached to update it. `Enable()` therefore clears a `stopRequested` flag that `Disable()` sets, re-checks it after every step, and unwinds through `Disable()` once it is set. A stop landing after the last check needs no unwinding: it runs after every write the sequence makes. The flag is written from the faulting context as well as the mainline, so it is set and cleared outright rather than counted — a read-modify-write shared with an interrupt is exactly what REQ-SM-021 excludes.

## Fault Latching

A fault is latched. `CmdEnable` is refused while the latch is set, so the only path out of `Fault` is an explicit `CmdClearFault`.

Clearing is gated twice: on the hardware condition, and on a retry budget. `FaultController::EvaluateClear()` reports the first gate that refuses, condition before budget, so the operator is told the physical cause rather than the accounting.

**The condition gate.** `PlatformFactory::BoardProtectionStatus()` reports `asserted`, `clear` or `unknown`. While a fault is held, `FaultController` polls it every `conditionPollInterval` (50 ms) on the event dispatcher — on its own schedule rather than only when a command arrives, because a condition that asserts and clears between two operator commands would otherwise never be observed as asserted at all. A clear is accepted only once the condition has been seen to leave `asserted` and to have stayed `clear` for `conditionDwell` (250 ms); the dwell is stamped on the `asserted` → `clear` transition, so any asserted sample inside the window restarts it and a condition chattering around its threshold cannot be cleared on one favourable reading. `IsEnableAllowed` refuses independently while the condition reads `asserted`, so a clear granted during a momentary deassertion still cannot arm the bridge.



`unknown` is the honest answer from a platform that cannot interrogate its protection hardware, and it neither blocks a clear nor satisfies the verification — such a platform behaves exactly as it did before the gate existed. Reporting `clear` instead would assert a verification the hardware never made, and a fault on those platforms comes from software (calibration failure, CAN bus-off, watchdog) rather than from a physical condition anything could hold. **Every platform reports unknown today:** the TI protection comparators are ADC digital comparators on a PWM-triggered ADC, which stops converting once the bridge is stopped, so no reading taken while faulted distinguishes a cleared condition from an unevaluated one. The gate is in place and tested, and becomes load-bearing as soon as a platform can answer — see REQ-EH-013.

**The budget.** Each accepted `CmdClearFault` increments a counter; once `maxConsecutiveFaultClears` (3) clears have happened without an intervening clean run, further clears are refused and a reset is required. The counter is reset by `CmdDisable` from `Enabled`, which is the only evidence the state machine has that the drive ran and was stopped deliberately rather than by the same condition re-tripping. The budget supplements the condition gate rather than replacing it: it bounds the platforms that cannot verify, and it bounds a condition that happens to read clear between attempts on those that can. A clear the condition refused does not spend from it.

`FaultController::Clear()` asserts on the budget alone. It runs where the transition commits, not where the command was issued, and a condition that re-asserts in that window must refuse the clear — asserting on it there would reset a board whose hardware is still faulted, turning a fault into a reset loop.

The last fault code is preserved in `LastFaultCode()` and remains readable even after the fault is cleared via `CmdClearFault`. Before any fault has occurred it reads `FaultCode::none`, so a client can distinguish “no fault yet” from a real hardware fault.

## Faults raised in interrupt context

`PlatformFaultNotifier`'s primary path runs in the PWM fault interrupt (board protection) or the CAN interrupt (bus-off). Neither is a context in which the transition itself can be taken: the transition to `Fault` traces, writes a multi-word `std::variant` that the CLI and the CAN bridge read concurrently, and completes a pending command that reaches non-volatile memory.

The split is therefore: **latch and cut the bridge in the interrupt, record the fault in the dispatcher**. The registered handler latches the fault and calls `Stop()` on the FOC controller synchronously — unconditionally, without consulting the state, because hardware saying “fault” outranks the state machine’s belief about what it was doing — and then hands the `FaultDetected` event to `infra::EventDispatcher`. The secondary handler, which broadcasts the fault over CAN, already worked this way.

Latching is three single-word writes (`pendingCode`, `faultPending`, `faultLatched`), which is all an interrupt may do here: the tracing, the multi-word `std::variant` write and the pending-command completion stay on the dispatcher. `faultLatched` says a fault is in force and is what commands consult; `faultPending` says the `FaultDetected` transition it owes has not been taken yet, and is consumed by whichever context takes it. `FaultController` keeps a third, dispatcher-only flag, `faultRecorded`, for “the transition has been taken” — the distinction `faultLatched` carried before latching moved into the interrupt, and what decides whether an arriving fault is the one that tripped the drive or a further one whose code is kept out of `LastFaultCode()`.

`TakePendingFault()` claims the flag and the code together and hands back the code, so a second interrupt landing mid-claim cannot leave the taker dispatching one fault’s transition under another’s code. Claiming is the *only* thing that clears `faultPending`: taking the transition does not, or a fault latched while that transition ran would be wiped before its own dispatcher turn reached it and would never be recorded at all. For the same reason `CmdClearFault` is refused while a fault is pending — clearing there would drop the latch of a fault that has already tripped the drive, and spend one of the three permitted clears on a state the next dispatcher turn immediately undoes. A fault whose transition is still owed also counts as pending asynchronous work, which is what stops a control-mode switch from destroying the state machine — and with it the latch and the queued `FaultDetected` — in that window.

A consequence worth knowing when reading the code or the tests: between the interrupt and the dispatcher turn, the bridge is off and the latch is set, but `CurrentState()` still reports the pre-fault state. Commands consult the latch, not the state, so an enable arriving in that window is refused rather than acted on. The transition is taken exactly once — whichever of the dispatcher turn or the enable sequence reaches it first takes the pending fault, and the other finds nothing left to record.

## Registration lifetime

A `FaultNotifier` outlives the state machines that register with it — a control-mode switch destroys one state machine and constructs the next in the same `std::variant` — so each state machine releases its registration and aborts its calibration services in its destructor. Without that, the notifier’s `infra::Function` and the services’ pending callbacks both point into freed storage.

Hardware protection events reach the state machine through `PlatformFaultNotifier`, the production `FaultNotifier`. It registers a callback with `PlatformFactory::RegisterBoardProtection()` during construction and translates each `BoardProtectionReason` into the matching `FaultCode`:

| BoardProtectionReason | FaultCode       |
|-----------------------|-----------------|
| overCurrent           | overcurrent     |
| overVoltage           | overvoltage     |
| underVoltage          | undervoltage    |
| overTemperature       | overtemperature |

A protection event raised before the state machine has registered its handler is discarded by the state machine. A platform whose protection reflects a *standing* condition rather than an edge is expected to re-raise it once a handler exists, so a motor does not enable into a condition the board had already detected.

### Async-Callback State Invariant

A hardware fault, an operator command, or a second calibration attempt may move the state machine between the moment a service call is issued and the moment its callback fires. The invariant is: **a callback may only apply its result if the state machine is still in the state that issued the service call.**

This invariant is no longer a convention that every callback re-implements. Each callback translates its result into an event and dispatches it; whether the result is applied is decided by the transition table:

- Results from calibration steps (`EstimateNumberOfPolePairs`, `EstimateResistanceAndInductance`, `ForceAlignment`, `EstimateFrictionAndInertia`) become the events `CalibrationStepChanged`, `AlignmentSucceeded`, `MechanicalParametersIdentified` and `CalibrationStepFailed`, which only have rows in `Calibrating`. The mechanical result additionally carries a guard on the active sub-step. The calibration orchestrator also drops results of a run that was aborted.
- The `SaveCalibration` completion becomes `CalibrationSaved`, which only has rows in `Calibrating`, and only ever carries a non-`Busy` status: `CalibrationFlow` retries a `Busy` response internally (see C2c) rather than dispatching it. The callback also carries the epoch of the transition that issued the save, so a save completing after an emergency stop and a fresh `CmdCalibrate` is discarded instead of being consumed by the new run.
- The boot-time NVM completions become `BootValidityChecked` and `BootCalibrationLoaded`, which only have rows in `Idle`.
- The `InvalidateCalibration` completion becomes `CalibrationInvalidated`, with rows in `Idle` and `Ready`, leading to `Idle` on success, to `Fault` on failure, and completing the command with `rejected` when the NVM is busy. It needs no request identity: the guard on every command that starts NVM work means one invalidation at most is outstanding.
- The flux-linkage save completion becomes `FluxLinkageSaved`, handled in `Idle` and `Ready` and completing the pending command without a transition.
- Both NVM completions also have rows in `Fault`, because a fault or an emergency stop while the operation is in flight aborts the command but not the erase or the write. The RAM record follows the NVM outcome whatever happened to the command: a successful invalidation drops the record, so a later `CmdClearFault` returns to `Idle` rather than restoring a calibration that no longer exists in NVM, and a stored flux linkage is applied. An invalidation that fails while already in `Fault` is ignored; the fault is already latched.

Any of these events arriving in a state without a row for it is **forbidden**, and the save completion arriving after a transition is **discarded** before it reaches the table. Either way it is traced, and it never overwrites a later state such as **Enabled** or **Fault** with a stale result.

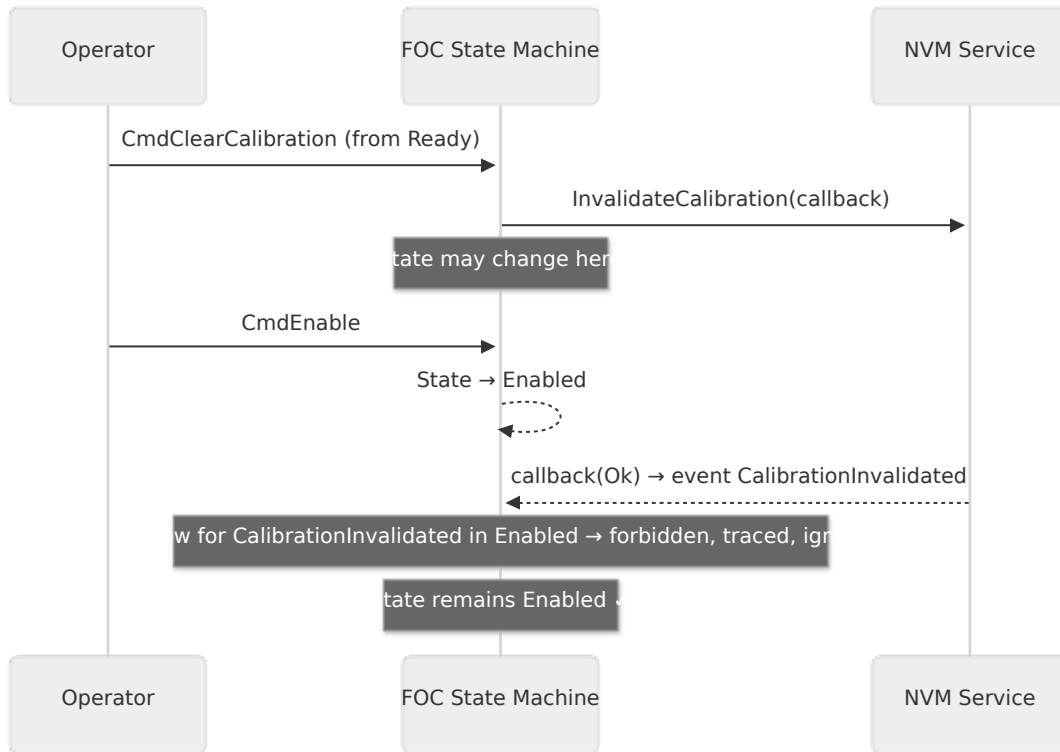


Figure 47: Sequence diagram

A matching race with a hardware fault that arrives between the `CmdClearCalibration` call and the `InvalidateCalibration` callback:

## Transition Policies

Each state machine instance is constructed with a transition policy that only decides whether the lifecycle commands appear on the terminal:

- **CLI policy:** the commands `calibrate`, `align`, `enable`, `disable`, `clear_fault` and `clear_cal` are registered on the connected terminal. Used by the state machine unit tests and for commissioning over a serial console.
- **Automatic policy:** no terminal commands are registered. The caller drives transitions programmatically, for example from CAN message handlers. This is the policy the `ControlModeStateMachine` uses for the torque, speed and position machines it owns.

Both policies share the same transition table; they differ only in whether lifecycle commands appear on the terminal.

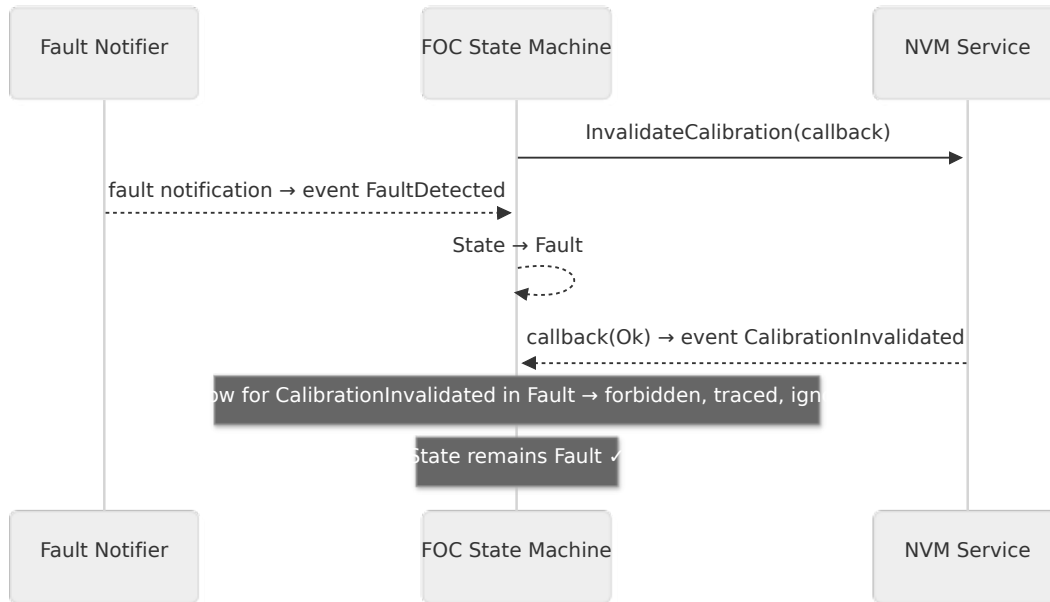


Figure 48: Sequence diagram

## Mechanical Identification and Control Mode

The calibration sequence extends with a mechanical identification step for speed and position control modes. This step estimates rotor inertia and viscous friction, then computes initial velocity-loop PID gains.

For **both speed and position mode**, the mechanical identification service is constructed by `OuterLoopStateMachine::ResolveMechIdent` when no external override is supplied in `CalibrationServices`. The two modes share that path; position mode does not require an override and does not fail calibration for the lack of one. An override is a test and integration seam, not a configuration requirement.

## Boot-Time NVM Check

On construction, the state machine asynchronously checks whether valid calibration data exists in NVM. If data is found and loads successfully, the electrical parameters (R, L, pole pairs, current-loop bandwidth) are applied to the FOC controller, but the machine **remains in Idle** — the rotor-reference (encoder zero offset) is RAM-only and is lost on every reset, so it must be re-established by a live alignment before the motor can be enabled. If the check fails or the data is absent, the machine starts in **Idle** with no parameters applied.

## Alignment-Only Recovery

`CmdReAlign()` re-establishes the rotor reference without re-running the full identification chain. It requires the machine to be in **Idle** or **Ready** with valid electrical calibration already loaded (non-zero pole pairs and positive phase resistance). The procedure:

1. Enters **Calibrating** at the alignment sub-step, copying the stored electrical parameters into

- pendingData.
2. Calls the Motor Alignment service using the stored pole pairs.
  3. On success: marks `rotorReferenceValid`, saves the updated offset to NVM through the same completion path as a full calibration, applies mode-specific calibration from the stored parameters, and transitions to `Ready`.
  4. On failure: enters `Fault` with code `calibrationFailed`.

The CLI command is `align` (short form `aln`). Because the same completion path is used, the speed and position loop parameters are also re-applied from the stored calibration, so mechanical identification does not need to be re-run after an alignment-only recovery.

## Online Parameter Estimation (Speed/Position Modes)

For speed and position control modes, the state machine creates and manages two online parameter estimators alongside the main control loop:

- **Online Mechanical Estimator** — continuously refines rotor inertia ( $J$ ) and viscous friction ( $B$ ) using a Recursive Least Squares estimator that runs at the outer-loop rate (1 kHz).
- **Online Electrical Estimator** — continuously refines phase resistance ( $R$ ) and d-axis inductance ( $L_d$ ) using a d-axis voltage-model RLS estimator running at the same rate.

**Lifecycle of online estimators:**

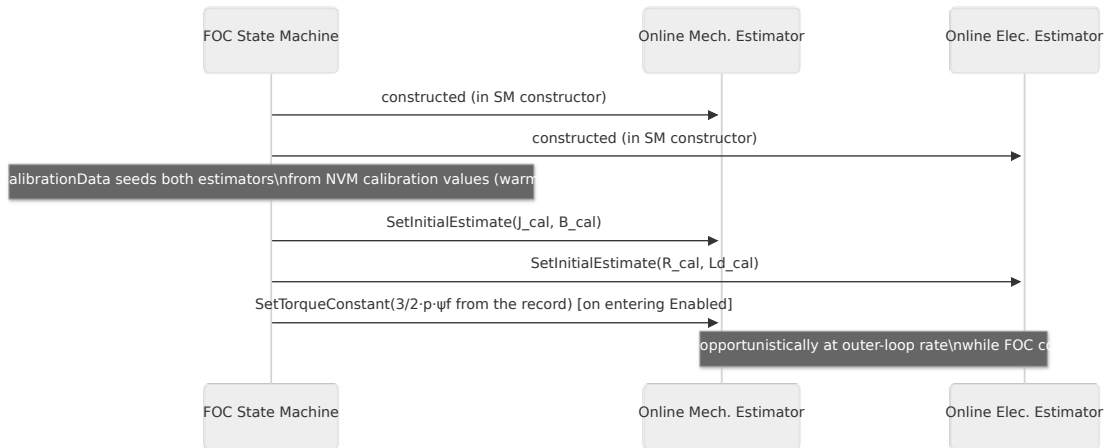


Figure 49: Sequence diagram

**Seeding from calibration data:** When `ApplyCalibrationData` is called (either after calibration completes or on NVM boot load), both estimators are seeded with the values from `CalibrationData`. This warm-starts the RLS theta vector at the known-good calibration values rather than zero, ensuring the estimators produce physically meaningful outputs from the first update.

The electrical estimator is seeded using `Ld` (d-axis inductance), as the underlying model assumes a non-salient motor ( $L_d \approx L_q$ ). For interior PMSMs, a 3-parameter model with separate  $L_d$  and  $L_q$  would be required; this is a known limitation.

**Estimate consumption is explicit:** Estimates are NOT applied to PID gains continuously. The operator (or application logic) explicitly triggers a PID retune by calling `ApplyOnlineEstimates()`.

This prevents gain oscillation before the estimators have converged and guards against applying poorly-conditioned estimates at runtime.

When `ApplyOnlineEstimates()` is called while in `Enabled` state: 1. Current inertia and friction estimates are read from the mechanical estimator 2. Speed PID gains are recomputed using the bandwidth-based rule from `documentation/theory/speed-loop-pi.md` (“What the implementation uses”):  $k_p = 2J\omega_{bw}/K_t$ ,  $k_i = k_p \max(B_f/J, \omega_{bw}/4) T_s$  with  $T_s$  the outer-loop sample period, where  $J$  is the estimated inertia,  $B_f$  the viscous friction, and  $K_t = \frac{3}{2}p\psi_f$  the torque constant derived from the calibration record 3. Current resistance and inductance estimates are read from the electrical estimator 4. Current PID gains are recomputed from the bandwidth-based tuning rule

If called from any state other than `Enabled`, the call is silently ignored.

#### Runtime control commands (CLI policy, speed/position modes only):

| Command                      | Short           | Description                                           |
|------------------------------|-----------------|-------------------------------------------------------|
| <code>apply_estimates</code> | <code>ae</code> | Apply online estimates to speed and current PID gains |
| <code>estimate_status</code> | <code>es</code> | Print current J, B (micro-units), R, Ld to the tracer |

### Control Mode Selection (`ControlModeStateMachine`)

`ControlModeStateMachine` is a higher-level coordinator that owns one `FocStateMachineCommon` subclass per supported control mode (`TorqueStateMachine`, `SpeedStateMachine`, `PositionStateMachine`), held in a `std::variant`. At any given time exactly one mode’s state machine is the **active** instance; the others are idle. This section documents three behavioral invariants introduced to make runtime mode switching safe and predictable.

#### C1 — CAN Wire-Scale Convention

`ControlModeStateMachine` receives setpoints from the CAN bridge in physical unit types (`foc::Ampere`, `foc::RadiansPerSecond`, `foc::Radians`). The fixed-point wire encoding (scale constants in `core/can/FocMotorMessages.hpp`, round-to-nearest with saturation) is specified once, in `documentation/design/service-can.md` Part A.

#### C2 — Re-entrancy Guard for In-Flight Selection

Switching control mode involves an asynchronous NVM write to persist the new default. A second `Select()` call issued while this write is still outstanding must not be applied until the first operation completes, since the NVM driver does not support concurrent write requests.

Behavioral rule: if a `Select()` is called while a previous `Select()` callback has not yet fired, the call returns `SelectResult::busy` immediately and takes no other action. The caller is responsible for retrying.

The same guard covers the active mode’s own asynchronous work. Applying a selection destroys the active `FocStateMachineCommon` instance, while its outstanding NVM and identification callbacks still capture that instance. `Select()` therefore also returns `SelectResult::busy` when `ActiveStateMachine().HasPendingAsyncWork()` is true — that is, while a command callback is pending, any NVM operation is in flight, a calibration step is running, or a fault latched in an interrupt still owes its transition — in addition to the existing check that the active machine is stopped.

Without that last condition the machine, its latch and the queued transition would all be destroyed in the window between the interrupt and the dispatcher turn, and the machine built in its place would start unlatched. `NvmActivity` counts every NVM operation from the call until its callback, so a save or invalidation still outstanding after an emergency stop keeps the machine alive until it completes.

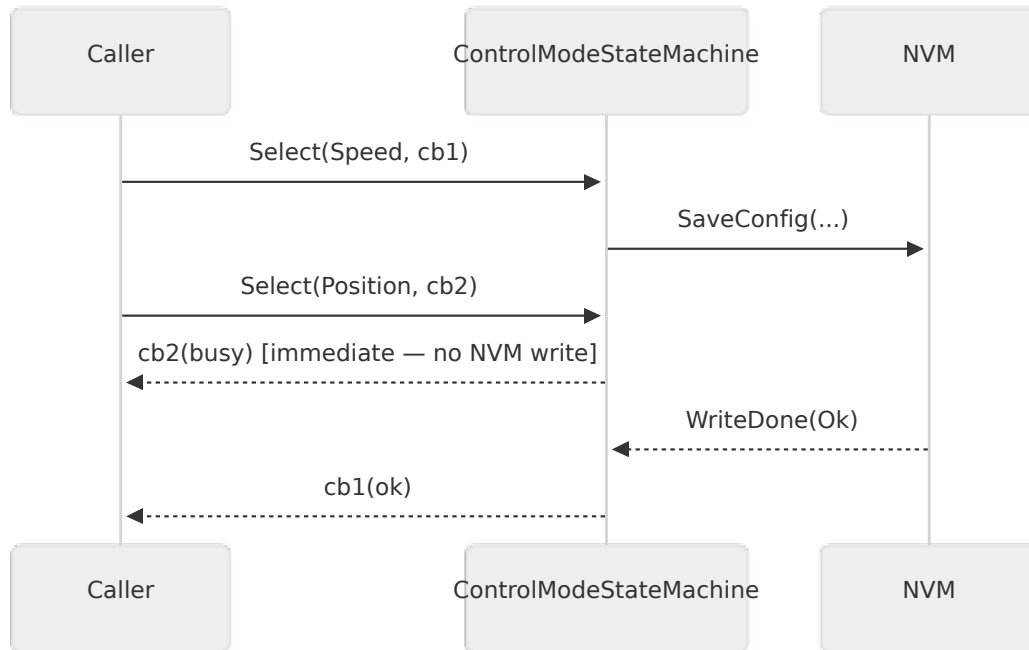


Figure 50: Sequence diagram

### C2a — One `NvmActivity` for Every Writer of the Shared NVM

`NonVolatileMemoryImpl` serves one operation at a time and answers a concurrent call with `Busy`, without queueing. `NvmActivity` is therefore owned by `ControlModeStateMachine`, which outlives every mode switch, and every writer of the shared instance brackets its operation with it:

| Writer                                                                                                                              | Accounting                                                                                                                                                      |
|-------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Calibration, maintenance and boot<br>( <code>CalibrationFlow</code> , <code>MaintenanceFlow</code> ,<br><code>BootSequence</code> ) | <code>LifecycleEnvironment::nvmActivity</code> , a<br>reference to the shared counter                                                                           |
| Select's <code>SaveConfig</code> and its rollback write<br><code>AlgorithmPersistence::PersistConfig</code>                         | <code>Begin/End</code> around each write<br><code>Begin</code> before the first attempt, <code>End</code> once a<br>status other than <code>Busy</code> arrives |
| <code>FocMotorCanBridge</code> encoder-resolution and<br>telemetry-rate writes                                                      | <code>ControlModeStateMachine::BeginNvmActivity/EndNvmActi</code>                                                                                               |

`HasPendingAsyncWork()` is therefore true while any of them is outstanding, and every guard built on it — `CmdCalibrate`, `CmdEnable`, `CmdClearCalibration`, `Select` — refuses instead of colliding.

### C2b — Algorithm Selection Is Refused While Busy

`SelectCurrentAlgorithm`, `SelectSpeedAlgorithm` and `SelectPositionAlgorithm` apply the same guard as `Select` — the active machine is **Idle** or **Ready** and has no pending asynchronous work — and return **busy** otherwise. Selection is therefore refused while **Calibrating**, **Enabled** or **Fault**, and while any NVM operation is outstanding. The cascades' own refusal while enabled remains as a second guard.

### C2c — A Busy Calibration Save Is Retried

`CalibrationFlow` retries a `SaveCalibration` answered with **Busy** every 10 ms from a copy of the record, keeping `NvmActivity` raised and reusing the completion captured at the epoch of the run, so `CalibrationSaved` is only dispatched with the status of a write that actually ran. An abort while a retry is waiting cancels it and releases `NvmActivity`: an aborted run is not persisted.

### C3 — Deferred Apply and Re-check Before Activate

`Select()` does not switch the active state machine up front. It records the previous `defaultControlMode`, writes the new one into `configData` and starts the asynchronous `SaveConfig`; `activeSm` is untouched until `OnSaveConfigDone` runs.

`Select()` starts the write only when the active machine is stopped and has no pending asynchronous work, but that is a snapshot: a fault, an emergency stop or a calibration can land while the write is outstanding. Activating the new mode then would destroy the machine that holds the fault and start the new one unlatched. `OnSaveConfigDone` therefore decides:

1. **Write failed:** restore `configData.defaultControlMode` (NVM still holds the previous default) and report **busy** for a transient failure or **nvmFailed** otherwise. The active mode was never touched.
2. **Write succeeded and the guard still holds:** `Activate(pendingSelectMode)` and report ok.
3. **Write succeeded but the guard no longer holds:** keep the active mode, restore `configData.defaultControlMode` and write it back so NVM agrees with the running mode, then report **busy** when that write completes, whatever its status. The pending callback stays set during the rollback write, so a `Select()` in that window is refused with **busy** (C2).

## Interfaces

### Provided

| Interface Purpose                                                                                   | Contract                                                                                                                                                                                 |
|-----------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>FocStateMachineBase</code><br>Machine lifecycle controller — state query and command dispatch | Constructed once per application; all command methods are safe to call from any state (a command without a row in the current state is rejected with a status and traced, never applied) |

| Interface Purpose                                                                                                       | Contract                                                                                                                                                                                                                                                                                                      |
|-------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>CurrentState()</b> Returns the current <b>State</b> variant for inspection                                           | Returns a const reference; valid for the lifetime of the state machine                                                                                                                                                                                                                                        |
| <b>LastFaultCode()</b> Returns the most recent fault code                                                               | Returns <b>FaultCode::none</b> until the first fault occurs; afterwards it retains the last fault code, also after the fault is cleared                                                                                                                                                                       |
| <b>CmdCalibrate()</b> Starts start of full calibration sequence                                                         | Only effective from <b>Idle</b> or <b>Ready</b> ; ignored from all other states                                                                                                                                                                                                                               |
| <b>CmdReAlign()</b> Re-establishes the rotor reference without re-running R/L or mechanical identification              | Only effective from <b>Idle</b> or <b>Ready</b> with valid electrical calibration loaded; enters <b>Calibrating</b> (alignment sub-step only), then <b>Ready</b> with <b>rotorReferenceValid</b> on success, or <b>Fault</b> on failure                                                                       |
| <b>CmdEnableFoc()</b> Requests enabling the FOC controller                                                              | Only effective from <b>Ready</b> when <b>rotorReferenceValid</b> is true; rejected when the rotor reference has not been established since the last reset                                                                                                                                                     |
| <b>CmdDisableFoc()</b> Requests disabling the FOC controller                                                            | Only effective from <b>Enabled</b> ; ignored from all other states                                                                                                                                                                                                                                            |
| <b>CmdClearFault()</b> Clears the fault and returns to <b>Ready</b> if valid calibration is held, otherwise <b>Idle</b> | Only effective from <b>Fault</b> ; ignored from all other states                                                                                                                                                                                                                                              |
| <b>CmdClearCalibrationNvm()</b> Clears calibration and returns to <b>Idle</b>                                           | Only effective from <b>Idle</b> or <b>Ready</b> ; ignored from <b>Calibrating</b> , <b>Enabled</b> , and <b>Fault</b> . On NVM failure transitions to <b>Fault</b> .                                                                                                                                          |
| <b>CmdEmergencyStopNvm()</b> Immediately and leaves the active states                                                   | Accepted from every state and always returns <b>ok</b> . From <b>Enabled</b> or <b>Calibrating</b> it goes to <b>Ready</b> when calibration data is valid, otherwise to <b>Idle</b> . <b>Idle</b> , <b>Ready</b> and <b>Fault</b> are left unchanged. Aborts any pending command with <b>abortedByFault</b> . |
| <b>HasPendingAsyncWork()</b> Returns whether a service callback capturing the machine is outstanding                    | True while a command callback is pending, any NVM operation is in flight, a calibration step is running, or a fault latched in an interrupt still owes its transition. Used by <b>ControlModeStateMachine</b> to refuse destroying the machine.                                                               |
| <b>HasPartialCalibration()</b> Returns true if <b>Idle</b> holds a non-empty but incomplete NVM record                  | True only in <b>Idle</b> , when <b>stage != complete</b> but pole pairs or resistance are non-zero (an old-schema or interrupted record). Used by the CAN bridge to broadcast <b>FocMotorState::partialCalibration</b> instead of <b>idle</b> .                                                               |
| <b>ApplyOnlineEstimates()</b> Applies estimated (and current PID gains from online estimators                           | Only effective from <b>Enabled</b> ; silently ignored from all other states. Skips non-physical estimates (non-finite or $\leq 0$ ). Speed/position modes only.                                                                                                                                               |

## Required

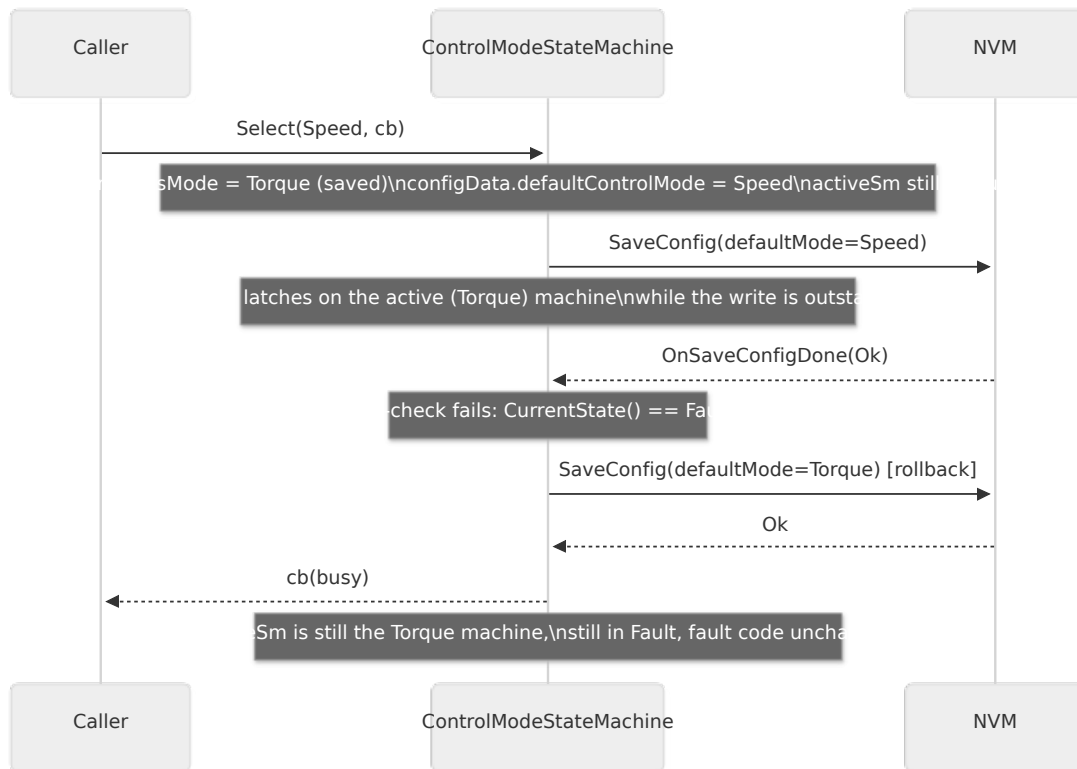


Figure 51: Sequence diagram

| Interface                                       | Purpose                                                                                     | Contract                                                                                                                                                                       |
|-------------------------------------------------|---------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>NonVolatileMemory</b>                        | Persists and retrieves calibration data across power cycles                                 | Must remain valid for the lifetime of the state machine                                                                                                                        |
| <b>ElectricalParameters</b>                     | Electrical identification: phase resistance, and dq inductances                             | Operations are asynchronous; callback fires on the same event loop                                                                                                             |
| <b>MotorAlignment</b>                           | Forces rotor to a known angle and returns the encoder zero offset                           | Operation is asynchronous; result is optional (nullopt = failure)                                                                                                              |
| <b>MechanicalParameters</b>                     | Electrical identification and viscous friction (speed/position modes only)                  | Operation is asynchronous; result is optional (nullopt = failure)                                                                                                              |
| <b>FaultNotifier</b>                            | Delivers hardware fault notifications to the state machine                                  | <b>Register()</b> must be called during construction; callback may fire at any time. Production implementation is <b>PlatformFaultNotifier</b> .                               |
| <b>ThreePhaseInverter</b>                       | Used by the FOC controller to issue PWM and read phase currents                             | Stopped immediately on any fault from <b>Enabled</b> or <b>Calibrating</b> state                                                                                               |
| <b>Encoder</b>                                  | Rotor position sensor; zero point established by the alignment step                         | Read-only from the state machine's perspective; <b>SetZero()</b> is called by <b>MotorAlignment</b> during calibration, never by the state machine itself                      |
| <b>TerminalWithStorage</b>                      | Serial command interface for CLI-mode transition policy                                     | Commands registered in constructor; terminal must outlive the state machine                                                                                                    |
| <b>Tracer</b>                                   | Debug trace output for lifecycle events                                                     | All state transitions and calibration steps are traced                                                                                                                         |
| <b>RealTimeFrictionAndInertiaEstimator</b>      | Online parameter estimator for rotor inertia and viscous friction (speed/position only)     | Seeded from calibration data; torque constant $\frac{3}{2} \cdot p \cdot \psi / f$ from the record set on entering <b>Enabled</b> ; updates run while FOC outer loop is active |
| <b>RealTimeResistanceAndInductanceEstimator</b> | Online parameter estimator for phase resistance and d-axis inductance (speed/position only) | Assumes non-salient motor ( $L_d \approx L_q$ ); seeded using <b>LD</b> from calibration                                                                                       |

## Data Model

| Entity          | Field           | Type / Unit   | Range | Notes                                           |
|-----------------|-----------------|---------------|-------|-------------------------------------------------|
| CalibrationData | PolePairs       | count (uint8) | 1–255 | Number of electrical pole pairs                 |
| CalibrationData | PhaseResistance | Ohm (float)   | >0    | Phase resistance identified by electrical ident |

| Entity          | Field                | Type /<br>Unit                                     | Range    | Notes                                                                                                                                                                         |
|-----------------|----------------------|----------------------------------------------------|----------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| CalibrationData | D/Q                  | mH (float)                                         | > 0      | D/Q inductances (set equal; anisotropy not estimated)                                                                                                                         |
| CalibrationData | ZeroOffset           | (bit-cast float Radians)                           | any      | Mechanical angle at encoder zero when rotor settled during alignment; stored for reference only — not re-applied to the encoder at boot or after calibration (see REQ-SM-019) |
| CalibrationData | Inertia              | $\text{N} \cdot \text{m} \cdot \text{s}^2$ (float) | $\geq 0$ | Rotor inertia; populated only for speed/position modes                                                                                                                        |
| CalibrationData | Viscous              | $\text{N} \cdot \text{s} / \text{rad}$ (float)     | $\geq 0$ | Viscous friction coefficient; populated only for speed/position modes                                                                                                         |
| CalibrationData | Coulomb              | $\text{N}$ (float)                                 | $\geq 0$ | Coulomb friction; currently 0 (not identified)                                                                                                                                |
| CalibrationData | SpeedLoopBandwidth   | (float)                                            | $\geq 0$ | Speed loop closed-loop bandwidth; populated only for speed/position modes                                                                                                     |
| CalibrationData | CurrentLoopBandwidth | (float)                                            | $\geq 0$ | Current loop closed-loop bandwidth; defaults to $2\pi \cdot f_s / \text{nyquistFactor}$ when zero                                                                             |
| FaultCode       |                      | enum (uint8)                                       | 8 values | overcurrent, overvoltage, undervoltage, overtemperature, encoderLoss, watchdogTimeout, hardwareFault, calibrationFailed                                                       |

`watchdogTimeout` is reserved and not raised by the state machine. A watchdog expiry cuts the power stage from the watchdog interrupt and the hardware resets the target, so there is no dispatcher turn in which a latched fault code could be read or broadcast; the expiry is reported after reboot through `ResetCause::watchdog` instead. The watchdog does not reach the state machine at all. See Watchdog Design (Chapter 35).

## Error Handling

All calibration steps produce `std::optional` results. A `nullopt` from any step immediately enters `Fault` with code `calibrationFailed`. The entire calibration sequence is safe to retry by issuing `CmdCalibrate` after `CmdClearFault`.

NVM operations (save, load, invalidate) are also asynchronous and report a `NvmStatus`. On any non-Ok status during save or load, the machine enters `Fault` or remains in `Idle`, respectively, without corrupting application state.

| Field     | Value                    |
|-----------|--------------------------|
| Title     | Service: Motor Alignment |
| Type      | design                   |
| Status    | draft                    |
| Version   | 0.1.0                    |
| Component | service-alignment        |

---

| Field | Value      |
|-------|------------|
| Date  | 2026-04-07 |

---

**IMPORTANT — Implementation-blind document:** This document describes *behavior, structure, and responsibilities* WITHOUT referencing code. **No code blocks using programming languages (C++, C, Python, CMake, shell, etc.) are allowed.** Use Mermaid diagrams to express behavior instead. Prose descriptions of algorithms are encouraged; source-level details are not.

**Diagrams:** All visuals must be either a Mermaid fenced code block (````mermaid`) or ASCII art inline in the document. External image references using Markdown image syntax are **not allowed**.

---

## Responsibilities

**Is responsible for:** - Calibrating the encoder zero-offset so that the encoder frame and the motor's electrical rotor frame are aligned (position = 0 in the encoder corresponds to electrical angle = 0° in the rotor) - Applying an open-loop DC voltage vector at electrical angle 0° via the three-phase inverter to force the rotor to a known magnetic position - Detecting rotor settlement by monitoring encoder position change over consecutive samples - Reporting the calibrated encoder offset (in radians) via a single completion callback on success, or reporting failure if the rotor does not settle within the allowed time - Stopping the inverter cleanly after the procedure completes, whether successfully or not - Enforcing mutual exclusion: silently ignoring re-entry while an alignment is already in progress

**Is NOT responsible for:** - Persisting the returned encoder offset to NVM — that is the caller's decision - Running during normal FOC operation — the FOC loop must be stopped by the caller before alignment begins - Measuring or interpreting motor electrical parameters (R, L, pole pairs) - Providing any feedback during the alignment beyond the single completion callback

---

## Component Details

### Preconditions and Configuration

Before invoking alignment, the caller must have:

1. Stopped the normal FOC loop so the three-phase inverter is not claimed by a current control loop.
2. Provided the number of pole pairs for the motor under test.
3. Constructed an `AlignmentConfig` specifying:
  - **test voltage percent** — fraction of DC bus voltage (0–100 %) applied to the d-axis during the procedure
  - **sampling frequency** — the rate at which encoder samples are taken and settlement is checked (Hz)
  - **max samples** — the total number of sampling ticks before the procedure times out

- **settled threshold** — the maximum absolute encoder position change (radians) that qualifies as “not moving”
- **settled count** — the number of consecutive below-threshold samples required to declare settlement

These parameters are tuned by the application engineer for the specific motor and mechanical load. A typical configuration applies 10–20 % of bus voltage and requires 20 consecutive samples within 0.001 rad of one another before declaring settlement.

## Open-Loop Voltage Application

The alignment service drives the inverter directly in open-loop mode, bypassing all current feedback. It synthesises a stationary voltage vector by computing the inverse Clarke and inverse Park transforms with:

- d-axis voltage = `test_voltage_percent / 100` (normalised to DC bus)
- q-axis voltage = 0 (no torque component)
- electrical angle = 0°

The resulting three-phase duty cycles are submitted to the inverter each sampling tick for the duration of the procedure. Because the q-axis component is zero, the resulting stator field is purely aligned with the d-axis at 0°. A surface PMSM or BLDC rotor will rotate to align its permanent-magnet d-axis with this field and remain there.



Figure 52: Block diagram

## Settlement Detection

Settlement detection is a simple sliding window state machine. At each sampling tick (controlled by the inverter’s ADC interrupt callback):

1. The encoder position is read.
2. The absolute difference between the current and previous position sample is computed.
3. If this difference is below `settledThreshold`, the consecutive-settlement counter is incremented; otherwise the counter is reset to zero.
4. If the counter reaches `settledCount`, the rotor is declared settled.
5. If the total sample count reaches `maxSamples` without settlement, a timeout is declared.
6. If any phase current in a sample exceeds `ThreePhaseInverter::MaxCurrentSupported()`, the injection is aborted immediately and the timeout path is taken. Alignment drives an open-loop voltage with no current regulation, so this per-sample check is the only software limit on the resulting current.

Sampling is gated on the pending completion rather than on the inverter’s callback slot being reclaimed: reassigning that slot from inside its own invocation would destroy the closure being executed, and the over-current path fails from inside it. A sample arriving after the procedure ended is discarded before it reaches anything.

**Abort()** is the caller-facing form of the same shutdown. It stops the alignment voltage and drops the pending completion **without invoking it**, so a fault raised by the state machine cannot be overwritten by an alignment result that arrives after it.

Every decision above is taken in the sampling interrupt, and so is everything that must happen at that instant: the inverter is stopped there and, on success, the encoder zeroed there. The completion is not. The caller stores calibration data and changes state from it, which is event-loop work and far longer than a control period, so the outcome is queued and delivered from the event loop. Until then the run is still in progress: samples are discarded, a second **ForceAlignment** is rejected, and **Abort()** drops the queued outcome. The outcome carries the run it belongs to, so one that an abort overtook cannot complete a run started after it. The step timeout already runs on the event loop and completes the run directly, unless an outcome is already queued, which it then leaves to be delivered.

The encoder position at the moment of settlement declaration becomes the calibration offset. This offset represents the mechanical angle the rotor adopts when the electrical d-axis is at 0°; subtracting it from any subsequent encoder reading yields the corrected angle for the closed-loop FOC controller.

The offset is in **mechanical** radians, which is the unit **Encoder::Set** and **Encoder::SetZero** consume. It is not scaled by pole pairs; scaling it would make it an electrical angle and misorient the field.

The service applies the offset itself, by calling **Encoder::SetZero()** at the settled instant while the rotor is still held at electrical zero. It does not wait for the caller to re-apply the reported value, because the rotor drifts as soon as the bridge is released. The reported value is therefore a record of where the encoder read before it was zeroed, and the calibration record persisted to NVM is diagnostic: an incremental encoder loses its reference across a reset, so the state machine does not re-apply a stored alignment at boot and calibration must be re-run to orient the field.

## State Machine

The service follows a four-state machine:

While in the **Aligning** state, any further call to **ForceAlignment** is silently ignored — the service will not restart or nest. Only one alignment may be in progress at any time. Transitions out of the **Aligning** state always stop the inverter before invoking the callback, so the caller's callback receives control only after the hardware is in a safe, stopped state.

## Completion Callback Semantics

The **onDone** callback accepts a single **std::optional<Radians>** argument:

- **std::optional** with a value: alignment succeeded; the contained value is the encoder offset in mechanical radians, already applied to the encoder by the service.
- Empty **std::optional**: alignment failed (timeout); the caller should not use any offset and may retry.

The callback fires exactly once per **ForceAlignment** invocation, from the event loop and never from the sampling interrupt. The callback is reset (using **infra::AutoResetFunction** semantics) after firing so that stale references cannot cause a second invocation.

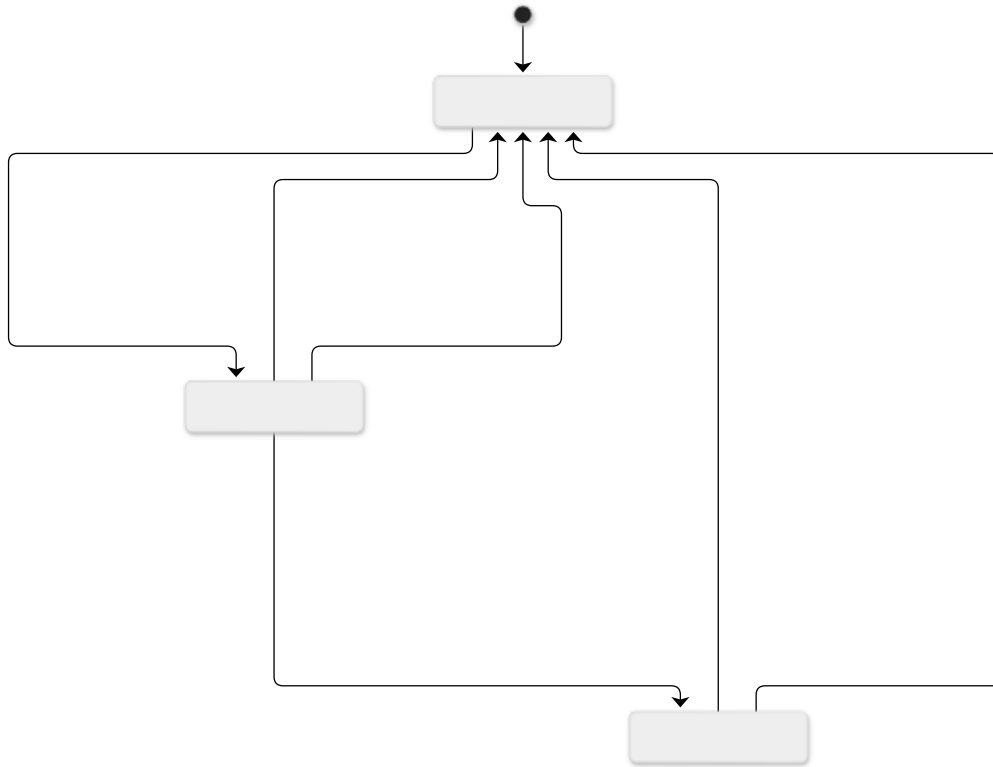


Figure 53: State diagram

## Interfaces

### Provided

| Interface                                                | Purpose                                                                                                                                         | Contract                                                                                                                                        |
|----------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>ForceAlignment</b> (Source, Alignment-Config, onDone) | Starts alignment procedure using the supplied configuration; reports the calibrated encoder offset (or failure) via <b>onDone</b> when complete | Silently ignored if already aligning; <b>onDone</b> fires exactly once, from the event loop; inverter is stopped before <b>onDone</b> is called |

### Required

| Interface                 | Purpose                                                                                                                          | Contract                                                                                |
|---------------------------|----------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------|
| <b>ThreePhaseInverter</b> | Provides voltage application during alignment; source of ADC sampling callbacks that drive the settlement state machine          | Must not be concurrently claimed by any other controller during the alignment procedure |
| <b>Encoder</b>            | Reads the current rotor mechanical angle at each sampling tick so that settlement can be detected and the offset can be captured | Must be initialised and tracking position before <b>ForceAlignment</b> is called        |

---

| Field     | Value                                         |
|-----------|-----------------------------------------------|
| Title     | Service: Electrical Parameters Identification |
| Type      | design                                        |
| Status    | draft                                         |
| Version   | 0.2.0                                         |
| Component | service-electrical-ident                      |
| Date      | 2026-08-30                                    |

---

**IMPORTANT — Implementation-blind document:** This document describes *behavior, structure, and responsibilities* WITHOUT referencing code. **No code blocks using programming languages (C++, C, Python, CMake, shell, etc.) are allowed.** Use Mermaid diagrams to express behavior instead. Prose descriptions of algorithms are encouraged; source-level details are not.

**Diagrams:** All visuals must be either a Mermaid fenced code block (````mermaid`) or ASCII art inline in the document. External image references using Markdown image syntax are **not allowed**.

---

## Responsibilities

**Is responsible for:** - Measuring phase resistance ( $R_s$ ) via DC voltage step using `ResistanceEstimator` - Measuring d-axis inductance ( $L_s$ ) via HF sinusoidal injection using `SinusoidalInductanceEstimator` - Estimating the number of pole pairs by rotating an open-loop voltage vector and comparing electrical angle to mechanical encoder angle - Delivering a combined `ResistanceInductanceResult` — resistance, inductance, and fit quality — via a single completion callback once both sub-estimators finish - Enforcing that R/L identification and pole-pairs estimation cannot run concurrently - Stopping the inverter cleanly before invoking any completion callback - Delivering every completion from the event loop, never from the sampling interrupt that decides it - Rejecting calibration when `fitQuality < 0.5` (incoherent injection response)

**Is NOT responsible for:** - Persisting identified parameters — the caller decides what to do with results - Encoder zero-offset calibration — that is the Motor Alignment service - Performing any closed-loop current control — all voltage application is open-loop - Running concurrently with the normal FOC loop — the FOC loop must be stopped before either procedure begins

---

## Architecture

The R/L estimation is split across three cooperating components:

`ResistanceEstimator` and `SinusoidalInductanceEstimator` are standalone components; the orchestrator calls them sequentially and assembles their results into a single `ResistanceInductanceResult`. Application code (hardware test terminal, etc.) may also call these components directly.

---

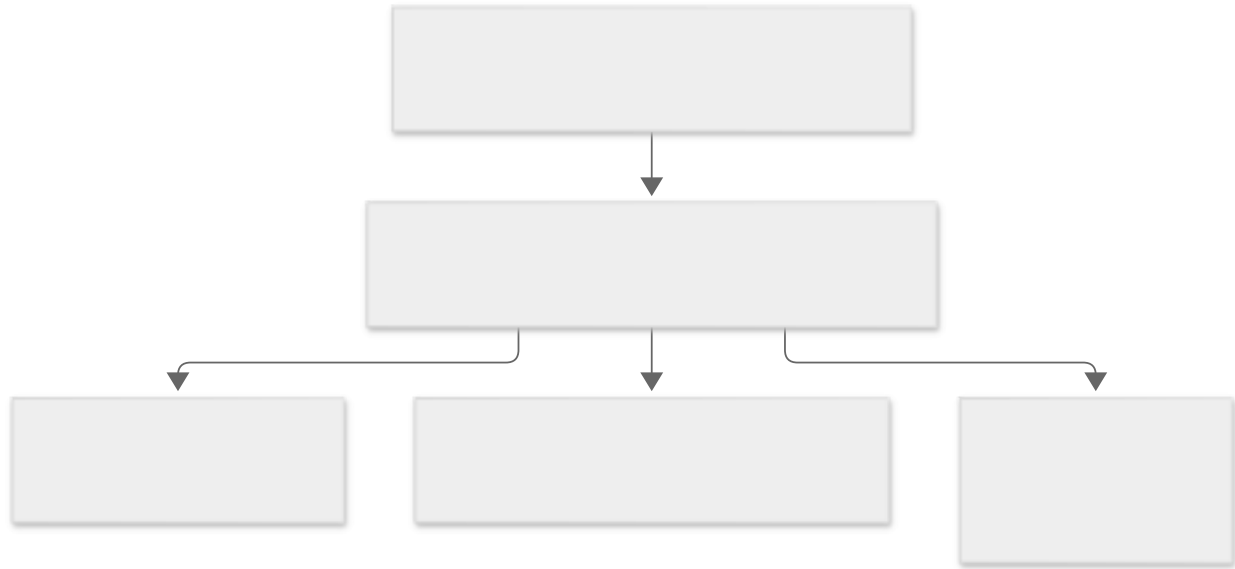


Figure 54: Block diagram

## Component Details

### Open-Loop Injection Current Limit

Every procedure in this service drives the bridge open-loop: a fixed duty is applied and held, with no current regulator in the path. The resulting current is set entirely by the applied voltage and the winding impedance, and a low-resistance motor reaches a large current within the settle time. The resistance test in particular holds a DC duty for seconds.

Each procedure therefore registers a phase-current callback that checks every sample against `ThreePhaseInverter::MaxCurrentSupported()` and aborts immediately if any phase exceeds it — stopping the inverter, cancelling the step timer, and reporting failure through the procedure’s normal failure path. Previously the pole-pairs and resistance procedures registered an empty callback and discarded the samples, leaving hardware comparators (present on some boards, absent on others) as the only backstop. `MotorAlignment` applies the same check.

The limit is a per-sample abort, not regulation: it stops a runaway, it does not hold a setpoint.

### Ending a procedure

A procedure that has finished, failed or been aborted does **not** reclaim the inverter’s phase-current callback slot. Reassigning that slot from inside its own invocation destroys the closure being executed, and every one of these procedures fails from inside that callback. Instead each procedure is gated on its own pending completion, and a sample that arrives after the procedure ended is discarded before it reaches anything. This matters most for the inductance injection, whose callback writes duty cycles on every sample: ungated, it would re-arm the peripheral that the abort had just stopped.

`Abort()` is the caller-facing form of the same shutdown. It stops injection, cancels all timers (including the no-sample watchdog), and drops the pending completion **without invoking it** — the

caller that aborted owns the outcome, so a fault raised by the state machine cannot be overwritten by the result of the procedure it interrupted. It covers whichever stage is live: resistance, inductance injection, or the pole-pairs sweep.

The decisions that end a procedure on a sample — the resistance buffer filling, the inductance block completing, a phase current over the limit — are taken in the sampling interrupt, and so is the one thing that must happen at that instant: the injection is stopped there. The completion is not. Computing the result, starting the inductance stage once the resistance is known, and whatever the caller does next — answering over CAN, tracing, starting the next calibration step — is event-loop work. Run from the interrupt, it started the inductance stage from inside the resistance stage's own sample callback, started and cancelled timers there, and in a full calibration took one execution of 6709 cycles, more than five control periods (`software-in-the-loop.md`, Part I). So the outcome is queued and delivered from the event loop. Until then the run is still in progress: samples are discarded, the settle and step timers and the no-sample watchdog do nothing, and `Abort()` drops the queued outcome. The outcome carries the run it belongs to, so one that an abort overtook cannot complete a run started after it. Outcomes decided on the event loop already — the pole-pair count, which the sweep computes on its step timer, and a no-sample failure — complete directly, unless an over-current outcome is already queued, which they then leave to be delivered. The interrupt can also trip while the event loop is between its check and a write to the bridge — the sweep's next step, or the first duty a procedure applies — and that write would re-arm the bridge the interrupt had just stopped, so each such write is followed by a second check that stops the bridge again.

Every estimator is also RAII-safe: if destroyed while active, the destructor cancels all timers and stops the driver without invoking the completion callback.

## ResistanceEstimator

Applies a DC differential voltage across phase A versus phases B and C held at neutral. A `TimerSingleShot` fires after the configured settle time to allow the current to reach steady state. A 5-sample moving-average filter (using a bounded deque of capacity 5) is applied in-flight to each ADC sample before it is pushed to the measurement buffer (bounded vector, capacity 123).

When the buffer is full, the steady-state current is read from the mean of the last 10 % of the buffer. Resistance is computed as  $V_{step}/I_{ss}$  divided by the winding topology factor. If  $I_{ss} \leq 0$ , the result is absent.

**No-sample watchdog:** a second `TimerSingleShot` is armed immediately after the DC duty is applied and resets on every ADC callback (during both settle and measurement phases). Default bound: 100 ms (`Config::noSampleTimeout`). If the ADC produces no callback within that window — whether at startup or after samples stop mid-measurement — the watchdog fires `FailMeasurement()`, stopping the driver and delivering an absent result. This ensures ADC loss does not leave the motor energised indefinitely regardless of the configured settle time.

**RAII cleanup:** the destructor cancels all timers and stops the driver if the estimator is destroyed while a measurement is active, so callers do not need to call `Abort()` explicitly before releasing the object.

**Internal buffers:**

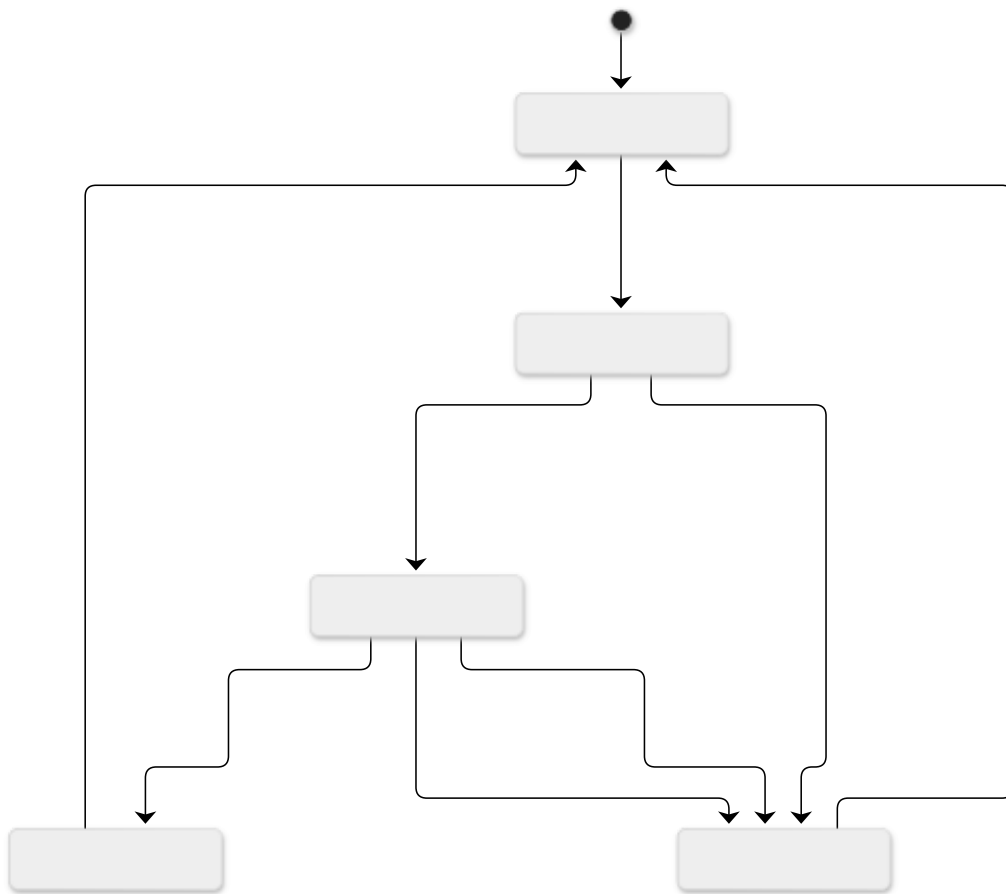


Figure 55: State diagram

| Buffer            | Capacity    | Purpose                                      |
|-------------------|-------------|----------------------------------------------|
| Moving-avg window | 5 samples   | In-flight noise reduction on ADC input       |
| Measurement store | 123 samples | Filtered step-response for steady-state read |

## SinusoidalInductanceEstimator

Injects a sinusoidal voltage on the alpha axis only ( $\beta = 0$ ), keeping net electromagnetic torque at zero so the rotor remains stationary. The injection frequency is snapped to the nearest integer-samples-per-period to align exactly with the Goertzel analysis bin, eliminating spectral leakage.

Two phases run consecutively:

1. **Warmup** — voltage is injected but the Goertzel is not fed. Allows the current transient to decay and the steady-state sinusoidal response to establish. Duration: `warmupPeriods` full injection cycles.
2. **Measurement** — the Goertzel accumulates `measurementPeriods` full cycles of current samples (block size  $N = N_{periods} \times N_{spp}$ , bin  $k = N_{periods}$ ). When `Ready()`, the complex result is delay-corrected (rotation by  $e^{+j\omega d T_s}$  to cancel the  $d$ -sample ADC pipeline lag), then converted to  $\text{Im}(Z)$  and finally to per-phase inductance.

The coherence metric  $\text{fitQuality} = 2|I_{\{\text{Goertzel}\}}|^2 / (N \cdot \Sigma i^2)$  is computed simultaneously from the accumulated current power. It equals 1.0 for a pure sinusoid at  $f_{inj}$  and falls toward zero with increasing noise or rotor motion.

**Working memory (streaming — no sample buffer required):**

| Variable       | Size    | Purpose                                              |
|----------------|---------|------------------------------------------------------|
| Goertzel state | 3 float | $s_1, s_2$ , sample count                            |
| sumSquared     | 1 float | Accumulates $\Sigma i^2$ for <code>fitQuality</code> |
| sampleCount    | 1 int   | Warmup/measurement gate                              |

**PWM bootstrap:** the ADC on TI targets is triggered by the PWM peripheral. Because the resistance estimator stops PWM on completion, the inductance estimator writes a zero-voltage vector immediately after registering the ADC callback to restart the PWM-triggered ADC chain before the first sample is expected.

**No-sample watchdog:** a `TimerSingleShot` is armed when injection begins and resets on every ADC callback. Default bound: 100 ms (`Config::noSampleTimeout`). Loss of ADC samples — whether the bootstrap never triggers or samples stop mid-measurement — causes the watchdog to fire `FailMeasurement()`, stopping the driver and delivering an absent result within the documented bound.

**RAII cleanup:** the destructor cancels the watchdog timer and stops the driver if the estimator is destroyed while a measurement is active.

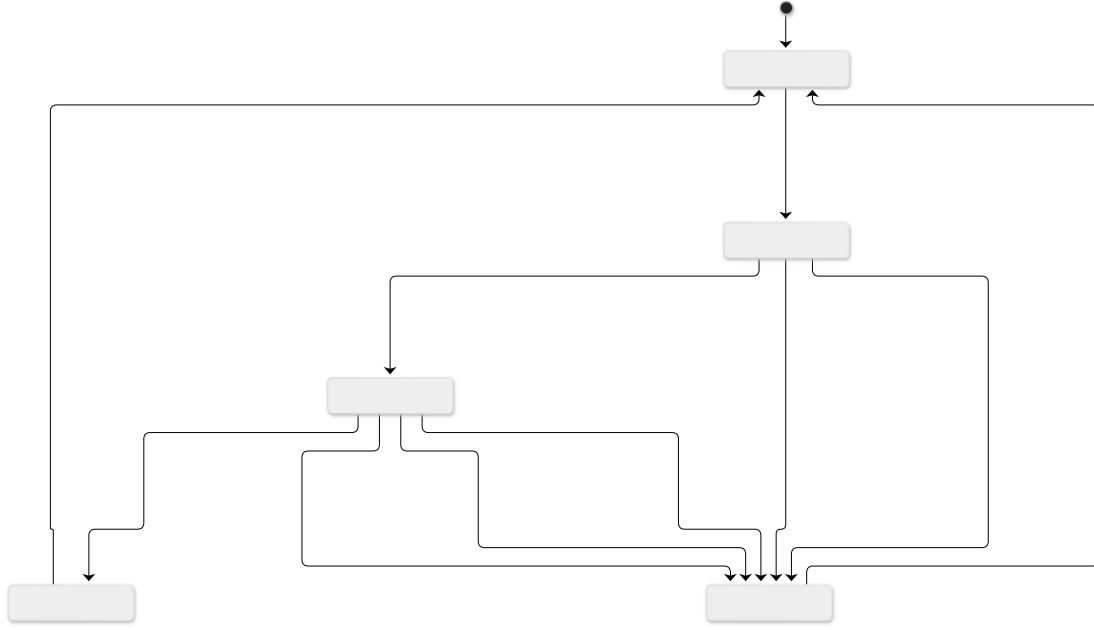


Figure 56: State diagram

### ElectricalParametersIdentificationImpl (orchestrator)

Sequences `ResistanceEstimator` then `SinusoidalInductanceEstimator` and assembles their results into a single `ResistanceInductanceResult`. If resistance estimation fails (absent result), the L stage is skipped and the combined callback fires immediately with an absent result. If the L stage yields `fitQuality < 0.5`, the orchestrator treats this as a fault and includes the low-quality result so the caller can make its own policy decision.

**Concurrent-call invariant:** a second call to `EstimateResistanceAndInductance` while any of the five states above is active causes the new request to be rejected immediately (callback fires with an absent result). The pole-pairs procedure shares this guard: neither procedure may start while the other is active.

---

### Procedure 2 — Pole Pairs Estimation

Starting at electrical angle  $0^\circ$ , the service advances the voltage vector by a small angular increment each settle-timer tick. After each step, the encoder is read and the mechanical displacement is accumulated. Over  $N_{rev}$  full electrical revolutions:

$$p = \text{round}(2\pi N_{rev} / \Delta\theta_{\text{mech},\text{total}})$$

If the total mechanical rotation is below  $\pi/2$  (rotor not following the field), the result is absent.

---

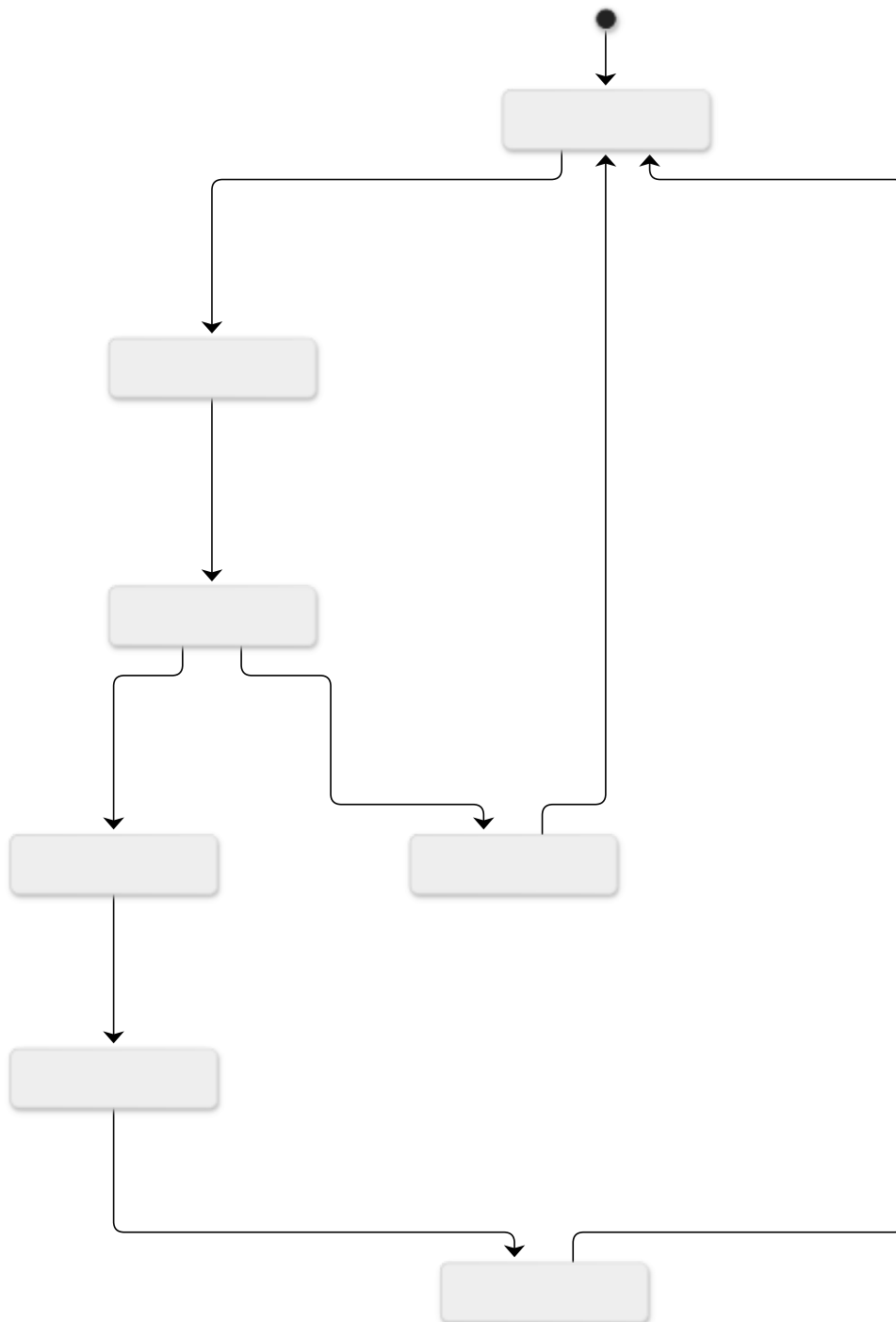


Figure 57: State diagram

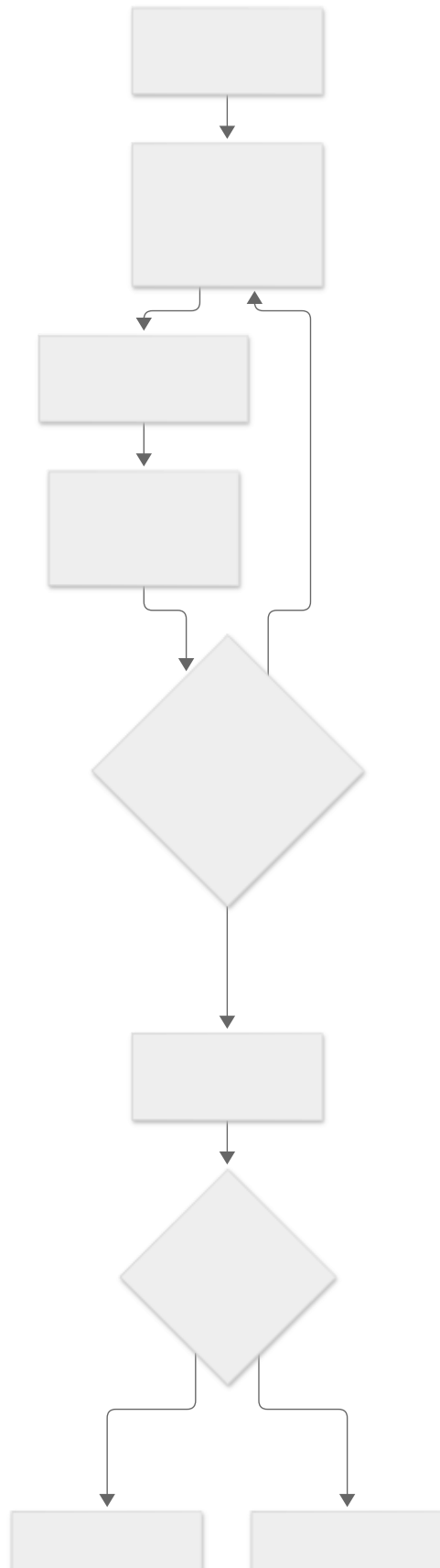


Figure 58: Block diagram

## Interfaces

### Provided

| Interface                                                   | Purpose                                                                                                                               | Contract                                                                                                                                                        |
|-------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>EstimateResistanceAndInductanceResult</code> (onDone) | Since ADC sampling then sinusoidal L estimation; delivers <code>ResistanceInductanceResult{resistance, inductance, fitQuality}</code> | Rejected (immediate empty callback) if pole-pairs procedure is already running; inverter stopped before callback fires; fires exactly once, from the event loop |
| <code>EstimateNumberOfPolePairs</code> (onDone)             | Off pole-pairs (no rotating vector) and delivers <code>optional&lt;size_t&gt;</code> pole pairs                                       | Rejected if R/L procedure is already running; inverter stopped before callback fires; fires exactly once, from the event loop                                   |

### Result Type

`ResistanceInductanceResult` carries:

| Field                   | Type                                    | Absent when                  |
|-------------------------|-----------------------------------------|------------------------------|
| <code>resistance</code> | <code>optional&lt;Ohm&gt;</code>        | DC step finds $I_{ss} = 0$   |
| <code>inductance</code> | <code>optional&lt;MilliHenry&gt;</code> | Goertzel yields $L \leq 0$   |
| <code>fitQuality</code> | float in $[0, 1]$                       | Always present; 0 on failure |

### Required

| Interface                             | Purpose                                                             | Contract                                                 |
|---------------------------------------|---------------------------------------------------------------------|----------------------------------------------------------|
| <code>ThreePhaseInverter</code>       | Voltage application and ADC sampling callbacks for both stages      | Must not be concurrently claimed by any other controller |
| <code>Encoder</code>                  | Reads mechanical angle during pole-pairs sweep                      | Must be initialised before pole-pairs estimation begins  |
| DC bus voltage ( <code>Volts</code> ) | Normalises applied voltage and interprets current in physical units | Must remain stable throughout any active procedure       |

## Online Resistance and Inductance Estimation

In addition to the one-shot calibration procedures above, a continuous online estimator (`RealTimeResistanceAndInductanceEstimator`) runs alongside the closed-loop speed/position controller to track slow parameter drift. See the state machine design document for when online estimates are applied.

## Model

The d-axis voltage equation for a non-salient PMSM is:

$$V_d = R \cdot I_d + L \cdot \left( \frac{dI_d}{dt} - \omega_e \cdot I_q \right)$$

Integrated over one outer-loop period  $T$  it needs no derivative of a sampled current:

$$\overline{V_d} = R \overline{I_d} + L \left( \frac{I_{d,end} - I_{d,start}}{T} - \overline{\omega_e I_q} \right)$$

The regressor vector is  $\phi = [\overline{I_d}, (\Delta I_d/T - \overline{\omega_e I_q})]^T$  in amperes and amperes per millisecond, and the parameter vector is  $\theta = [R, L]^T$  in ohms and millihenries. An RLS algorithm with forgetting factor 0.999 updates  $\theta$  once per outer-loop period (1 kHz).

The averages are accumulated by the control interrupt, tick by tick, and published with each window:

- $\overline{V_d}$  pairs each current sample with the voltage *applied during the tick that produced it* — the previous tick's command, not the one just computed from that same sample. Regressing the same-tick command put the PI's own  $-K_p I_d$  into the output and pulled the resistance negative, by about  $K_p$ .
- The applied vector is fixed in the stator for the tick while the rotor turns through  $\Delta\theta_e$ , so in the rotor frame it arrives advanced by half that angle:  $V_d$  is taken as  $v_d + \frac{1}{2}\Delta\theta_e v_q$ .
- $\overline{\omega_e I_q}$  integrates the rotation each tick actually made,  $\Delta\theta_e \overline{I_q}/T_s$ , rather than the outer loop's last speed, which lags an accelerating rotor.

## Excitation

The product regulates  $I_d$  to zero, so the resistance column is never excited on its own: the recursion drifted, read  $-0.68 \Omega$  for a  $0.36 \Omega$  winding, and could not by construction track a warming one. The speed and position cascades therefore add a d-axis square wave to the current reference while the estimator is attached:  $\pm 2.5\%$  of the drive's current limit at 10 Hz. It produces no torque on a non-salient machine and a copper loss proportional to its square. An update is taken only while  $|\overline{I_d}|$  exceeds 50 mA, so an amplitude of zero switches both the injection and resistance tracking off.

## Publication

Estimates are published after 200 updates since the last seed, and only when both are finite and positive.

## Seeding

When calibration data is loaded (from NVM or after fresh calibration), the online estimator is seeded with the identified values, avoiding a cold-start transient where estimates are physically meaningless.

---

| Field     | Value                                         |
|-----------|-----------------------------------------------|
| Title     | Service: Mechanical Parameters Identification |
| Type      | design                                        |
| Status    | draft                                         |
| Version   | 0.2.0                                         |
| Component | service-mechanical-ident                      |
| Date      | 2026-09-21                                    |

---

**IMPORTANT — Implementation-blind document:** This document describes *behavior, structure, and responsibilities* WITHOUT referencing code. **No code blocks using programming languages (C++, C, Python, CMake, shell, etc.) are allowed.** Use Mermaid diagrams to express behavior instead. Prose descriptions of algorithms are encouraged; source-level details are not.

**Diagrams:** All visuals must be either a Mermaid fenced code block (````mermaid`) or ASCII art inline in the document. External image references using Markdown image syntax are **not allowed**.

---

## Responsibilities

**Is responsible for:**

- Estimating rotor moment of inertia (J) and viscous friction coefficient (B) while the motor runs in closed-loop speed control
- Starting and stopping the drive for the duration of the procedure, and driving the two-level speed trajectory that provides the excitation
- Refusing a request whose configuration does not describe a bounded trajectory, and ending a run that leaves the current or speed envelope it was given
- Reporting an estimate only when it is converged and physically plausible, and reporting absent values otherwise
- Releasing the drive and discarding the pending completion when the caller aborts, so that a fault is not overwritten by the result of the procedure it interrupted
- Computing instantaneous electromagnetic torque from the q-axis current and the caller-supplied torque constant
- Estimating angular acceleration by finite-differencing successive speed measurements obtained from the encoder
- Maintaining a real-time Recursive Least Squares (RLS) estimator that continuously refines the parameter vector [J, B,  $\tau_{\text{friction}}$ ] using each new observation
- Enforcing a configurable one-shot estimation timeout, after which the latest RLS estimates are reported and the procedure stops
- Delivering results through a single completion notification that reports friction and inertia using explicit physical units

**Is NOT responsible for:**

- Configuring the speed control loop — the loop's tunings and algorithm selection are established before the procedure begins, and the caller is responsible for the electrical model and provisional mechanics that make those loops able to move the rotor at all (see the State Machine design document)
- Persisting the returned parameters — the caller decides what to do with them
- Auto-tuning the speed controller — this service provides the plant parameters that a separate tuning step may consume
- Measuring electrical parameters (R, L, pole pairs) — those are handled by the Electrical Parameters Identification service

---

## Component Details

### Prerequisites and Motor Model

The mechanical dynamics of a PMSM rotor are governed by Newton's second law for rotation:

$$\tau_{\text{motor}} = J \cdot d\omega/dt + B \cdot \omega + \tau_{\text{friction}}$$

where:

- $\tau_{\text{motor}}$  — electromagnetic torque produced by the motor ( $\text{N} \cdot \text{m}$ )
- $J$  — rotor moment of inertia ( $\text{kg} \cdot \text{m}^2$ )
- $B$  — viscous friction coefficient ( $\text{N} \cdot \text{m} \cdot \text{s}/\text{rad}$ )
- $\tau_{\text{friction}}$  — Coulomb (constant) friction term ( $\text{N} \cdot \text{m}$ )
- $\omega$  — rotor angular velocity ( $\text{rad}/\text{s}$ )
- $d\omega/dt$  — rotor angular acceleration ( $\text{rad}/\text{s}^2$ )

This is a linear model in the three unknowns  $[J, B, \tau_{\text{friction}}]$ . The RLS estimator treats each measurement instant as one observation of this linear equation.

### Torque Estimation

At each sampling callback, the instantaneous electromagnetic torque is estimated from the q-axis current:

$$\tau_{\text{motor}} = K_t \times I_q$$

where  $K_t$  is the torque constant ( $\text{N} \cdot \text{m}/\text{A}$ ) provided by the caller. The q-axis current  $I_q$  is the measured value the control loop is already sampling — the same current feedback path used by the FOC current controller.

The d-axis contribution to torque is zero for surface PMSM under the  $I_d = 0$  operating condition assumed during identification. For interior PMSM with reluctance torque, this approximation introduces a small systematic error in the torque estimate; the caller should account for this if  $L_d \neq L_q$ .

### Speed and Acceleration Estimation

Angular velocity ( $\omega$ ) at each observation instant is obtained from two successive encoder angle samples using a finite difference with wrap-around compensation:

$$\omega_k = (\theta_k - \theta_{\{k-1\}}) / \Delta t$$

where  $\Delta t$  is the sampling period and wrap-around compensation shifts the raw difference into the range  $(-\pi/\Delta t, +\pi/\Delta t)$  rad/s in exactly the same manner used by the speed control outer loop.

$\Delta t$  must be the period of the callback that produced the two samples. Observations arrive with the control loop's current samples, at the inverter's **base** frequency rather than the speed-command frequency, so  $\Delta t$  is derived from **BaseFrequency()**. Deriving it from the outer-loop rate instead scales  $\omega$  by the ratio of the two frequencies and  $\alpha$  by its square — at 20 kHz against 1 kHz that is 20× on speed and 400× on inertia, and the inertia estimate becomes speed-loop PID gains.

## Obtaining the observations

The inverter carries a **single** phase-current callback slot, and that slot is the control loop's only path to the PWM output. A procedure that claims it for itself evicts the control loop: no duty cycles are written, the inverter is never started, the rotor stays where it is, and the regressor carries no excitation for the estimator to converge on. The procedure then runs to its timeout and reports failure — on hardware only, since a test double that publishes samples on demand hides the whole effect.

So this procedure does not claim the slot. It registers as an **observer** on the running control loop, which publishes the phase currents it has just used, after the duty cycles are written. Starting and stopping the drive is likewise done through the controller rather than by writing the inverter directly.

Two consequences follow from the observer running in the ADC interrupt:

- The procedure's completion is handed to the event dispatcher rather than invoked from the interrupt. Releasing the observer from inside its own invocation would destroy the closure being executed, and the completion reaches non-volatile memory through the state machine.
- An abort takes effect immediately — the drive is stopped and the observer released — but the pending completion is dropped rather than invoked. The caller that aborted owns the outcome.
- Because the completion is deferred, an abort and a new run can both happen before the dispatcher gets to it. Each run therefore carries a generation, and a queued completion finishes only the run that queued it; a terminal outcome belonging to an aborted run can never complete the run that followed it.

Each observation reads the encoder once. Reading it twice within a callback samples two different rotor positions and mixes them into a single difference.

Angular acceleration ( $d\omega/dt$ ) is obtained from two successive velocity estimates by a further finite difference:

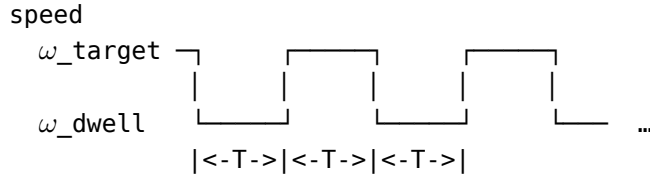
$$\alpha_k = (\omega_k - \omega_{k-1}) / \Delta t$$

The double finite difference amplifies noise; the quality of the acceleration estimate therefore depends on the encoder resolution and sampling rate. Low-resolution encoders or very low speeds produce noisy acceleration estimates and degrade identification accuracy. The caller is advised to command a non-zero target speed of sufficient magnitude to obtain a good signal-to-noise ratio.

## Excitation Trajectory

A single constant speed setpoint is not an excitation. Once the loop has settled on it the regressor reads  $[1, 0, \omega]$  with  $\omega$  constant: the acceleration column is zero, so  $J$  is not identified at all, and the intercept and speed columns are collinear, so  $B$  cannot be separated from the Coulomb term. Every sample then adds information in no direction while the forgetting factor keeps inflating the covariance.

The procedure therefore drives a two-level trajectory. It commands the target speed, then alternates between the target speed and a lower dwell speed every dwell period until the run ends. Each transition is a commanded acceleration, and the rotor spends the whole run above the minimum speed the update gate requires, so both the inertia and the friction directions are excited repeatedly.



## Bounded Envelope

The trajectory is given explicit limits and the run stays inside them:

| Limit    | Meaning                                                                                        |
|----------|------------------------------------------------------------------------------------------------|
| current  | The largest phase-current magnitude the run may produce, clamped to what the inverter supports |
| speed    | The largest rotor speed the run may reach, measured rather than commanded                      |
| duration | The timeout after which the run ends whether or not it has converged                           |

On the first sample whose phase current exceeds the current envelope in any phase and in either direction, or whose measured speed exceeds the speed limit, the run ends and reports absent values.

The power stage is stopped **in the interrupt that observed the sample**, not with the rest of the teardown. Deferring it would leave the inverter driving an out-of-envelope current for as long as the dispatcher takes to run, which is the one thing the envelope exists to prevent; stopping the inverter touches no callback slot, so it is safe from the interrupt, whereas releasing the observer there would destroy the closure being executed. Releasing the observer, stopping the drive through the controller and delivering the completion therefore stay deferred, as they are for every other way a run ends.

The envelope is evaluated on **every** sample for as long as the drive is live — including samples that arrive after a converged one has queued its completion but before the dispatcher has run. A run is only over once the drive has been released, so a sample that leaves the envelope in that window still invalidates the estimate rather than letting a converged value through.

A configuration that does not describe a usable bounded trajectory — a dwell speed that is not below the target speed, a non-positive envelope, a speed limit that is not finite and positive or that sits below the target, a timeout that does not outlast one dwell period, a forgetting factor outside  $(0, 1]$  — is refused through the completion rather than started, as is a request without a positive torque constant or pole-pair count. The speed limit in particular must be finite: an infinite bound passes every comparison against the target speed while making the run-time check unreachable, which is a bounded trajectory in name only.

## Recursive Least Squares Estimator

The RLS estimator maintains a  $3 \times 1$  parameter vector  $\theta = [J, B, \tau_{\text{friction}}]^T$  and a  $3 \times 3$  covariance matrix  $P$ . At each observation  $k$ , a  $1 \times 3$  regressor row vector is formed:

$$\varphi_k = [d\omega/dt_k, \omega_k, 1]$$

and the scalar observation is:

$$y_k = \tau_{\text{motor}_k}$$

The prediction error is:

$$\mathbf{e}_k = \mathbf{y}_k - \boldsymbol{\varphi}_k \cdot \boldsymbol{\theta}_{k-1}$$

The Kalman gain vector is:

$$\mathbf{K}_k = \mathbf{P}_{k-1} \cdot \boldsymbol{\varphi}_k^T \cdot (\lambda + \boldsymbol{\varphi}_k \cdot \mathbf{P}_{k-1} \cdot \boldsymbol{\varphi}_k^T)^{-1}$$

The parameter and covariance updates are:

$$\boldsymbol{\theta}_k = \boldsymbol{\theta}_{k-1} + \mathbf{K}_k \cdot \mathbf{e}_k$$

$$\mathbf{P}_k = (\mathbf{P}_{k-1} - \mathbf{K}_k \cdot \boldsymbol{\varphi}_k \cdot \mathbf{P}_{k-1}) / \lambda$$

The scalar  $\lambda$  ( $0 < \lambda \leq 1$ ) is the **forgetting factor** (default 0.9995). Values less than 1 cause older observations to carry exponentially decreasing weight, which is important for tracking slowly time-varying loads. At  $\lambda = 1$  the estimator is an ordinary batch least squares — it never forgets.

The RLS state ( $\theta$  and  $P$ ) is held in a statically allocated object (`std::optional<MotorRLS>`) that is emplaced in-place at the start of each procedure invocation and destroyed when the procedure ends. This avoids heap allocation while still permitting the state to be absent between procedures.

## Timeout and Result Delivery

A `TimerSingleShot` starts when the procedure begins. When it fires, the current RLS parameter vector is read and the results are extracted:

- **J** (inertia) is the first element of  $\theta$ ; reported as `NewtonMeterSecondSquared`.
- **B** (viscous friction) is the second element of  $\theta$ ; reported as `NewtonMeterSecondPerRadian`.
- **$\tau_{\text{friction}}$**  (Coulomb friction) is the third element of  $\theta$ ; it is observed internally but is not part of the output interface for this version.

## Acceptance Policy

The procedure and the online estimator described below share one acceptance policy, so a value one of them refuses is refused by the other:

| Gate              | Rule                                                                                                                                                        |
|-------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------|
| excitation        | An observation updates the estimator only when the rotor is both moving and accelerating: speed magnitude and acceleration magnitude are each above a floor |
| informed estimate | At least a minimum number of such observations must have been taken since the estimator was started or seeded                                               |
| convergence       | The innovation must be below a fixed bound and the covariance trace below a bound scaled to the forgetting factor                                           |
| plausibility      | J must be finite, strictly positive and below an upper bound; B must be finite, non-negative and below an upper bound                                       |

The covariance bound is expressed relative to the forgetting factor because an RLS that forgets cannot drive its covariance below a floor proportional to  $(1 - \lambda)$ . A single absolute bound would be structurally unreachable for a tracking estimator with a short memory and trivially met by one that never forgets.

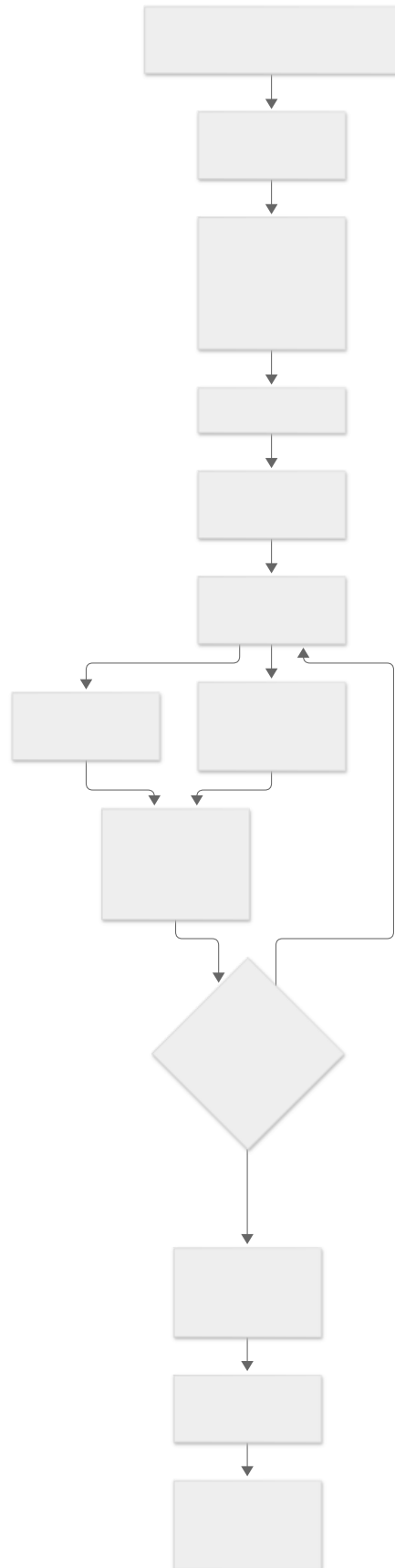
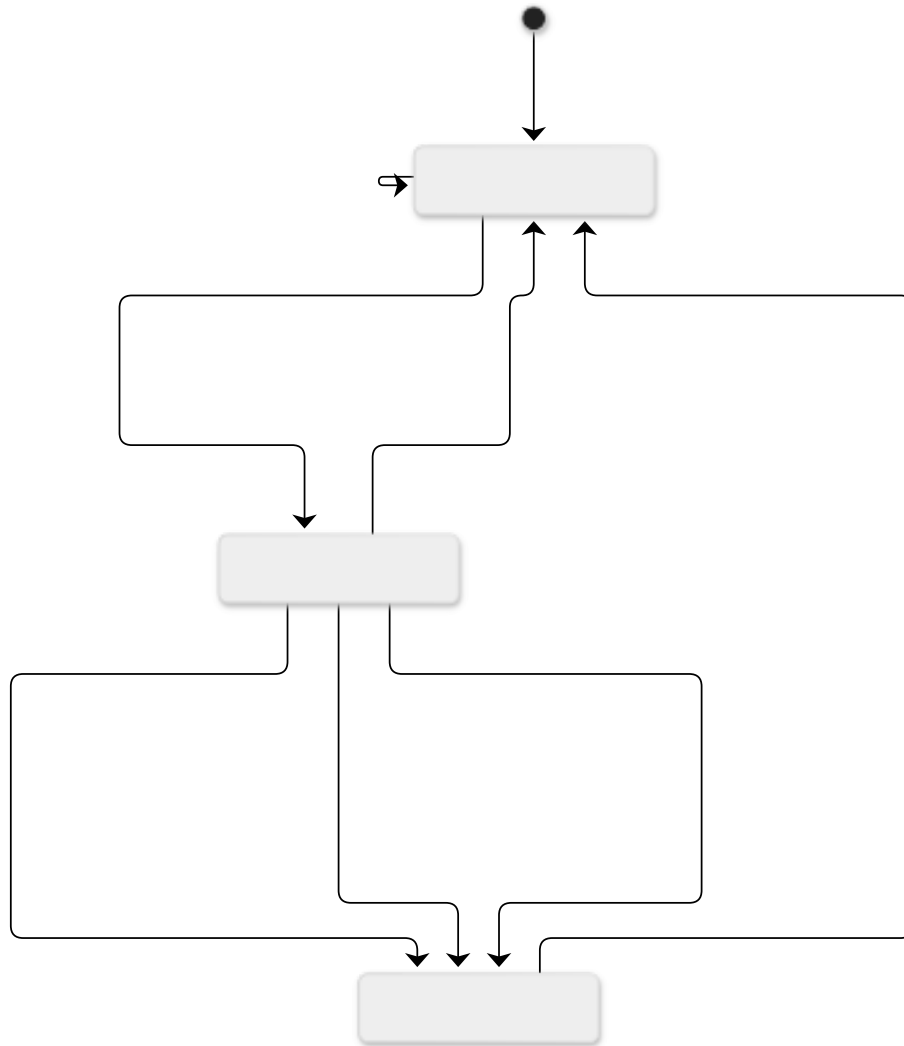


Figure 59: Block diagram



## Interfaces

### Provided

| Interface                                                                          | Purpose                                                         | Contract                                                                                                                                                                         |
|------------------------------------------------------------------------------------|-----------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>EstimateFrictionAndInertia(torqueConstant, polePairs, config, onDone)</code> | under active speed control along a bounded two-level trajectory | Delivers <code>(optional&lt;NewtonMeterSecondPerRadian&gt;, optional&lt;NewtonMeterSecondSquared&gt;)</code> exactly once; absent unless the estimate is converged and plausible |

A request is rejected through its own completion, with both values absent, when a run is already in flight, when the configuration does not describe a usable bounded trajectory, or when the torque constant or pole-pair count is not positive.

### Required

| Interface                       | Purpose                                                                                   | Contract                                                                                                                                                                       |
|---------------------------------|-------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>FocSpeed</code>           | Commands the two speed setpoints that excite the mechanical dynamics                      | Must already be active and in control of the motor before the procedure begins, with an electrical model and mechanics applied so its gains and current envelope are not inert |
| <code>ThreePhaseInverter</code> | Supplies ADC current callbacks that supply the $I_q$ measurement on each computation step | Must not be stopped during the estimation procedure                                                                                                                            |
| <code>Encoder</code>            | Supplies mechanical angle samples for speed and acceleration estimation                   | Must be tracking position at the configured sampling rate                                                                                                                      |

## Online Inertia and Friction Estimation

In addition to the one-shot timed procedure above, a continuous online estimator (`RealTimeFrictionAndInertiaEstimator`) runs alongside the closed-loop speed controller to track slow mechanical parameter drift.

### Model

The mechanical equation of motion for the motor shaft is:

$$T_e = J \cdot \dot{\omega} + B \cdot \omega + \tau_c$$

The electromagnetic torque is approximated as  $T_e = I_q \cdot k_t$ , where  $k_t = \frac{3}{2}p\psi_f$  is derived from the calibration record's pole pairs and flux linkage. Angular acceleration  $\dot{\omega}$  is derived from a finite difference of successive speed measurements scaled by the sampling frequency.

The regressor vector is  $\phi = [1, \dot{\omega}, \omega]^T$ , and the parameter vector is  $\theta = [\tau_c, J, B]^T$ . The scalar output is  $T_e$ . An RLS algorithm with a forgetting factor of 0.9995 updates  $\theta$  each outer-loop period from window averages the control interrupt accumulates.

## Torque Constant Dependency

The torque constant  $k_t$  must be provided before the estimator updates begin. In normal operation the state machine derives it from the calibration NVM record as  $\frac{3}{2}p\psi_f$  (`core/foc/math/TorqueConstant.hpp`) and supplies it via `SetTorqueConstant()` during the `EnterEnabled` transition; the one-shot procedure receives the same value.

A wrong  $k_t$  scales every identified  $J$  and  $B$  by the same factor, which is why it is not a per-target constant. Updates run opportunistically while the FOC controller is active.

## Persistence of Excitation

At standstill the regressor degenerates to  $\phi = [1, 0, 0]^T$ : the constant intercept is excited but the inertia and friction directions are not. The coefficients barely move, but the covariance  $P$  is **not** self-gating — an RLS update with a forgetting factor inflates  $P$  by  $\lambda^{-1}$  in every unexcited direction on every sample, so  $P$  grows as  $\lambda^{-n}$ . At  $\lambda = 0.995$  and 20 kHz that reaches roughly  $5 \times 10^{21}$  after ten seconds of standstill, and the first sample of real excitation then produces an enormous coefficient jump.

Standstill observations (below a minimum speed) are therefore skipped entirely, so the covariance is frozen rather than inflated and the previous coefficients are reported unchanged.

A rotor held at a constant speed is a different case. It excites neither the inertia direction nor, on its own, the friction direction separately from the intercept, but a trajectory that holds *two* speeds is exactly what separates friction from the intercept. An earlier gate refused every sample whose acceleration was under 1 rad/s<sup>2</sup>, which removed those plateaus; the offline friction estimate stayed accurate only because whole-percent duty ripple put a spurious acceleration on each plateau sample and let it through. With the duty carried as a fraction the ripple vanished, the plateaus were refused, and the friction the full calibration identified moved 19–27 % off the plant.

Both procedures now take every sample above the minimum speed, and count separately what the estimate needs before it may be published: at least 64 samples accelerating faster than 20 rad/s<sup>2</sup> (inertia), and observations spanning more than a quarter of the fastest speed seen (friction apart from the intercept). The shared `MechanicalExcitation` tracks both. The online estimator, which runs indefinitely, admits a non-accelerating sample only while the covariance trace is under a ceiling, so a long constant-speed run cannot inflate the inertia direction without bound; the offline run is bounded by its timeout and its trajectory re-excites every dwell.

## Plausibility Band and Update Rate

A finiteness test alone admits values such as  $10^{30}$ . Before an estimate is published to `CurrentInertia()` / `CurrentFriction()` — and therefore before it can become PID gains — it must pass every gate of the shared acceptance policy above: enough accelerating observations and a span of speeds since the last seed, a settled fit, and a value inside the physical band. The offline procedure judges the fit by its innovation and covariance; the online estimator, which runs on outer-loop window averages (see `documentation/theory/friction-inertia-estimation.md` §4.1), by an averaged residual power under 1 % of the averaged output power, because an instantaneous innovation

of  $0.1 \text{ mN} \cdot \text{m}$  sits under any real drive's torque ripple. A publication is therefore rate-limited by construction — no estimate reaches the accessors during the warm-up that follows a seed, and none reaches them while the fit is still moving. An estimate that fails any gate is discarded and the last accepted pair is retained.

## Seeding and Warm Start

When calibration data is loaded (from NVM or after a fresh calibration run), the online estimator is seeded with the calibration values  $(J_{cal}, B_{cal})$ . This initialises the RLS coefficient vector to  $\theta = [0, J_{cal}, B_{cal}]^T$ , setting the Coulomb-friction intercept to zero and warm-starting the inertia and viscous-friction estimates at the identified operating point. This avoids a cold-start transient where estimates are physically meaningless during the initial operating period.

## Forgetting Factor

The online forgetting factor  $\lambda = 0.9995$  applies an exponential weight decay to past observations, enabling the estimator to track gradual mechanical changes over the motor lifetime (bearing wear increases friction; load changes affect effective inertia).

## Estimate Consumption

Estimates are not applied automatically. The operator or application explicitly triggers a speed-PID retune via `ApplyOnlineEstimates()` on the state machine. See the State Machine design document for details.

| Field     | Value                        |
|-----------|------------------------------|
| Title     | Service: Non-Volatile Memory |
| Type      | design                       |
| Status    | draft                        |
| Version   | 0.3.0                        |
| Component | service-nvm                  |
| Date      | 2026-04-07                   |

**IMPORTANT — Implementation-blind document:** This document describes *behavior, structure, and responsibilities* WITHOUT referencing code. **No code blocks using programming languages (C++, C, Python, CMake, shell, etc.) are allowed.** Use Mermaid diagrams to express behavior instead. Prose descriptions of algorithms are encouraged; source-level details are not.

**Diagrams:** All visuals must be either a Mermaid fenced code block (````mermaid`) or ASCII art inline in the document. External image references using Markdown image syntax are **not allowed**.

## Responsibilities

**Is responsible for:** - Persisting motor calibration data (R, L, pole pairs, encoder offset, PID gains, inertia, friction) and system configuration (current limits, velocity limits, CAN parameters, teleme-

try rate, voltage thresholds) across power cycles using MCU internal EEPROM - Ensuring data integrity on every read through CRC-32 verification and on every write through read-back verification - Detecting and signalling version mismatches when a stored record was written by a different firmware version - Distinguishing the calibration and configuration regions by per-region magic numbers, preventing cross-region misinterpretation of data - Providing non-blocking, callback-based operations — no operation may block the calling context - Providing a **Format** operation that erases both regions to factory state

**Is NOT responsible for:** - Raw EEPROM hardware access — all erase, read, and write operations are delegated to injected **NvmRegion** instances - Computing the calibration or configuration values themselves — those are produced by the identification services and the application - Providing wear-levelling or journaling — a single record per region is maintained without historical versioning

## Component Details

### Logical Regions

The service manages two independent EEPROM regions. Each region holds exactly one record at a time — the latest write replaces any previous record in that address range.

| Region        | Magic      | Data structure  | Data size |
|---------------|------------|-----------------|-----------|
| Calibration   | 0xCAFEF00D | CalibrationData | 56 bytes  |
| Configuration | 0xDEADBEEF | ConfigData      | 20 bytes  |

The two regions use different **NvmRegion** instances (injected at construction) and can be read, written, and erased independently. An operation on the calibration region has no effect on the configuration region and vice versa.

### On-Media Record Format

Each region stores a single record consisting of a fixed-size header followed by the data payload. The header fields, in order, are:

| Field   | Size    | Description                                            |
|---------|---------|--------------------------------------------------------|
| Magic   | 4 bytes | Region identifier (0xCAFEF00D or 0xDEADBEEF)           |
| Version | 1 byte  | Layout version constant baked into firmware            |
| CRC-32  | 4 bytes | CRC-32 of the data bytes only (header excluded)        |
| Data    | N bytes | Serialized <b>CalibrationData</b> or <b>ConfigData</b> |

This layout ensures that a calibration record stored in EEPROM cannot be silently misinterpreted as a configuration record (and vice versa) because any cross-region read will fail the magic check immediately.

## Integrity Checks on Read

When a region is read back, the following checks are applied in order:

1. **Magic check** — the four-byte magic field must match the expected value for that region. A mismatch yields `NvmStatus::InvalidData`.
2. **Version check** — the version byte must match the version constant compiled into the current firmware. A mismatch yields `NvmStatus::VersionMismatch`, allowing the application to handle firmware upgrades gracefully.
3. **CRC-32 check** — a CRC-32 is computed over the data bytes and compared to the stored CRC field. A mismatch yields `NvmStatus::InvalidData`.

Only if all three checks pass is the data considered valid and `NvmStatus::Ok` reported.

These three checks establish that the record is *intact*, not that its values are *usable*: a record whose CRC matches can still hold a non-finite float. `FocStateMachineCommon::IsValidCalibration()` therefore applies a plausibility gate on top, requiring the electrical fields to be finite (and the phase resistance positive) before the restored calibration can carry the drive to **Ready**. The same gate runs on externally supplied calibration in `CmdCompleteExternalCalibration`, so a value that arrives over CAN cannot be persisted either. Positive infinity passes every `> 0` test, which is why finiteness is checked explicitly rather than inferred from a range comparison.

## Write Sequence

Every write operation follows a three-step sequence to protect against partial writes (e.g., due to unexpected power loss after the erase but before the write completes). Each step is driven by an asynchronous callback from the `NvmRegion` abstraction; no step blocks:

The read-back verification step catches EEPROM hardware faults, programming errors, and marginal cells that pass the erase but corrupt the write. `NvmStatus::WriteFailed` is specifically reserved for this condition.

## EEPROM Address-Space Map

The two NVM regions occupy fixed, non-overlapping address ranges within the MCU EEPROM. Addresses are byte offsets from the start of the EEPROM address space.

| Region        | Base address | Size      | Content                  |
|---------------|--------------|-----------|--------------------------|
| Calibration   | 0            | 128 B     | One calibration record   |
| Configuration | 128          | 128 B     | One configuration record |
| Reserved      | 256+         | remainder | Unused; not accessed     |

Both regions fit comfortably within the 2048-byte EEPROM available on the TM4C1294 target. ST and Host targets use an in-memory stub of identical layout for regression testing.

## Power-Loss Mitigation

EEPROM cells retain data indefinitely once a write word-program cycle completes. The write sequence (erase region → write record → read-back verify) is designed so that power loss at any point leaves the system in a known state:

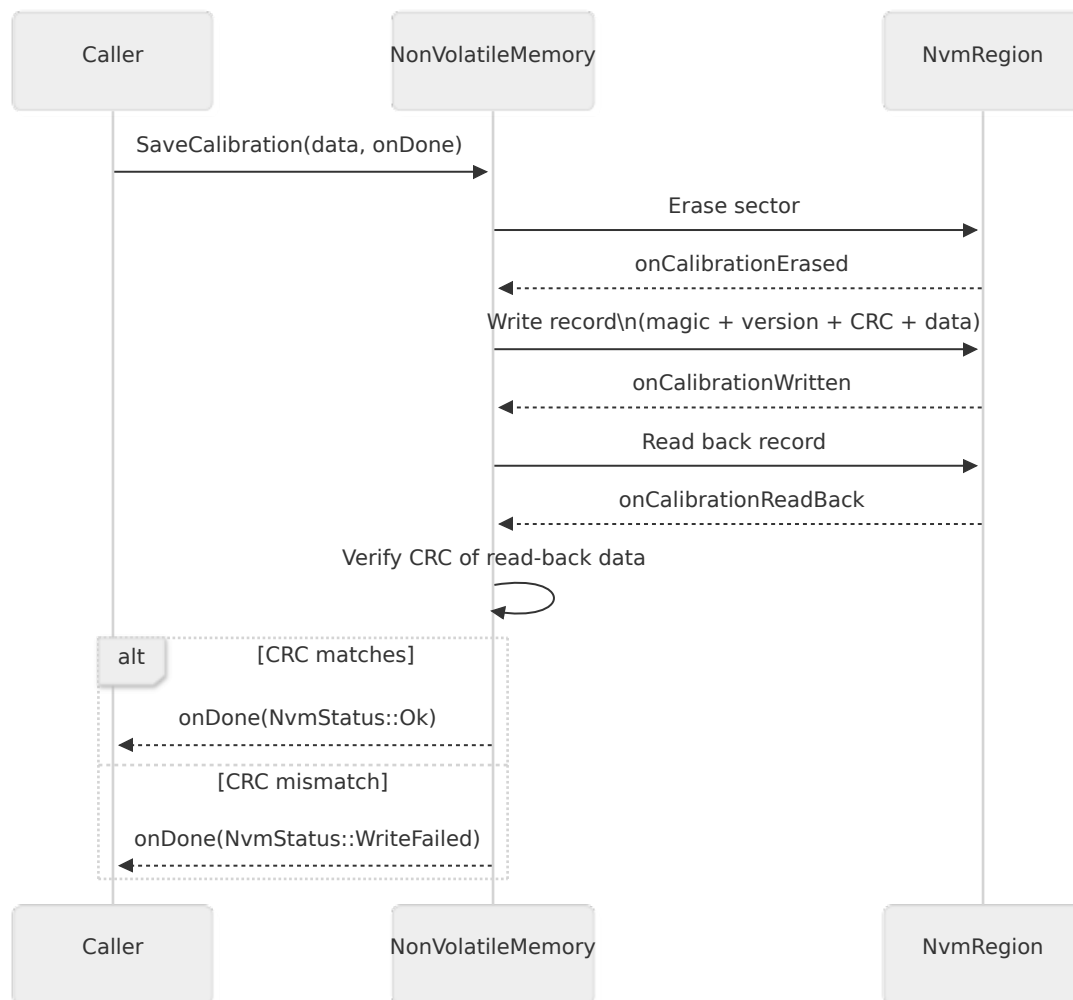


Figure 61: Sequence diagram

- **Power loss during erase:** The region contains all-0xFF bytes. On next read, the magic check fails and `NvmStatus::InvalidData` is returned — the application must re-run identification or use factory defaults.
- **Power loss during write:** The partially-written record will fail either the magic check or the CRC-32 check on next read. The same `InvalidData` result is returned; no silent corruption is possible.
- **Power loss during read-back:** The write has already completed. On next power-on, the record will re-read correctly. The application may retry the write if the previous session reported `WriteFailed`.

No journaling or wear-levelling is implemented. Each write operation overwrites the same address range (erase-then-write), which is sufficient for infrequent commissioning writes. If frequent writes are required, a wear-levelling strategy must be added at the `NvmRegion` layer.

## Buffer Management — No Heap

Each region owns two statically allocated byte arrays:

- **Write buffer** — sized to hold one complete record (header + data); populated before the `NvmRegion::Write` call.
- **Read-back buffer** — same size; populated after the `NvmRegion::Read` call for post-write CRC verification.

Both buffers are `std::array<uint8_t, recordSize>` members of the service object. They are reused across invocations. No heap is used at any point.

## Concurrency — One Operation At A Time, Device-Wide

Both logical regions are backed by a single EEPROM device, and that device driver serialises access: it accepts exactly one erase, program, or read at a time and treats a second concurrent request as a programming error. Admission control is therefore **device-wide, not per region**: while any operation is in flight, every entry point — calibration save, load, invalidate, validity check, configuration save, load, reset-to-defaults, and format — is refused. No refused request ever reaches a region.

A refused request never silently disappears. Its callback is invoked immediately with `Busy`; the validity check, whose callback carries a boolean rather than a status, is answered `false`. Callers that arm a pending-command callback before issuing a request therefore always see it complete, and cannot be wedged by a dropped notification.

Refusal is decided before any internal state is touched, so a caller that issues a further request from inside a synchronously delivered `Busy` callback can only be refused in turn — re-entry cannot corrupt an in-flight operation. Admission is released just before the completion callback of the in-flight operation runs, so a follow-up request issued from that callback is accepted normally.

Callers are expected to treat `Busy` as “nothing was written”: any in-memory state speculatively updated in anticipation of a successful write must be rolled back, and the originating command rejected cleanly rather than escalated to a fault.

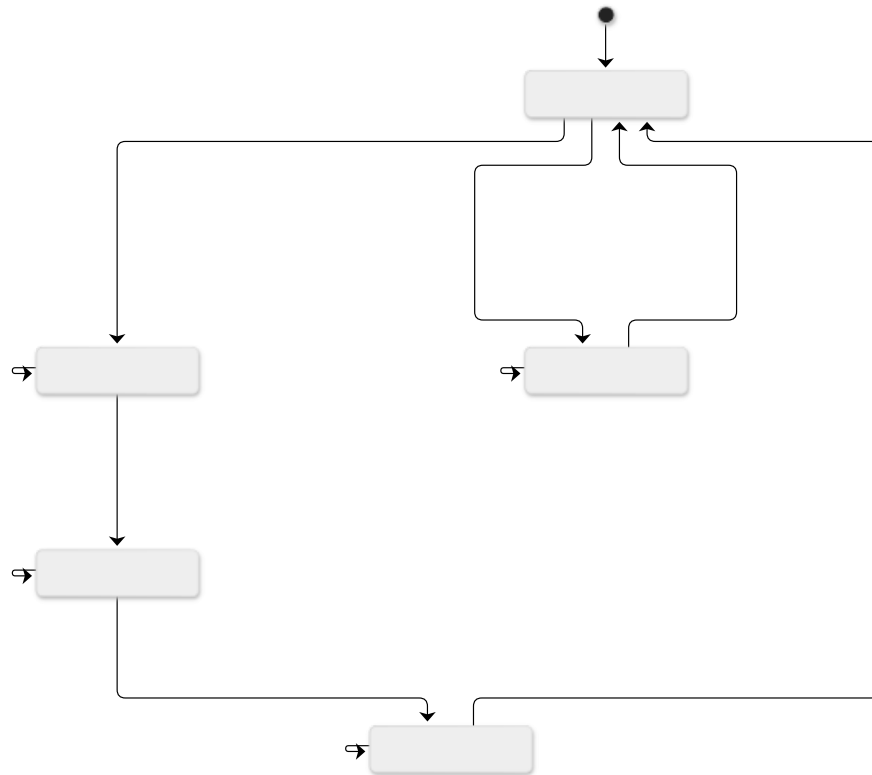


Figure 62: State diagram

## State Machine (Device-Wide)

### Error Codes

| Status          | Meaning                                                                    |
|-----------------|----------------------------------------------------------------------------|
| Ok              | Operation completed successfully                                           |
| InvalidData     | Magic mismatch or CRC-32 mismatch on read                                  |
| VersionMismatch | Version byte in stored record does not match firmware constant             |
| WriteFailed     | CRC-32 of read-back data does not match what was written                   |
| HardwareFault   | NvmRegion reported a hardware-level error (e.g., erase or program failure) |
| Busy            | Another operation is in flight; nothing was read or written                |

## Calibration and Configuration Data Fields

**CalibrationData** (52 bytes total):

| Field              | Physical unit | Description                               |
|--------------------|---------------|-------------------------------------------|
| rPhase             | $\Omega$      | Measured phase resistance                 |
| lD                 | H             | Measured d-axis inductance                |
| lQ                 | H             | Measured q-axis inductance                |
| currentOffsetA/B/C | A             | ADC current sensor zero offsets per phase |

| Field                | Physical unit                                       | Description                                         |
|----------------------|-----------------------------------------------------|-----------------------------------------------------|
| inertia              | $\text{kg} \cdot \text{m}^2$                        | Estimated rotor inertia                             |
| frictionCoulomb      | $\text{N} \cdot \text{m}$                           | Coulomb (constant) friction                         |
| frictionViscous      | $\text{N} \cdot \text{m} \cdot \text{s}/\text{rad}$ | Viscous friction coefficient                        |
| encoderZeroOffset    | rad                                                 | Encoder calibration offset (from alignment service) |
| currentLoopBandwidth | rad/s                                               | Current loop closed-loop bandwidth                  |
| speedLoopBandwidth   | rad/s                                               | Speed loop closed-loop bandwidth                    |
| encoderDirection     | —                                                   | Polarity correction (+1 or −1)                      |
| polePairs            | —                                                   | Number of motor pole pairs                          |

**ConfigData** (20 bytes total):

| Field              | Physical unit | Description                                  |
|--------------------|---------------|----------------------------------------------|
| canNodeId          | —             | CAN bus node identifier                      |
| canBaudrate        | bit/s         | CAN bus baud rate                            |
| telemetryRateHz    | Hz            | Rate at which telemetry frames are sent      |
| encoderResolution  | —             | Encoder counts per mechanical revolution     |
| defaultControlMode | —             | Control mode selected on power-up            |
| currentAlgorithm   | —             | Current-loop algorithm selected on power-up  |
| speedAlgorithm     | —             | Speed-loop algorithm selected on power-up    |
| positionAlgorithm  | —             | Position-loop algorithm selected on power-up |

The four selection fields are stored as raw byte identifiers. A record can pass the integrity check and still hold a byte outside the valid range for its selection, so each value is range-checked against its algorithm set before it is applied on power-up. A value outside the range, or a selection the current motor parameters do not support, leaves the loop on its built-in default and the stored byte is corrected to match what is actually running.

## Interfaces

### Provided

| Interface                             | Purpose                                                              | Contract                                                                           |
|---------------------------------------|----------------------------------------------------------------------|------------------------------------------------------------------------------------|
| <b>SaveCalibration</b> (Data, onDone) | Writes calibration sector, writes record, verifies read-back         | <b>onDone(NvmStatus)</b> fires exactly once; <b>Busy</b> if the device is occupied |
| <b>LoadCalibration</b> (onDone)       | Reads calibration sector and integrity-checks the calibration record | <b>onDone(NvmStatus)</b> fires exactly once; <b>Busy</b> if the device is occupied |
| <b>InvalidateCalibration</b> (onDone) | Erases calibration sector without writing a new record               | Makes <b>IsCalibrationValid</b> return false; <b>onDone(NvmStatus)</b> fires once  |
| <b>IsCalibrationValid</b> (onDone)    | Integrity-checks the calibration record, reporting validity only     | <b>onDone(bool)</b> fires exactly once; <b>false</b> if the device is occupied     |

| Interface                                 | Purpose                                                                       | Contract                                                                                 |
|-------------------------------------------|-------------------------------------------------------------------------------|------------------------------------------------------------------------------------------|
| <code>SaveConfig(data, onDone)</code>     | Same write+verify sequence for the configuration region                       | <code>onDone(NvmStatus)</code> fires exactly once; <b>Busy</b> if the device is occupied |
| <code>LoadConfig(onDone)</code>           | Reads and integrity-checks the configuration record                           | <code>onDone(NvmStatus)</code> fires exactly once; <b>Busy</b> if the device is occupied |
| <code>ResetConfigToDefault(onDone)</code> | Writes (onDone) containing factory-default values to the configuration region | Follows the same erase/write/verify sequence as <b>SaveConfig</b>                        |
| <code>Format(onDone)</code>               | Erases both calibration and configuration regions                             | <code>onDone(NvmStatus)</code> fires once, after both regions are erased                 |

## Required

| Interface                                       | Purpose                                                                 | Contract                                                                                   |
|-------------------------------------------------|-------------------------------------------------------------------------|--------------------------------------------------------------------------------------------|
| <code>NvmRegion</code> (calibration instance)   | Abstracts raw EEPROM write and read for the calibration address range   | Must be backed by a dedicated EEPROM address range; must not be shared with any other user |
| <code>NvmRegion</code> (configuration instance) | Abstracts raw EEPROM write and read for the configuration address range | Must be backed by a dedicated EEPROM address range separate from the calibration range     |

| Field     | Value                                 |
|-----------|---------------------------------------|
| Title     | Service: Command-Line Interface (CLI) |
| Type      | design                                |
| Status    | draft                                 |
| Version   | 0.1.0                                 |
| Component | service-cli                           |
| Date      | 2026-04-07                            |

**IMPORTANT — Implementation-blind document:** This document describes *behavior, structure, and responsibilities* WITHOUT referencing code. **No code blocks using programming languages (C++, C, Python, CMake, shell, etc.) are allowed.** Use Mermaid diagrams to express behavior instead. Prose descriptions of algorithms are encouraged; source-level details are not.

**Diagrams:** All visuals must be either a Mermaid fenced code block (````mermaid`) or ASCII art inline in the document. External image references using Markdown image syntax are **not allowed**.

## Responsibilities

**Is responsible for:** - Providing a human-readable serial command-line interface for commissioning, diagnostics, and runtime control of the motor - Parsing user-typed command strings into

typed arguments (numeric values, enums) using `TerminalHelper`, and returning structured `StatusWithMessage` responses on success or failure - Routing commands to the appropriate control mode interface (`FocTorque`, `FocSpeed`, `FocPosition`) based on which interactor is active - Delegating motor lifecycle and calibration to the `FocStateMachine` when lifecycle commands such as `calibrate`, `enable`, `disable`, `clear_fault`, and `clear_cal` are issued - Printing a welcome banner with version information and available commands when the terminal first connects - Ensuring all string handling uses bounded-size containers — no heap allocation at any point

**Is NOT responsible for:** - Physical serial framing, character echo, or byte-level UART management — those are handled by the `TerminalWithStorage` infrastructure component - Choosing which control mode is active — the active interactor is configured at construction by the application - Executing identification or alignment procedures itself — commands are forwarded to the relevant service; results are printed asynchronously when the service callback fires

## Component Details

### Command Registration

Every command is registered by the control-mode coordinator, which owns the only lifetime-safe reference to the active control mode. There is no per-mode interactor object in production: the active mode lives inside a `std::variant` that is replaced on a mode switch, so an object holding a reference to it could outlive its target. The coordinator instead dispatches each command to whichever mode is active and rejects the command when it does not apply.

### Lifecycle Commands

| Command                      | Alias            | Arguments | Action                                           |
|------------------------------|------------------|-----------|--------------------------------------------------|
| <code>calibrate</code>       | <code>cal</code> | —         | Runs the calibration sequence                    |
| <code>enable</code>          | <code>en</code>  | —         | Enables the motor; rejected outside <b>Ready</b> |
| <code>disable</code>         | <code>dis</code> | —         | Disables the motor                               |
| <code>clear_fault</code>     | <code>cf</code>  | —         | Clears a latched fault                           |
| <code>clear_cal</code>       | <code>cc</code>  | —         | Invalidates stored calibration                   |
| <code>active_mode</code>     | <code>am</code>  | —         | Prints the active control mode                   |
| <code>apply_estimates</code> | <code>ae</code>  | —         | Applies the online estimates to the loop gains   |
| <code>estimate_status</code> | <code>es</code>  | —         | Prints the current online estimates              |

### Algorithm Selection

| Command                               | Alias            | Arguments                                                                                | Action                                                                                       |
|---------------------------------------|------------------|------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------|
| <code>select_current_algorithm</code> | <code>alg</code> | <code>pid</code>   <code>decoupled</code>   <code>deadbeat</code>   <code>sliding</code> | Selects the current-loop law; rejected while enabled or before the motor model is identified |
| <code>select_speed_algorithm</code>   | <code>alg</code> | <code>pid</code>   <code>lqi</code>   <code>adrc</code>   <code>twodof</code>            | Selects the speed-loop law; speed and position modes only                                    |

| Command                                | Alias                                                                             | Arguments | Action                                                                                                     |
|----------------------------------------|-----------------------------------------------------------------------------------|-----------|------------------------------------------------------------------------------------------------------------|
| <code>select_position_algorithm</code> | <code>scascade</code>   <code>lqr</code>   <code>lqi</code>   <code>twodof</code> |           | Selects the position-loop law; position mode only. LQR and LQI are refused when their Riccati design fails |
| <code>active_algorithm</code>          | <code>las</code>                                                                  | —         | Prints the active current, speed and position laws                                                         |

## Setpoints

Each setpoint command applies to exactly one control mode and is rejected with a mode-mismatch message in the others, and rejected when the lifecycle state does not accept setpoints.

| Command                   | Alias           | Arguments     | Accepted range | Mode     |
|---------------------------|-----------------|---------------|----------------|----------|
| <code>set_torque</code>   | <code>st</code> | Iq (A)        | $\pm 100$      | Torque   |
| <code>set_speed</code>    | <code>ss</code> | omega (rad/s) | $\pm 1000$     | Speed    |
| <code>set_position</code> | <code>sp</code> | theta (rad)   | $\pm 2\pi$     | Position |

Ranges come from `foc::CommandLimits`. The CAN path checks speed and position against the same limits, but checks torque against the inverter's `MaxCurrentSupported()`. Either way the torque cascade scales every setpoint onto the `MaxCurrentSupported()` envelope, so neither path can command more current than the inverter supports. A value outside the range checked at entry is rejected and never reaches the control law.

The torque command takes the q-axis current only. The d-axis reference is held at zero by the control law, so exposing it would let an operator command a flux reference the cascade immediately overwrites.

## Bandwidths

| Command                             | Alias             | Arguments         | Accepted range | Availability             |
|-------------------------------------|-------------------|-------------------|----------------|--------------------------|
| <code>set_current_bandwidth</code>  | <code>scbw</code> | bandwidth (rad/s) | 1 ... 20000    | All modes                |
| <code>set_speed_bandwidth</code>    | <code>ssbw</code> | bandwidth (rad/s) | 1 ... 2000     | Speed and position modes |
| <code>set_position_bandwidth</code> | <code>spbw</code> | bandwidth (rad/s) | 1 ... 500      | Position mode            |

Retuning is refused while the motor is enabled, and refused when the active law cannot be redesigned for the requested bandwidth. Bandwidths are bounded below at 1 rad/s: a zero or negative bandwidth produces zero or sign-inverted gains, which inverts the sense of the loop's feedback on a motor that may be spinning.

Both bounds and the enabled-state policy are enforced inside `ControlModeStateMachine::TrySet*Bandwidth` rather than in the command handlers, so the CLI and the CAN bridge cannot drift apart. The

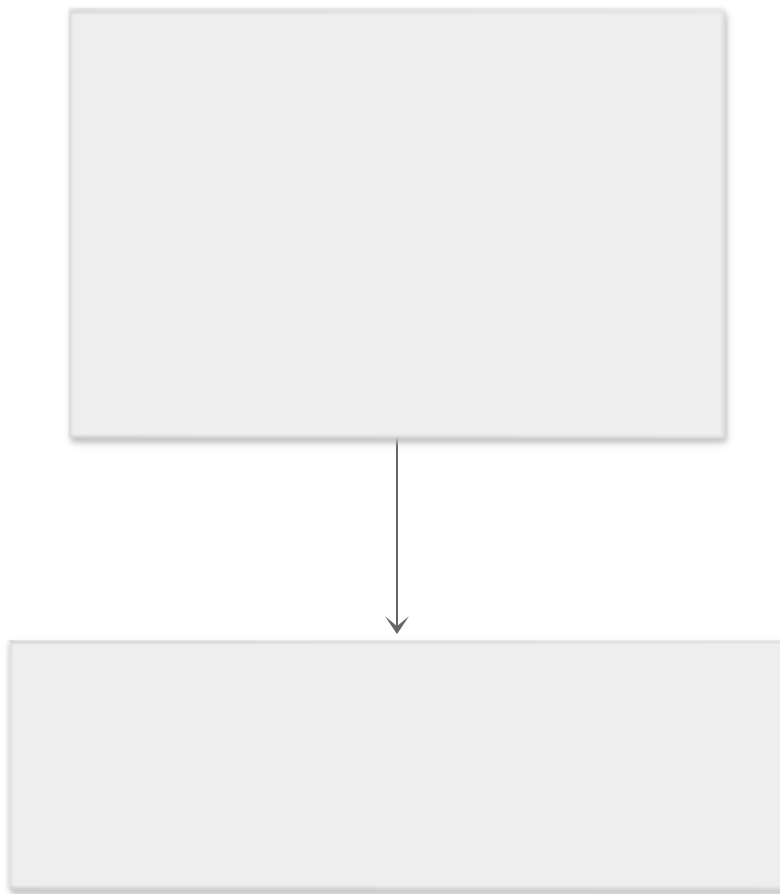


Figure 63: Class diagram

ranges come from `foc::CommandLimits`; the predicates that apply them live in `core/foc/math-/ParameterValidation.hpp`. A rejected value is reported before any controller is touched, so a refused retune leaves the previously accepted bandwidth live.

The same per-loop predicates gate bandwidths arriving from persisted calibration: `CurrentTuningsFor` and `OuterLoopStateMachine::ApplyMechanics` fall back to the compiled default unless the stored value is within the loop's range, so a record written by an older firmware cannot install an out-of-range or non-finite gain.

## Diagnostics Commands

Three commands expose what the control loop and the CAN bus have been doing. Each of these numbers was previously either computed and discarded or never gathered at all.

| Command                  | Alias           | Reports                                                                                                               |
|--------------------------|-----------------|-----------------------------------------------------------------------------------------------------------------------|
| <code>loop_stats</code>  | <code>ls</code> | Control-interrupt execution: sample count, budget, last/min/average/max cycles, overruns, deadline misses, re-entries |
| <code>can_stats</code>   | <code>cs</code> | CAN error counters, total and per class, printing only the classes that have occurred                                 |
| <code>clear_stats</code> | <code>xs</code> | Resets both sets of counters                                                                                          |

**Overflow versus deadline miss.** The two are distinct and both are reported. An overflow means the interrupt exceeded the share of the period it is budgeted — the margin is gone, but the deadline was still met. A deadline miss means it exceeded the period itself: the next conversion was already due when it finished.

**Re-entry** is the same failure caught as it happens rather than measured afterwards. A conversion arriving while the previous interrupt has not returned is recorded directly, which no measured duration can reveal.

Counters saturate rather than wrap, so a long run cannot present itself as a healthy one.

## TerminalWithBanner — Decorator

`TerminalWithBanner` wraps any `TerminalWithStorage` instance and intercepts the first connection event. On first connect, it prints a formatted welcome message containing:

- Firmware version string
- Board/hardware identifier
- A list of available commands (derived from the registered interactor)

After the banner is printed, all subsequent inputs and outputs pass through to the underlying `TerminalWithStorage` unchanged. The banner is printed at most once per connection — reconnecting the serial terminal causes the banner to be printed again.

## TerminalHelper — Argument Parsing

`TerminalHelper` is a stateless utility that is invoked by each command handler to convert bounded-string tokens extracted from the command line into typed values. It handles:

- **Floating-point numbers** — parsed from a bounded-string token; out-of-range or malformed input produces a `StatusWithMessage` with a human-readable description and a non-ok status so that the interactor can return the error directly to the caller without further processing.
- **Enumerations** — matched case-insensitively against a compile-time list of valid string representations; unknown values produce an error with the list of valid options in the message.
- **Integer values** — parsed with range checking against a caller-supplied minimum and maximum.

`TerminalHelper` operates entirely on `infra::BoundedString` tokens — no dynamic string allocation occurs.

## Lifecycle Commands via `FocStateMachine`

The `FocStateMachine` registers additional commands on the same `TerminalWithStorage` instance that enforce the motor lifecycle state machine. These commands are always available regardless of which control mode is active:

| Command                  | Alias            | Action                                                                                                       |
|--------------------------|------------------|--------------------------------------------------------------------------------------------------------------|
| <code>calibrate</code>   | <code>cal</code> | Runs the full calibration sequence (pole pairs → R/L → alignment → mechanical ident for speed modes)         |
| <code>enable</code>      | <code>en</code>  | Enables the FOC controller (only allowed from <code>Ready</code> state)                                      |
| <code>disable</code>     | <code>dis</code> | Disables the FOC controller and returns to <code>Ready</code>                                                |
| <code>clear_fault</code> | <code>cf</code>  | Clears the active fault and returns to <code>Ready</code> (if calibration data is held) or <code>Idle</code> |
| <code>clear_cal</code>   | <code>cc</code>  | Invalidates NVM calibration data and returns to <code>Idle</code>                                            |

## Response Model — `StatusWithMessage`

Every command handler returns a `StatusWithMessage`, which is a pair:

- **Status** — an enumeration value (`ok`, `badArgument`, `notAllowed`, `hardwareError`, ...)
- **Message** — an `infra::BoundedString` containing a human-readable description of the outcome

The terminal writes the message to the serial output verbatim. For asynchronous commands (alignment, identification, NVM), the immediate return to the terminal is `{ok, "Operation started"}` and the final result is written by the service's completion callback.

## Asynchronous Command Flow

For commands that trigger asynchronous services, the interactor initiates the service and returns `{ok, "Operation started"}` immediately. The service's `onDone` callback captures a reference to the `TerminalWithStorage` and writes the final result:

## Memory Constraints

All string handling within the CLI layer uses `infra::BoundedString` with compile-time fixed capacities. Command names, argument tokens, and response messages are all bounded. The maximum response message length is enforced statically.

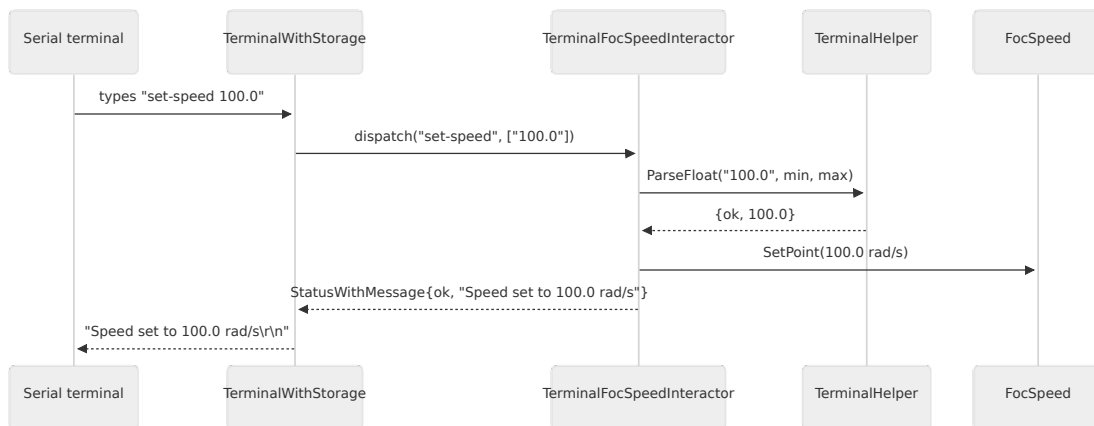


Figure 64: Sequence diagram

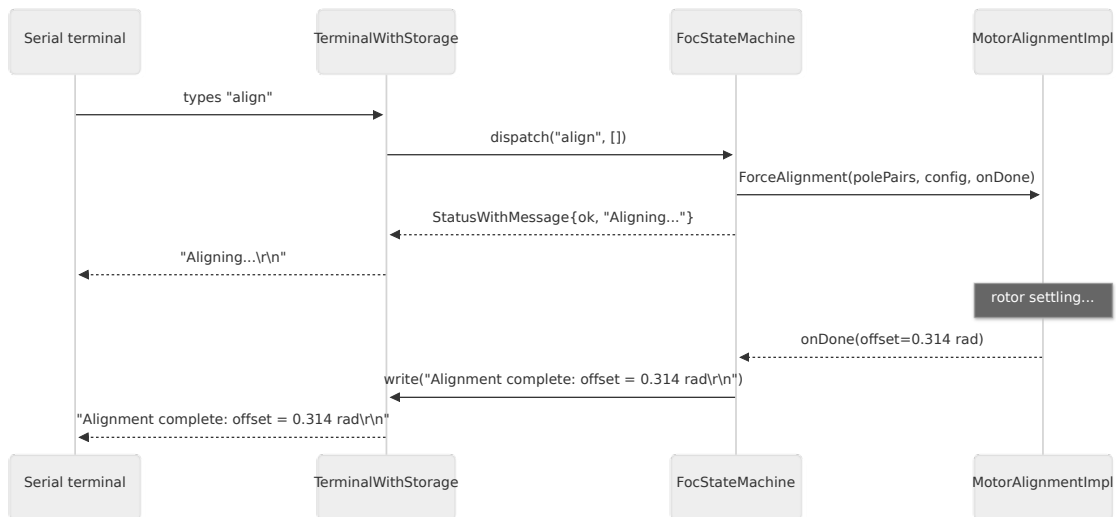


Figure 65: Sequence diagram

Registered command handlers are stored in a fixed-size look-up structure in the **TerminalWithStorage** infrastructure — no dynamic registration table is used.

## Interfaces

### Provided

| Interface                                         | Purpose                                                                                                                                                                                                                | Contract                                                                              |
|---------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------|
| <b>TerminalFocBaseInteractor</b> (and subclasses) | Registers the <code>set_current_bandwidth</code> command on <b>TerminalWithStorage</b> ; exposes a <code>Terminal()</code> accessor for the <b>FocStateMachine</b> to register lifecycle commands on the same terminal | Constructed once per application; exactly one interactor subclass is active at a time |
| <b>TerminalWithBanner</b>                         | Derives <b>TerminalWithStorage</b> to print a welcome banner on first connection                                                                                                                                       | Transparent to the underlying terminal after the banner has been printed              |

### Required

| Interface                                               | Purpose                                                                                                | Contract                                                                             |
|---------------------------------------------------------|--------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------|
| <b>TerminalWithStorage</b>                              | Receives and dispatches parsed command tokens; writes string responses to the serial output            | Must be connected to the physical serial driver before any interactor is constructed |
| <b>CurrentLoopTunables</b>                              | Provides <code>SetCurrentTunings()</code> , shared by all control modes                                | Must remain valid for the lifetime of the interactor                                 |
| <b>FocTorque</b> / <b>FocSpeed</b> / <b>FocPosition</b> | Provides mode-specific setpoint and tuning methods                                                     | The concrete interface must match the constructed interactor subclass                |
| DC bus voltage (Volts)                                  | Supplied to the interactor for normalising PID gain inputs before forwarding them to the FOC component | Must reflect the actual DC bus voltage at the time tunings are applied               |

## Hardware-Test EEPROM Commands

The hardware-test application registers three additional terminal commands that exercise the EEPROM hardware directly through the `hal::Eeprom` interface. These commands are available only in the hardware-test build target and are intended for commissioning, production testing, and EEPROM verification.

### Command Reference

| Command                    | Alias | Arguments                                                                         | Action                                                                                                                                                                                   |
|----------------------------|-------|-----------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>eeeprom_write</code> |       | <code>&lt;addr&gt;</code><br><code>&lt;b0&gt; [b1</code><br><code>... b63]</code> | Writes up to 64 bytes to the EEPROM starting at the given byte address. The callback fires when the hardware write completes.                                                            |
| <code>eeeprom_read</code>  |       | <code>&lt;addr&gt;</code><br><code>&lt;size&gt;</code>                            | Reads <code>size</code> bytes (1–64) from the EEPROM starting at <code>addr</code> and traces each byte value to the serial output. The callback fires when the hardware read completes. |
| <code>eeeprom_erase</code> |       | <code>none</code>                                                                 | Erases the entire EEPROM to the all-0xFF state. The callback fires when the hardware erase completes.                                                                                    |

## Asynchronous Completion

All three commands are asynchronous. The terminal prompt (`>`) does not appear until the hardware operation completes and `ProcessResult` is called from the completion callback. This correctly models the interrupt-driven behaviour of the TM4C EEPROM peripheral.

## Argument Constraints

- `addr`: unsigned integer in the range 0 to 65535
- `b0 ... b63`: unsigned integer byte values in the range 0 to 255
- `size`: unsigned integer in the range 1 to 64

Invalid argument count or out-of-range values return an error `StatusWithMessage` immediately without starting the EEPROM operation.

| Field     | Value                    |
|-----------|--------------------------|
| Title     | CAN Service Layer Design |
| Type      | design                   |
| Status    | draft                    |
| Version   | 0.1.0                    |
| Component | can-service              |
| Date      | 2026-08-21               |

**IMPORTANT — Implementation-blind document:** This document describes *behavior, structure, and responsibilities* WITHOUT referencing code. **No code blocks using programming languages (C++, C, Python, CMake, shell, etc.) are allowed.** Use Mermaid diagrams to express behavior instead. Prose descriptions of algorithms are encouraged; source-level details are not.

**Diagrams:** All visuals must be either a Mermaid fenced code block or ASCII art inline in the document. External image references (`![alt](/home/runner/work/e-foc/e-foc/documentation/design/path)`) are **not allowed**.

## Responsibilities

**Is responsible for:** - Defining the FOC motor CAN category (ID `0x02`) message format — command IDs, response IDs, scale factors, and error codes — in one canonical location. - Implementing the server category: decoding incoming CAN frames into typed observer callbacks and encoding typed results into response frames. - Implementing the client category: encoding typed setpoints into CAN frames and decoding response frames into typed callbacks. - Bridging the server category to `ControlModeStateMachine`, translating motor control commands into state machine transitions. - Providing a user-facing client API that composes the protocol client, transport, and category client into a single object for use by host-side tools and SIL tests. - Applying tracing decorators at every layer boundary so all CAN frames and protocol lifecycle events are observable. - Delegating identification commands to `ElectricalParametersIdentification` and `MechanicalParametersIdentification` services and broadcasting results via response frames. - Persisting encoder resolution and telemetry rate changes to `NonVolatileMemory`. - Broadcasting on-demand telemetry status frames in response to `RequestTelemetry`.

**Is NOT responsible for:** - The CAN protocol framing, sequence validation, or ACK dispatch — those belong to `can-lite` core. - The FOC control algorithms, state machine transitions, or calibration — those belong to `core/foc/` and `core/state_machine/`. - Motor physics or hardware-specific logic.

---

## Component Details

### Part A — FOC Motor Message Catalogue

Message type IDs, the category ID, the enums and the scale factors live in `FocMotorMessages.hpp`; the field layout of every message — order, width, signedness, scale and unit — lives beside them in `FocMotorWireContract.hpp` as a `constexpr` descriptor table. Together these are the authoritative contract for category `0x02`, and no other component may restate any part of them. The tables below are checked against that table by `TestWireContractDocumentation`, so a change to one that is not made to the other fails the build.

The category occupies slot `0x02` (the first application-reserved category ID). Commands occupy message types `0x00–0x7F`; responses occupy `0x80–0xFF`. The category error frame uses the `can-lite` reserved type `0xFE`.

Every command carries the `can-lite` sequence byte ahead of its fields; responses and telemetry do not. A frame whose length is not exactly the length its descriptor states is rejected by its handler and never partially decoded; the `can-lite` protocol server answers a rejected command with `invalidPayload` and does not advance the sequence or client-liveness bookkeeping for it — a payload of another length is a foreign layout, not a tolerable variant of this one. The category-error frame on the `can-lite` reserved type `0xFE` carries a descriptor of its own and is held to the same exactness, so no decode path in this category escapes it.

The descriptors are the specification and the length authority, not a codec: encoders and decoders are written against them by hand. A descriptor edit therefore changes the accepted length and fails the byte-level vectors, which is what surfaces the encode or decode change that has to accompany it.

**Contract version.** `focContractVersionMajor.focContractVersionMinor` identifies the contract, and `QueryContractVersion (0x12)` answers it in `ContractVersionResponse (0x92)`. A client whose major differs from the server’s shall report the mismatch and send no further category-`0x02` commands; equal majors are compatible. Adding a message is a minor bump; changing any field layout, or the meaning of any enumerator, is a major bump.

Physical-to-wire conversions use fixed-point scale factors. `int16` fields round to nearest and saturate (`CanFrameCodec::FloatToFixed16`). Friction and inertia of the reference motors are around  $1e-5$  and  $1e-6$  in SI units, which any useful `int16` scale rounds to zero, so they travel as unsigned 32-bit nano-unit counts:

| Quantity   | Wire type           | Scale factor | Example                                                                      |
|------------|---------------------|--------------|------------------------------------------------------------------------------|
| Current    | <code>int16</code>  | 10           | 1.5 A $\rightarrow$ 15 wire                                                  |
| Speed      | <code>int16</code>  | 1            | 300 rad/s $\rightarrow$ 300 wire                                             |
| Position   | <code>int16</code>  | 100          | 3.14 rad $\rightarrow$ 314 wire                                              |
| Voltage    | <code>int16</code>  | 10           | 24.0 V $\rightarrow$ 240 wire                                                |
| Bandwidth  | <code>int16</code>  | 1            | closed-loop bandwidth in rad/s                                               |
| Resistance | <code>int16</code>  | 1000         | 0.36 $\Omega$ (Teknic M-2310P-LN-04K) $\rightarrow$ 360 wire                 |
| Inductance | <code>int16</code>  | 1000         | 0.20 mH (Teknic M-2310P-LN-04K) $\rightarrow$ 200 wire                       |
| Friction   | <code>uint32</code> | 1000000000   | 1.5e-5 N $\cdot$ m $\cdot$ s/rad (Teknic M-2310P-LN-04K) $\rightarrow$ 15000 |
| Inertia    | <code>uint32</code> | 1000000000   | 7.06e-6 kg $\cdot$ m <sup>2</sup> (Teknic M-2310P-LN-04K) $\rightarrow$ 7060 |

The `uint32` fields round to nearest, saturate at `UINT32_MAX` and encode a negative or NaN value as `0`: friction and inertia are never negative in a physical plant. The reference-motor values come from `documentation/theory/foc-plant-models.md` “Reference motors”.

## Deviations from the can-lite reference example

`infra/can-lite` (pinned at `ea80b3e`) ships `examples/foc_motor`, which also claims category `0x02` and disagrees with this contract. e-foc’s definitions are authoritative for this product; the example is `documentation`. Because every decode is length-exact, a frame built from the example is rejected with `invalidPayload` rather than `misread` — visibly, not silently.

| Message / item              | e-foc                                       | can-lite example                     | Why                                                                                                     |
|-----------------------------|---------------------------------------------|--------------------------------------|---------------------------------------------------------------------------------------------------------|
| <code>0x03/0x04/0x05</code> | <code>int16</code><br>closed-loop bandwidth | three <code>int16</code><br>kp/ki/kd | <code>documentation/requirements/foc/*-controller.yaml</code> forbid externally supplied per-axis gains |
| <code>0x0A</code>           | <code>Align</code>                          | reserved, unused                     | alignment is a first-class command here                                                                 |
| <code>FocMotorState</code>  | states partialCalibration = 4               | stops at calibrating                 | speed and position modes can hold a partial record                                                      |
| <code>0x12 / 0x92</code>    | contract version query and response         | absent                               | the example has no versioning                                                                           |

## Part B — FocMotorCategoryServer

Inherits `CanCategoryServer` from `can-lite`. Registers one `CanMessageHandler` per command type; each handler decodes the payload using `CanPayloadReader`, skips the sequence byte (required for server categories), converts wire values to physical units, and calls the corresponding observer method.

The observer interface provides callbacks for: - `OnStart`, `OnStop`, `OnClearFault`, `OnEmergencyStop` — lifecycle commands with a `CanAckStatus` result callback. - `OnSelectControlMode` — takes a `FocMotorMode` and a result callback. - `OnSetTorqueSetpoint`, `OnSetSpeedSetpoint`, `OnSetPositionSetpoint` — take a typed unit quantity and a completion callback. - `OnSetPidCurrent`, `OnSetPidSpeed`, `OnSetPidPosition` — receive a bandwidth parameter parsed from the CAN frame and forward to the corresponding `TrySet*Bandwidth` on `ControlModeStateMachine`. - `OnIdentifyElectrical` — calls `CmdReserveExternalCalibration()` before starting estimation; `invalidState` if rejected; on estimation success calls `CmdCompleteExternalCalibration()` and broadcasts `focElectricalParamsResponseId`. - `OnIdentifyMechanical` — calls `ActiveCalibrationData()` to obtain pole pairs and guard the state in one step; `invalidState` if not in `Ready`; on success broadcasts `focMechanicalParamsResponseId`. - `OnRequestTelemetry` — broadcasts current state, fault code, measured speed and measured position via `focTelemetryStatusResponseId`. Position is read from the encoder; speed is the active mode's `ObserveMotion().measuredSpeed` — the outer loop's window speed in speed and position mode, the interrupt's filtered speed in torque mode (REQ-INT-014). - `OnSetEncoderResolution`, `OnConfigureTelemetryRate` — validate payload, update and persist `ConfigData` via `NonVolatileMemory`.

## Part C — FocMotorCategoryClient

Inherits `CanCategoryClient` from `can-lite`. Provides typed `Send*` methods that encode physical units into wire format via `CanPayloadWriter` and call `SendCommand(nodeId, messageType, payload)`.

Registers response handlers for the server-to-client response message types. Response handlers decode the payload and invoke the corresponding observer callbacks.

## Part D — FocMotorCanBridge

Implements the `FocMotorCategoryServerObserver` interface. Holds references to `FocMotorCategoryServer` and `ControlModeStateMachine`. Translates each observer callback into the appropriate state machine call:

- `OnStart` → `CmdEnable()`; success calls the ack callback, rejection sends `invalidState`.
- `OnStop` → `CmdDisable()`.
- `OnClearFault` → `CmdClearFault()`.
- `OnEmergencyStop` → `CmdEmergencyStop()`.
- `OnSelectControlMode` → `Select(mode, onDone)`; the result callback sends the mode response or a category error.
- `OnSetTorqueSetpoint`, `OnSetSpeedSetpoint`, `OnSetPositionSetpoint` → validate mode and range, then delegate to `TrySet*` on the state machine.
- PID bandwidth commands → `TrySet*Bandwidth(bandwidth)` on the state machine. The state machine applies the same `foc::CommandLimits` ranges and the same enabled-state policy the CLI gets: a value outside the range for that loop, or one that does not apply to the active

mode, is acknowledged `invalidPayload`; a redesign refused because the drive is running is acknowledged `invalidState`.

- `OnIdentifyElectrical` → `CmdReserveExternalCalibration()` (state guard + `Calibrating` entry); run estimation; `CmdCompleteExternalCalibration(data, cb)` on success, which aligns the rotor and persists; `invalidState` / `calibrationFailed` / `busy` on failure. The command covers the electrical steps only — it never drives the mechanical estimator — so speed and position end in `Idle` reporting `partialCalibration`, while torque reaches `Ready`.
- `OnIdentifyMechanical` → `ActiveCalibrationData()` (state guard + pole-pairs extraction in one call); run estimation; broadcast on success; `invalidState` / `calibrationFailed` / `busy` on failure.
- `OnRequestTelemetry` → broadcast current state, fault code, measured speed and measured position via `focTelemetryStatusResponseId`; always succeeds. `ToCanMotorState` maps `Idle` to `FocMotorState::partialCalibration` instead of `idle` when `HasPartialCalibration()` reports a non-empty but incomplete NVM record (old-schema or interrupted external calibration).
- `OnSetEncoderResolution` / `OnConfigureTelemetryRate` → persist to `NonVolatileMemory`; `persistenceFailed` on write error; `busy` if a prior NVM save is in flight.

The bridge validates that the correct control mode is active before accepting a setpoint command. An out-of-range setpoint results in `invalidPayload`. A mode mismatch results in `categoryError/modeMismatch`.

## Part E — `FocMotorCanClient`

A self-contained client object for use in host tools and SIL tests. Internally composes `CanFrameTransport`, `CanProtocolClient`, and `FocMotorCategoryClient` as value members (no heap). Constructed with a `hal::Can&` reference and a target node ID.

Provides the same `Send*` API as `FocMotorCategoryClient` but hides the transport and protocol client, so callers only need a CAN bus reference and a node ID.

## Part F — `CanLivenessWatchdog`

`CanProtocolServer` already tracks client liveness: it restarts a timer on every frame from the client and calls `Offline()` on its observers when `clientTimeout` elapses with no traffic. Nothing acted on that notification, so unplugging the bus mid-spin left the drive holding its last setpoint indefinitely.

`CanLivenessWatchdog` is a `CanProtocolServerObserver` that closes the loop. On `Offline()` it checks whether the active state machine is in `Enabled`; if so it traces and issues `CmdEmergencyStop`, which stops the inverter. When the drive is not enabled the notification is ignored — a bus that goes quiet while the motor is stopped is not a hazard, and faulting there would obstruct commissioning.

It is composed in the instantiation layer alongside the CAN bridge, so the bridge and category server stay free of lifecycle policy.

## Part G — Fault Broadcast Priority

`BroadcastFaultStatus` sends its frame through `SendEmergency`, at `CanPriority::emergency` (0), rather than through `SendTelemetry`, which uses `CanPriority::telemetry` (12). CAN arbitration is by identifier, and the priority field is the high bits of the identifier, so a telemetry-priority fault

notification loses arbitration to routine telemetry on a busy bus — exactly when the bus is busiest and the notification matters most. The payload is unchanged.

Arbitration only helps a frame that reaches the controller. The `can-lite` send queue therefore orders queued frames by priority, keeps two of its slots for emergency frames only, and lets an emergency frame displace the newest lowest-priority frame when it is full, so a fault notification is not refused behind queued telemetry.

## Part H — Tracing Decorators

All tracing is applied via `can-lite`'s decorator classes:

| Decorator                                     | Wraps                          | Traces                                       |
|-----------------------------------------------|--------------------------------|----------------------------------------------|
| <code>TracingCan</code>                       | <code>hal::Can</code>          | Every raw CAN frame sent and received        |
| <code>TracingCanProtocolServerObserver</code> | <code>CanProtocolServer</code> | Server online/offline events                 |
| <code>TracingCanProtocolClientObserver</code> | <code>CanProtocolClient</code> | Server online/offline, ACK timeouts per node |

These are composed in the instantiation layer (`Logic.hpp`) so the production code paths gain full observability without any tracing logic in the category or bridge classes themselves.

## Interfaces

### Provided

| Interface                                   | Purpose                                                                                                                                         | Contract                                                                                  |
|---------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------|
| <code>FocMotorCategoryObserver</code>       | Observer interface for the server category                                                                                                      | Callbacks must not block or allocate; called from the event dispatcher context            |
| <code>FocMotorCategoryClientObserver</code> | Observer interface for the client category                                                                                                      | Callbacks must not block or allocate                                                      |
| <code>FocMotorCanClient</code> public API   | User-facing send methods ( <code>Start</code> , <code>Stop</code> , <code>SetTorque</code> , <code>SetSpeed</code> , <code>SetPosition</code> ) | Returns <code>bool</code> indicating whether the frame was queued; ACK is delivered async |

### Required

| Interface                            | Purpose                           | Contract                                                                            |
|--------------------------------------|-----------------------------------|-------------------------------------------------------------------------------------|
| <code>CanFrameTransport</code>       | Sends and receives raw CAN frames | Provided by <code>can-lite</code> ; must be constructed before the category objects |
| <code>ControlModeStateMachine</code> | Dispatches motor control commands | All methods are synchronous or deliver results via <code>infra::Function</code>     |

| Interface                     | Purpose            | Contract                                                                      |
|-------------------------------|--------------------|-------------------------------------------------------------------------------|
| <code>hal::Can</code>         | Raw CAN bus access | Required by <code>CanProtocolServer</code> and <code>CanProtocolClient</code> |
| <code>services::Tracer</code> | Diagnostic output  | Must outlive the tracing decorator objects                                    |

## Data Model

| Entity                | Field    | Type / Unit   | Range                            | Notes                                    |
|-----------------------|----------|---------------|----------------------------------|------------------------------------------|
| FocMotorMode          | mode     | enum<br>uint8 | torque=0, speed=1,<br>position=2 | Transmitted as single byte               |
| Torque setpoint       | iq       | int16 wire    | INT16_MIN –<br>INT16_MAX         | Physical = wire / 10 [A]                 |
| Speed setpoint        | speed    | int16 wire    | INT16_MIN –<br>INT16_MAX         | Physical = wire / 1<br>[rad/s]           |
| Position setpoint     | position | int16 wire    | INT16_MIN –<br>INT16_MAX         | Physical = wire / 100<br>[rad]           |
| FocMotorCategoryError | code     | enum<br>uint8 | 0–5                              | See<br><code>FocMotorMessages.hpp</code> |

## Sequence Diagrams

### Start Command (server path)

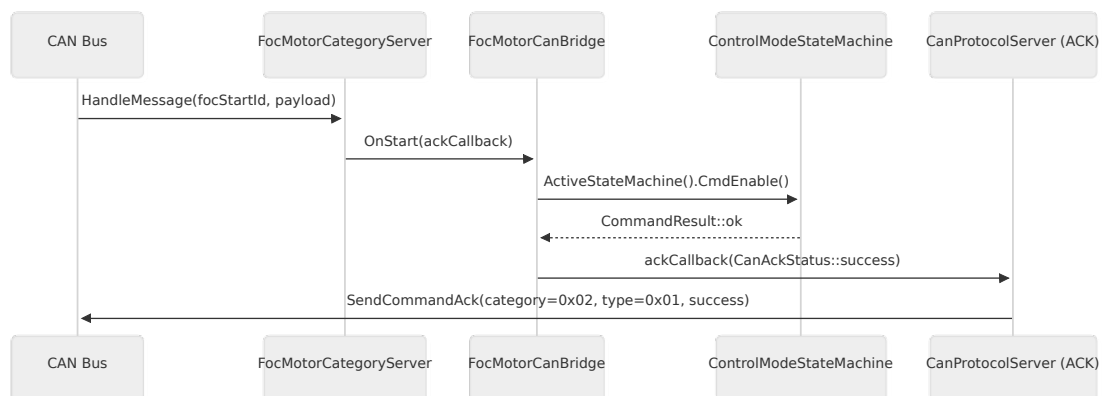


Figure 66: Sequence diagram

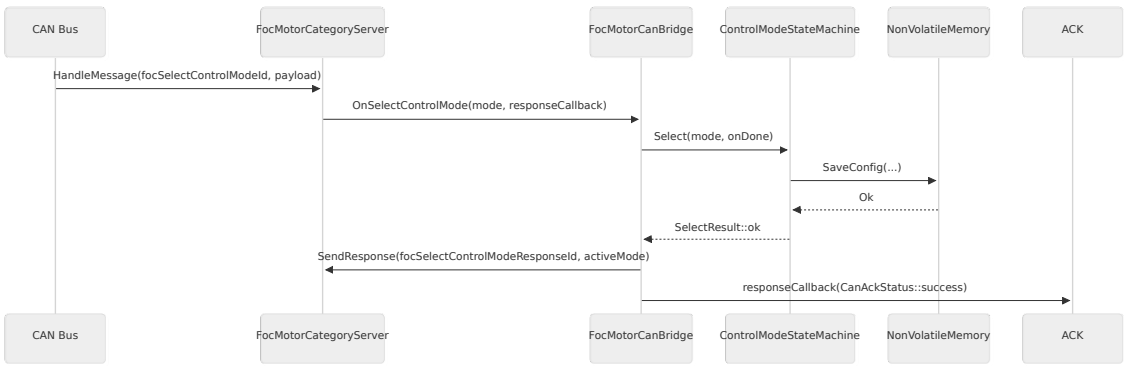


Figure 67: Sequence diagram

SelectControlMode (server path)

Bandwidth Rejection Path



Figure 68: Sequence diagram

outOfRange and wrongMode map to invalidPayload; notWhileEnabled maps to invalidState, so a client can tell a badly formed request from one that was well formed but arrived while the drive was running.

Block Diagram

Constraints & Limitations

| Constraint | Value / Description                                                                               |
|------------|---------------------------------------------------------------------------------------------------|
| No heap    | All objects are value members or statically allocated; <code>infra::Function</code> for callbacks |

| Constraint              | Value / Description                                                                                                                     |
|-------------------------|-----------------------------------------------------------------------------------------------------------------------------------------|
| One observer per server | <code>CanCategoryServer</code> uses <code>infra::Subject&lt;Observer&gt;</code> — only one bridge may attach at a time                  |
| Setpoint range          | Torque: bounded by inverter <code>MaxCurrentSupported</code> ; speed: 1000 rad/s; position: $\pm 2\pi$ rad                              |
| Sequence byte           | Server handlers always skip the first byte (sequence number) before reading payload fields                                              |
| Ident re-entrancy       | A second identification command while one is in-flight returns <b>busy</b> via <code>SendCategoryError</code>                           |
| NVM re-entrancy         | A second config-persist command while one is in-flight returns <b>busy</b> via <code>SendCategoryError</code>                           |
| Mechanical ident        | <code>mechIdent</code> is optional (nullable pointer); if absent, <code>OnIdentifyMechanical</code> returns <code>notImplemented</code> |
| Telemetry speed/pos     | Speed is the active mode's <code>ObserveMotion().measuredSpeed</code> , zero while disabled; position is the encoder                    |

## Open Questions

| # | Question                                                         | Resolution                                                                                                                                         | Status   |
|---|------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------|----------|
| 1 | Implement telemetry push (BroadcastFaultStatus, periodic status) | On-demand via <code>OnRequestTelemetry</code> ; periodic timer deferred                                                                            | resolved |
| 2 | PID bandwidth commands                                           | Accepted: single bandwidth value per command decoded by server, forwarded to <code>TrySet*Bandwidth</code> on <code>ControlModeStateMachine</code> | resolved |
| 3 | Mechanical identification via CAN                                | Delegated to injected <code>MechanicalParametersIdentification*</code> ; nullable for targets lacking it                                           | resolved |

| Field     | Value                 |
|-----------|-----------------------|
| Title     | Error Handling Design |
| Type      | design                |
| Status    | stable                |
| Version   | 1.0.0                 |
| Component | error-handling        |
| Date      | 2026-04-13            |

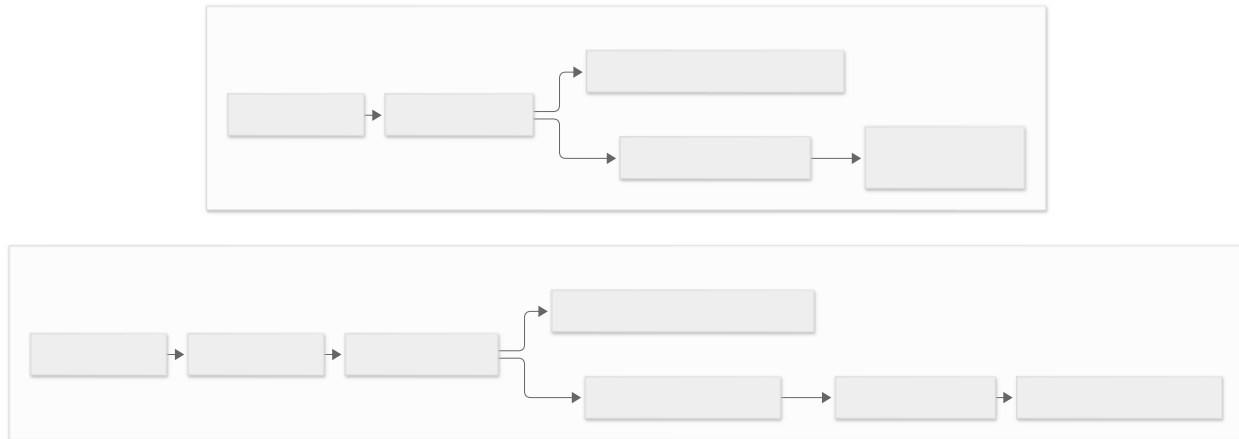


Figure 69: Block diagram

## Responsibilities

**Is responsible for:** - Capturing Cortex-M CPU register state and a stack-address backtrace when a HardFault exception occurs, before the system resets. - Persisting that diagnostic snapshot in a memory region that is not cleared on reset, so it survives the subsequent boot. - Reporting the cause of the most recent reset (power-up, brown-out, software, hardware, or watchdog) to the application layer via `PlatformFactory`. - Exposing the formatted fault snapshot and reset cause to the CLI at boot time. - Providing a platform-initiated software reset via `PlatformFactory::Reset()`.

**Is NOT responsible for:** - Deciding what the application does after a fault (handled by the application layer). - Supervising the event dispatcher with the MCU watchdog — that is the watchdog component’s work (Watchdog Design (Chapter 35)). This component only reports a watchdog reset once it has happened. - Correcting or masking the fault condition. - Logging to persistent storage beyond the single 1 KiB fault region. - CAN-bus boot notification or any network-layer diagnostics.

## Component Details

### Fault capture — HardFault handler

A naked assembly trampoline runs in HardFault exception context before any C function prologue executes. It determines whether the exception was entered from thread mode (using PSP as the stack pointer) or from handler mode (using MSP), then passes the correct stack frame pointer to the C-level capture routine.

The capture routine’s **first** action is to call `CutPowerStage()`. Everything after it — the register capture, the stack scan, the reset — takes time during which six gate drivers otherwise hold whatever state they had when the fault hit.

`CutPowerStage()` is declared by the handler and **defined by each platform**; it is deliberately not a weak fallback. A weak definition in the handler’s own translation unit is the one the linker settles on, because the archive member carrying the real definition is never pulled in — nothing

references a symbol only it defines. The result is a hook that is called on every fault and cuts nothing. Requiring the definition means a platform that forgets it fails to link.

On TI, the definition clears `PWMENABLE` on each PWM module that `SYCTL->PRPWM` reports as clocked, which drives every `pwmN` output low and so turns all six gate signals off. It is a handful of instructions with no loop and no dependency on driver state, and it touches no unlocked peripheral — a write to one would fault again, and a nested fault here is the lockup this call exists to prevent.

On ST the definition is empty, and says so: that target does not drive a bridge yet — both `PinsAndPeripherals.hpp` are empty and the encoder reports a resolution of zero — so there is no timer instance to name. Cutting the stage there means clearing `TIM_BDTR_MOE` on whichever advanced-control timer the board wires to the gate drivers, and that belongs with the work that makes the target functional.

## Board protection state — `BoardProtectionStatus()`

`PlatformFactory::BoardProtectionStatus()` answers `asserted`, `clear` or `unknown` for the board-protection condition, and is what lets the state machine refuse to re-arm a drive whose hardware is still faulted (REQ-SM-025, REQ-SM-026). It is a dispatcher-context query — an implementation may read a peripheral status register, but nothing calls it from an interrupt, so it need not be interrupt-safe.

`unknown` is a first-class answer, not a failure. A platform reports it whenever it cannot observe the condition, and the state machine then falls back to the retry budget alone — which is the behaviour those platforms had before the query existed. Reporting `clear` instead would claim a verification the hardware never performed, and on a board with no protection comparators the fault being cleared came from software anyway.

Every platform reports `unknown` today:

- **TI** — the protection comparators are ADC digital comparators (`DigitalComparatorConfig`, steps 3 and 4) whose outputs drive the PWM fault inputs. Their counts come from the motor board's `BoardCharacteristics`; the total-current sense is biased to mid-rail, so `OvercurrentThresholdCounts` is an offset above half scale (6.6 A, 80 % of the +8.25 A a 5 A/V sense spans on a 3.3 V ADC). That ADC is triggered by the PWM generator, so once the bridge is stopped it stops converting, the comparators stop evaluating, and no status register distinguishes a cleared condition from an unevaluated one. Answering this honestly needs either a sampling path that survives the bridge stopping — `PowerSupplyVoltage()` already runs on a separately triggered ADC and could verify overvoltage against `BoardCharacteristics::OvervoltageThresholdCounts` — or a software/continuous ADC trigger in `hal-ti` so the comparators keep evaluating. Overcurrent is arguably not observable at all while stopped: no current flows through a disabled bridge.
- **ST, QEMU** — neither wires board protection, so there is no condition to observe.
- **host** — settable, so tests can hold the condition asserted across a re-arm attempt.

The capture routine then writes all eight words of the ARM exception stack frame (R0–R3, R12, LR, PC, xPSR) plus three Cortex-M fault status registers (CFSR, MMFAR, BFAR) into the persistent region. It also scans the stack above the exception frame and records any word whose value falls within the `.text` section as a probable return address — this constitutes the backtrace. The scan is bounded to a fixed number of words, not run to `_estack`: on a bus or MPU fault the stack pointer itself may be the corrupt value, and dereferencing it can nest-fault into lockup, which leaves the

motor energised indefinitely because the reset never happens. A validity sentinel is written last; this is the atomic commit point. If the MCU loses power or watchdog-resets mid-write, an incomplete capture is detected and discarded on the next boot.

After capture, the handler calls `abort()`, which the platform startup maps to `NVIC_SystemReset()`.

## Persistent fault region

The fault data structure (`PersistentFaultData`) is placed in a dedicated `.error_handling` linker section that is marked `NOLOAD`. The linker scripts for both TI and ST targets locate this section inside RAM but exclude it from the BSS zero-fill at startup, so its contents survive across a warm reset. The region is exactly 1 KiB and the linker asserts this at link time.

## Reset cause detection

Each platform's `PlatformFactoryImpl` reads the hardware-specific reset status register during construction, before anything else modifies it. It converts the raw vendor bits to the platform-agnostic `ResetCause` enumeration and clears the register so that subsequent resets are not misreported. The `ResetCause` value is then accessible for the lifetime of the application via `PlatformFactory::GetResetCause()`.

## Boot-time CLI reporting

The CLI banner reads both the reset cause and any pending fault snapshot from `PlatformFactory` during construction. If a valid snapshot exists, it is formatted into a human-readable string and the validity sentinel is cleared immediately after formatting, so the snapshot is consumed at most once. The formatted string is retained for the lifetime of the application for later `fault_status` CLI queries. The banner prints the reset cause unconditionally and surrounds fault data with visible delimiters.

---

## Control-loop and bus statistics

Alongside the post-mortem fault record, the platform keeps two live sets of counters, both readable from the CLI (`loop_stats`, `can_stats`, `clear_stats`).

**Control-interrupt execution.** The Cortex-M cycle counter is left free-running and read on entry to and exit from the application callback that executes inside the control interrupt; the difference is one measured execution. Unsigned subtraction makes the measurement correct across the counter's own wrap, which at 120 MHz comes around every 36 seconds. The measured window covers encoder read, control law and PWM write — the work the cycle budget applies to. Hardware ISR entry/exit overhead is excluded; REQ-PERF-001 reserves 25% of the control period for that overhead. The measurement is taken in the platform layer so it is consistent across targets. The emulated target has no cycle counter and reads a free-running timer of its instruction-counted clock instead ([software-in-the-loop.md](#), Part B3).

From it the platform maintains the last, minimum, maximum and a filtered average duration, the execution count, and three failure counts: **overruns** (over the budget, margin gone, deadline still met), **deadline misses** (over the period itself), and **re-entries** (a conversion arriving while the

previous interrupt had not returned — the same failure observed as it happens rather than inferred afterwards).

**CAN errors.** Every error class the adapter reports is counted, not only the bus-off condition that raises a fault. Counting happens on the adapter base class, so no adapter’s reporting path can omit it. This is what separates a bus with marginal termination from one that is unplugged.

Both sets saturate rather than wrap. The cost on the control path is a pair of register reads and about two dozen inlined instructions, under one percent of the 20 kHz period.

## Interfaces

### Provided

| Interface                                     | Purpose                                        | Contract                                                                  |
|-----------------------------------------------|------------------------------------------------|---------------------------------------------------------------------------|
| <code>PlatformFactory::Reset()</code>         | Trigger an immediate software reset            | Called synchronously; does not return                                     |
| <code>PlatformFactory::GetResetCause()</code> | Reset cause captured at boot                   | Valid for the lifetime of the application; thread-safe by value semantics |
| <code>PlatformFactory::FaultStatus()</code>   | The formatted fault string (empty if no fault) | Valid for the lifetime of the application once the constructor returns    |

### Required

| Interface                                                                                     | Purpose                                                    | Contract                                                         |
|-----------------------------------------------------------------------------------------------|------------------------------------------------------------|------------------------------------------------------------------|
| Platform linker script                                                                        | Provide <code>.error_handling</code> NOLOAD section in RAM | Must be placed before <code>.bss</code> to avoid zero-fill       |
| ARM Cortex-M NVIC                                                                             | Deliver HardFault exception to the handler trampoline      | Always-enabled; not configurable via <code>NVIC_EnableIRQ</code> |
| <code>SYSCTL-&gt;RESC</code> (TI) / <code>RCC-&gt;CSR</code> or <code>RCC-&gt;RSR</code> (ST) | Provide reset source bits at boot                          | Read once; cleared after read                                    |

## Data Model

| Entity              | Field                             | Type / Unit     | Range           | Notes                           |
|---------------------|-----------------------------------|-----------------|-----------------|---------------------------------|
| PersistentFaultData | <code>data</code>                 | 32-bit unsigned | 0 or 0xDEADBEEF | Written last to commit capture  |
| PersistentFaultData | <code>r0, r12, lr, pc, psr</code> | 32-bit unsigned | Any             | ARM exception stack frame words |

| Entity              | Field           | Type / Unit           | Range                                           | Notes                                              |
|---------------------|-----------------|-----------------------|-------------------------------------------------|----------------------------------------------------|
| PersistentFaultData | mmfar, bfar     | 32-bit unsigned       | Any                                             | Cortex-M fault status registers                    |
| PersistentFaultData | StackTrace[243] | 32-bit unsigned array | Any                                             | Addresses within <code>.text</code> found on stack |
| PersistentFaultData | TraceCount      | 32-bit unsigned       | 0–243                                           | Number of valid trace entries                      |
| ResetCause          | —               | enum                  | powerUp, brownOut, software, hardware, watchdog | MCU-agnostic                                       |

`ResetCause::watchdog` comes from the reset source bits alone. The watchdog never resets by software: on expiry it cuts the power stage and stops refreshing, and the peripheral’s own reset sets the watchdog bit. No marker has to survive the reset on an MCU; the emulated target, which has no reset source bits, keeps a semihosting marker file instead. See Watchdog Design (Chapter 35).

---

## State Machine

---

## Sequence Diagrams

Boot with prior fault

HardFault capture

| Field     | Value           |
|-----------|-----------------|
| Title     | Watchdog Design |
| Type      | design          |
| Status    | draft           |
| Version   | 0.3.0           |
| Component | watchdog        |
| Date      | 2026-09-26      |

---

## Responsibilities

**Is responsible for:** - Proving that the software is running. Every MCU platform supervises its event dispatcher with the MCU watchdog peripheral from the moment the dispatcher is constructed until the next reset. - Bringing the power stage to a safe state from interrupt context when the

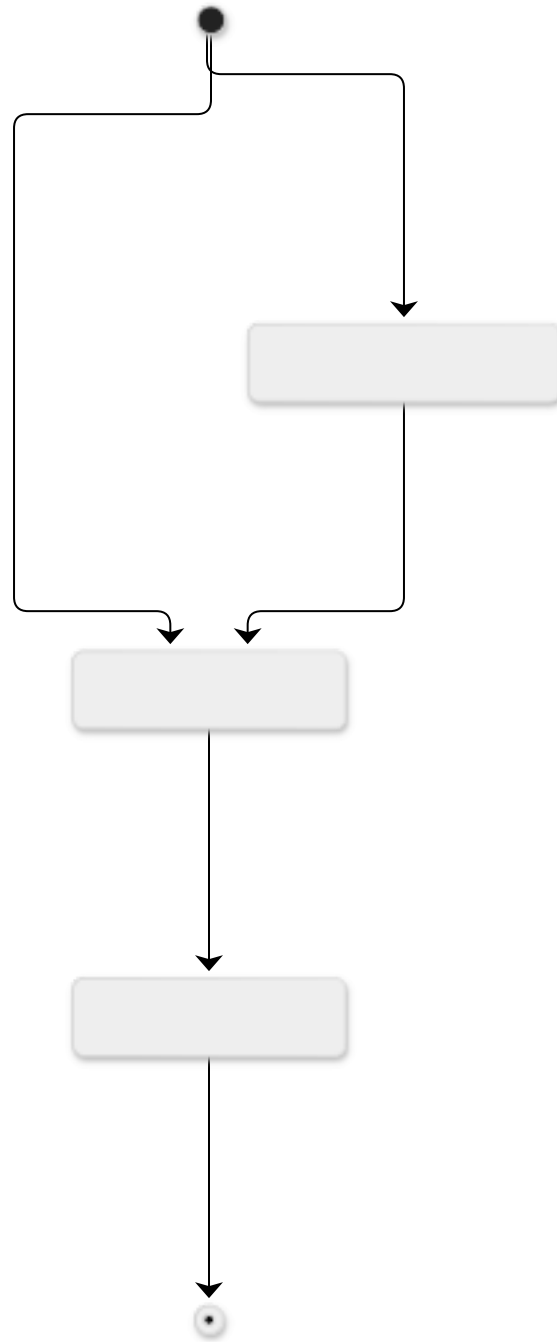


Figure 70: State diagram

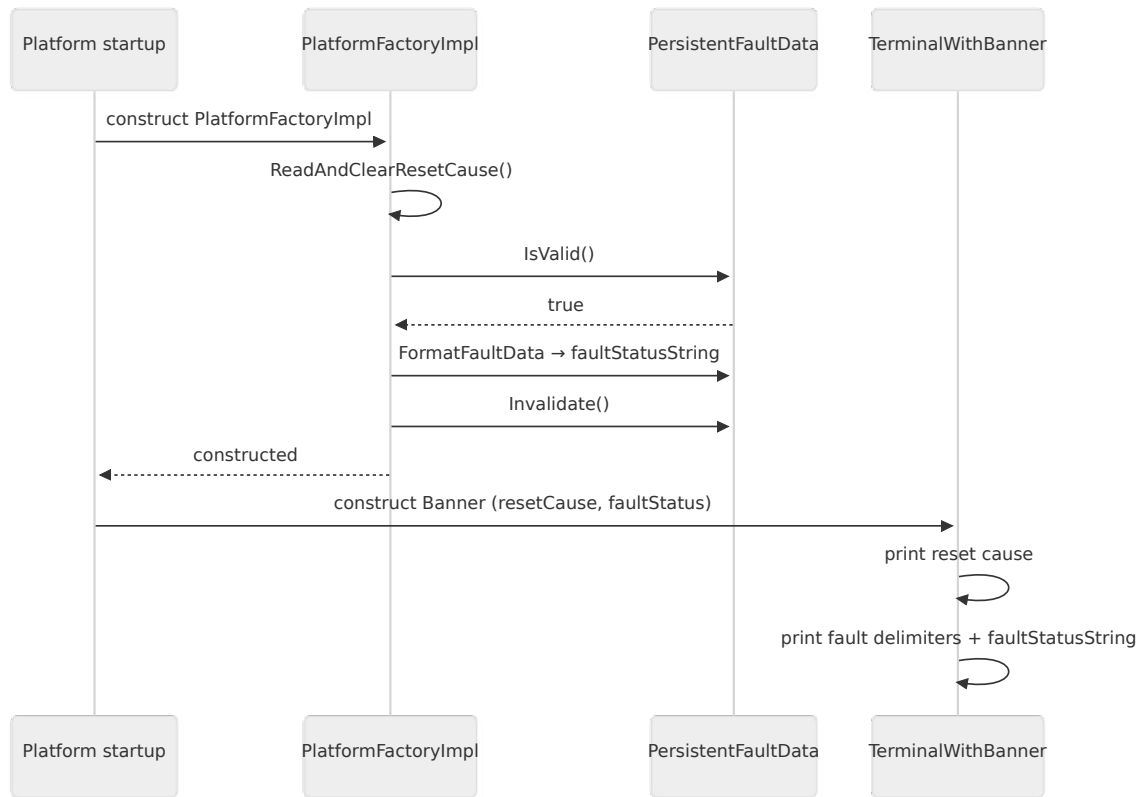


Figure 71: Sequence diagram

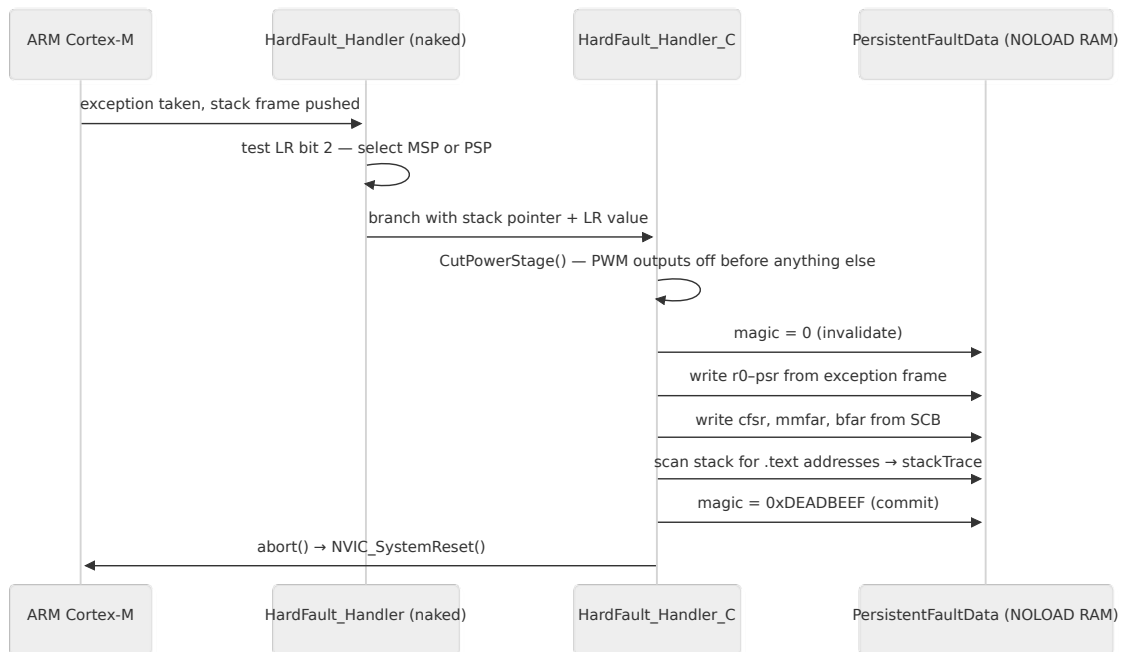


Figure 72: Sequence diagram

dispatcher has stopped making progress, before the hardware resets the target. - Resetting the target from hardware, so that the next boot reports the reset cause as *watchdog*.

**Is NOT responsible for:** - Proving that the control loops are running. A current loop or outer loop that stops while the dispatcher keeps running is not a watchdog failure. It is left to the protections that watch the bridge itself: the hardware overcurrent and overvoltage comparators, board protection and the fault state machine. - Being switched on, off or reconfigured by the application. There is no watchdog port in the platform factory. The platform owns the watchdog, and supervision is always on. - Detecting the reset cause after a watchdog reset. That is the error-handling component's work, reached through the reset-cause port.

---

## Component Details

### Part A — supervision lives in the event dispatcher

The mechanism is EMIL's, not e-foc's. EMIL provides: - a hardware watchdog interface, `hal::Watchdog`, implemented by the ST, TI and emulated-target drivers; - an event dispatcher worker that supervises it, `services::EventDispatcherWatchdogWorker`.

Each MCU platform builds its dispatcher from that worker and hands it the platform's watchdog. The dispatcher then does three things:

- **Counts progress.** The worker counts one step when an action starts and one when it finishes, so an odd count means an action is executing.
- **Refreshes on every early warning.** The hardware raises an early-warning interrupt once per early-warning period. The handler always refreshes the hardware.
- **Counts missed early warnings.** An early warning counts as missed when an action is executing and the count has not changed since the previous early warning. Early warnings that find the dispatcher idle or moving do not count, and they reset the tally.

After enough consecutive misses to cover the expiration timeout, the worker calls the platform's expiry handler once and stops refreshing. The hardware then resets the target at its next timeout.

Supervision is always on. It starts in the dispatcher's constructor, and nothing can disable it. The application needs no feed, no health check and no startup grace.

### Part B — what happens on expiry

The expiry handler runs inside the watchdog's early-warning interrupt. It runs there *because* the event loop is stuck, so nothing can be scheduled and no application code can run.

The handler does only what is safe in an interrupt and needs no driver state: 1. It calls the direct power-stage cutoff, which the hard-fault handler also uses. On TI this clears the PWM output enables, so every gate is driven inactive and the motor coasts. 2. It returns.

The handler does not reset by software. The watchdog is no longer refreshed, so the hardware resets the target one early-warning period later. That reset shows as *watchdog* in the MCU's own reset-cause register, with no record to write before the reset.

This is the most graceful stop that is possible at this point. A torque ramp-down would need the event loop that has just been proven stuck, and the time left before the hardware reset is only one early-warning period.

## Part C — interrupt priority of the early warning

Progress is counted only inside actions. A dispatcher that is idle, or frozen between actions because an interrupt never returns, looks like it is making progress. Because of that, the early-warning interrupt must not be able to preempt any interrupt that could hang:

- The early-warning interrupt runs at the **lowest** priority the NVIC implements, no higher than the outer loop's PendSV.
- A hung interrupt at any priority then starves the early warning, so the hardware resets the target without the handler running.
- A hung action in the event loop leaves the early warning free to run, so the handler cuts the power stage first and the hardware resets the target afterwards.

| Failure                                               | Early warning runs? | Outcome                                    |
|-------------------------------------------------------|---------------------|--------------------------------------------|
| An event-loop action never returns                    | yes                 | Power stage cut, then hardware reset       |
| An interrupt never returns (control, outer loop, ...) | no, starved         | Hardware reset; pins return to reset state |
| Interrupts disabled / CPU lockup                      | no                  | Hardware reset; pins return to reset state |
| The control loop stops while the event loop runs      | yes, sees progress  | Not detected, by design                    |

## Part D — platforms

| Platform         | Hardware                            | Early-warning period                        | Expiration timeout            |
|------------------|-------------------------------------|---------------------------------------------|-------------------------------|
| TI (TM4C123/129) | Watchdog 0, reset on second timeout | 25 ms                                       | 100 ms                        |
| ST (STM32)       | Window watchdog, prescaler 1        | Fixed by PCLK1 ( $\approx 16$ ms at 16 MHz) | 100 ms (seven early warnings) |
| Emulated target  | CMSDK watchdog of the MPS2 machine  | 25 ms                                       | 100 ms                        |
| Host             | none                                | —                                           | —                             |

- **ST:** the vendor driver pins the window watchdog to the highest priority when it starts. The platform lowers it to the lowest NVIC level straight after the dispatcher is constructed, for the reason given in Part C. That level is set with the CMSIS encoding, because EMIL's priority enum assumes three priority bits and the STM32 parts implement four. The cutoff is empty while the ST platform drives no bridge.

The window watchdog resets one counter tick after its early warning ( $4096 \cdot \text{prescaler} / \text{PCLK1}$ , 256  $\mu\text{s}$  at 16 MHz with prescaler 1), so the early-warning handler must run within

that tick. Prescaler 1 keeps the early-warning period short, and with it the rounding of the timeout; a larger prescaler buys handler latency at the cost of a later expiry.

- **Emulated target:** the MPS2 machine wires its watchdog to NMI, which cannot be lowered. So on the emulated target a hung interrupt is caught only when the event loop is inside an action. That is acceptable for a target that drives a simulated plant.

The machine has no reset-cause register, and its reset reloads RAM. So the expiry handler writes a marker file through semihosting before the reset, and the next boot reads the marker, clears it and reports *watchdog*. The simulated bridge has no gate to cut.

- **Host:** the host build keeps a plain dispatcher. It is not a target that can be reset.

## Part E — validation surface

The bring-up application keeps one command:

- **watchdog\_stall** — schedules an action that never returns. The expiry handler cuts the power stage, the hardware resets the target, and the target comes back reporting the reset cause as *watchdog*.

The emulated target registers the same **watchdog\_stall** command from its platform rather than from the application, so the production firmware carries no stall command. A SIL scenario uses it to stall the event loop and then waits for the reboot banner to report the reset cause as *watchdog*. That runs the whole expiry path in continuous integration: the stalled action, the NMI early warning, the semihosting marker, the machine reset and the reset-cause report.

There is no command to enable or query the watchdog, because it has no state the application can change.

---

## Interfaces

### Provided

| Interface                   | Purpose                                         | Contract                                                               |
|-----------------------------|-------------------------------------------------|------------------------------------------------------------------------|
| Supervised event dispatcher | Run the application and prove it makes progress | Every MCU platform; supervision starts in the constructor, never stops |

### Required

| Interface                  | Purpose                                                     | Contract                                                                                             |
|----------------------------|-------------------------------------------------------------|------------------------------------------------------------------------------------------------------|
| <code>hal::Watchdog</code> | Early-warning interrupt, refresh, reset on a missed refresh | Constructed before the dispatcher; early warning at the lowest priority where the hardware allows it |
| Direct power-stage cutoff  | Safe state from the early-warning interrupt                 | Interrupt-safe, depends on no driver state; shared with the fault handler                            |

---

| Interface            | Purpose                                    | Contract                                                      |
|----------------------|--------------------------------------------|---------------------------------------------------------------|
| Reset-cause register | Report the watchdog reset on the next boot | Read and cleared once at boot by the error-handling component |

---



---

## Data Model

---

| Entity      | Field                 | Type / Unit | Range    | Notes                                                         |
|-------------|-----------------------|-------------|----------|---------------------------------------------------------------|
| Supervision | steps                 | count       | wraps    | Incremented before and after each action; odd while executing |
| Supervision | missed early warnings | count       | 0 ... N  | Reset whenever progress is seen                               |
| Supervision | N                     | count       | $\geq 1$ | expiration timeout / early-warning period                     |
| Supervision | expired               | flag        | set once | Refreshing stops; the hardware reset follows                  |

---



---

## State Machine

---

## Sequence Diagrams

**An event-loop action never returns**

**An interrupt never returns**

---

## Block Diagram

---

## Timing

With early-warning period  $P$  and  $N = \text{timeout} / P$ :

- The first early warning after an action starts always counts as progress. A hung action is therefore detected between  $N \cdot P$  and  $(N+1) \cdot P$  after it started, and the power stage is cut at that point.
- The hardware reset follows one period later, at most  $(N+2) \cdot P$  after the action started.

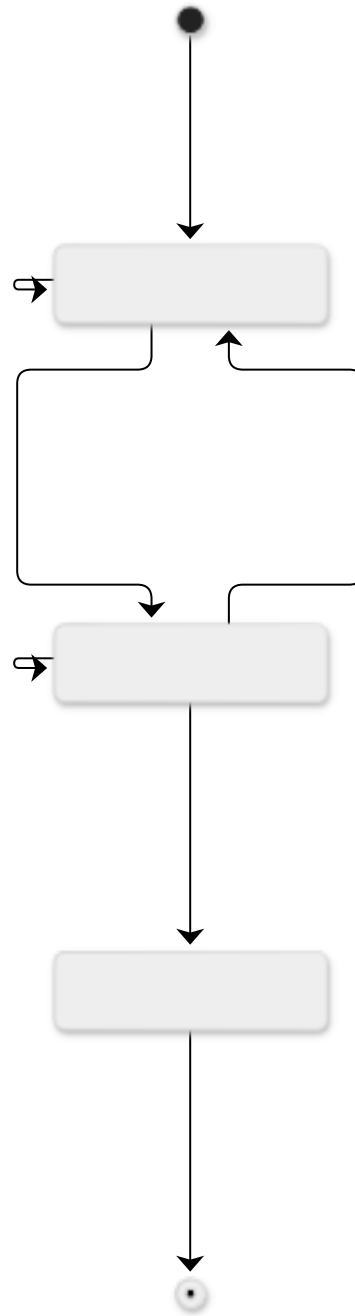


Figure 73: State diagram

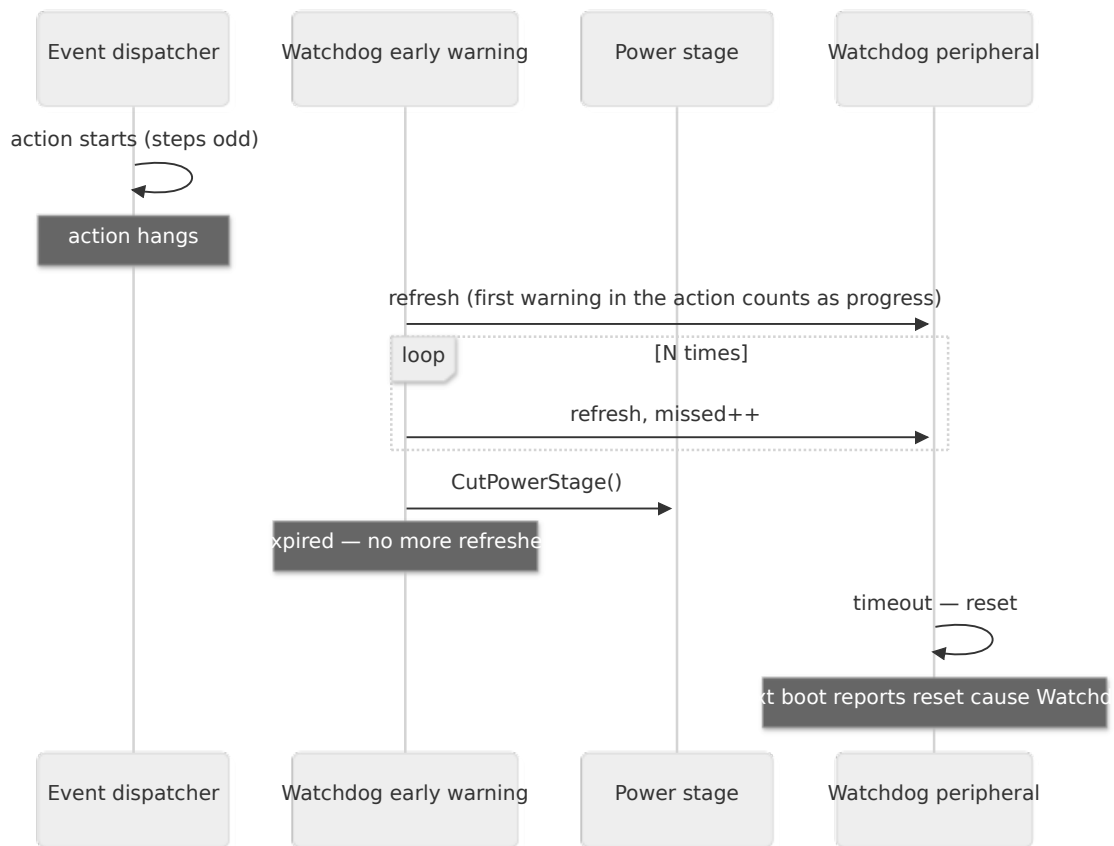


Figure 74: Sequence diagram

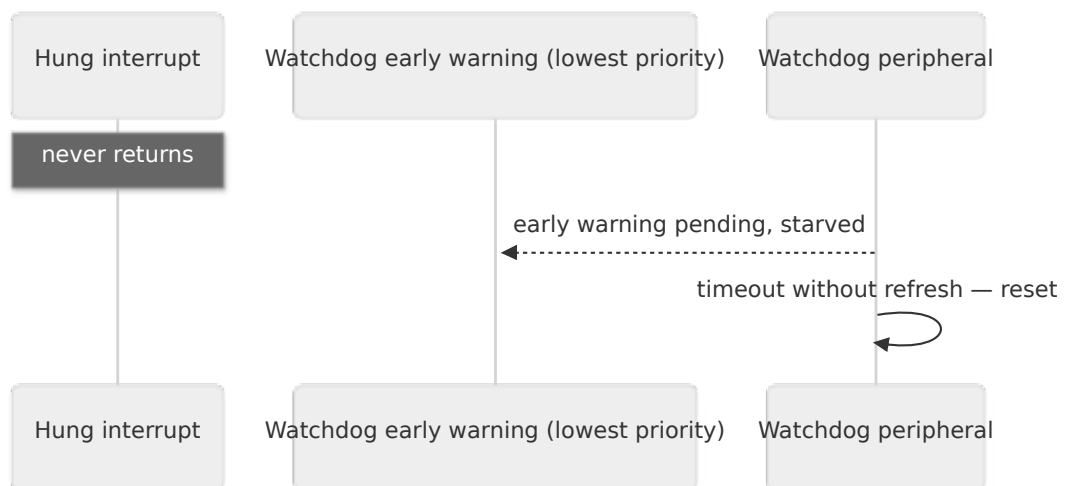


Figure 75: Sequence diagram

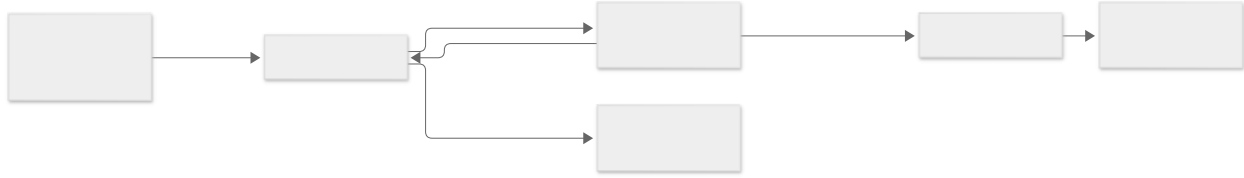


Figure 76: Block diagram

- With TI at  $P = 25$  ms and  $N = 4$ , that is 100–125 ms to the cutoff and at most 150 ms to the reset.
- With ST at  $P \approx 16$  ms (PCLK1 at 16 MHz) and  $N = 7$ , that is about 113–129 ms to the cutoff and at most about 145 ms to the reset.

A legitimate action must finish well inside  $N \cdot P$ . Long work is already split across actions: - the TI EEPROM mass erase is polled from a timer rather than waited for; - EEPROM writes wait only word by word; - identification sequences run from timers and phase-current callbacks.

New work that could block for longer than the expiration timeout has to be split across actions the same way.

## Constraints & Limitations

| Constraint                      | Value / Description                                                                                    |
|---------------------------------|--------------------------------------------------------------------------------------------------------|
| Scope                           | Proves the software is running; does not prove the control loops are running                           |
| Always on                       | Starts with the dispatcher; there is no enable, disable or query                                       |
| Boot before the dispatcher runs | Construction outside an action looks like progress; a hang there is caught only if interrupts stop too |
| Expiry handler                  | Interrupt context; power-stage cutoff only, then returns                                               |
| Early-warning priority          | Lowest on TI and ST, so a hung interrupt starves it; NMI on the emulated target                        |
| ST target                       | Cutoff is empty while the platform drives no bridge                                                    |
| Host build                      | No watchdog                                                                                            |
| Reset cause                     | Read from the MCU's reset-cause register; the emulated target reads a semihosting marker file instead  |

## Open Questions

| # | Question                                                      | Answer or options                                                               | Status   |
|---|---------------------------------------------------------------|---------------------------------------------------------------------------------|----------|
| 1 | Should the watchdog also prove the control loops are running? | No. It proves the software is running; bridge protection covers the power stage | answered |

| # | Question                                                                        | Answer or options                                                                                                      | Status   |
|---|---------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------|----------|
| 2 | Should a missed deadline be a fault code before the reset?                      | No — it is reported as the reset cause, because deferred reporting cannot outrun the reset                             | answered |
| 3 | Should the ST window-watchdog driver take the early-warning priority as config? | Yes, as the Tiva driver does; today the platform overrides the driver after start, which depends on construction order | open     |
| 4 | Can the emulated target's reset be asserted by a SIL job?                       | Yes: the SIL watchdog scenario stalls the event loop and asserts the <i>watchdog</i> reset cause after the reboot      | answered |

| Field     | Value                      |
|-----------|----------------------------|
| Title     | Integration Testing Design |
| Type      | design                     |
| Status    | accepted                   |
| Version   | 1.1.0                      |
| Component | integration-testing        |
| Date      | 2026-09-21                 |

**Note — Design-level document:** This document describes *how the integration tests are structured*. It expands on the architecture by specifying component responsibilities, data flows, and design decisions that would not be obvious to a reviewer unfamiliar with the system.

**Diagrams:** All visuals must be Mermaid fenced code blocks or ASCII art. External image references are **not allowed**.

## Overview

The suite is a **black-box driver**. It starts a target running the real firmware, speaks to it over the product's own CAN command set, and asserts on the telemetry and traces that come back. It does not reach into the firmware, does not mock its components, and knows nothing of its internal types.

Scenarios are authored in Gherkin and live in a single directory. A tag decides which target implements a scenario, and the runner is invoked with that tag.

| Tag  | Target                                           | Detailed in                                                  |
|------|--------------------------------------------------|--------------------------------------------------------------|
| @sil | Real firmware under an emulator, simulated motor | <a href="#">documentation/design/software-in-the-loop.md</a> |
| @hil | Real firmware on hardware, over the bridge       | This document                                                |

Earlier revisions described an in-process host fixture that mocked the platform and drove the state machine directly. No such fixture exists, and none is planned: mocking

the platform and then asserting on the state machine tests the wiring of the test, not the product.

---

## Responsibilities

**Is responsible for:** - Starting, stopping and isolating a target for each scenario - Presenting one interactor seam so a scenario reads the same whichever target implements it - Encoding commands and decoding telemetry on the product's wire format - Describing the simulated world a software-in-the-loop target runs in

**Is NOT responsible for:** - Control algorithm correctness in the small — that is what the unit tests under each library cover - CAN framing and payload layout — that is the wire contract tests' job - Substituting for hardware-in-the-loop, which alone exercises silicon, timing and analogue paths

---

## Component Details

### The interactor seam

Everything a scenario needs from a target sits behind one interface: lifecycle, command and telemetry transport, and — for a simulated target only — the world the firmware runs in.

The simulation seam — describe the plant, describe stored memory, restart — defaults to doing nothing and to reporting that simulation is unsupported. A hardware target inherits that default, and a scenario that configures a plant fails loudly rather than quietly testing something else.

### Scenario lifecycle

Each scenario gets a fresh target. Setup accumulates in the scenario's own state and is applied in a single restart, so a scenario that describes a plant, a calibration and a set of algorithms still starts the target once rather than once per description.

### Observation

Three channels, all of them the product's own:

- **Telemetry** carries the lifecycle state, the fault code, and the measured speed and position.
- **Command acknowledgements** carry acceptance or the reason for refusal.
- **Traces** carry what the firmware decided, most importantly which algorithm each loop ended up running, which can differ from what was asked for.
- **Plant trajectory**, on a simulated target only, carries what the motor actually did, sampled on the control-tick time base, and the tick each command frame was delivered on. This is the channel the performance and disturbance scenarios measure; see the software-in-the-loop design.

Two of the traces are measurements in their own right: the calibration record the firmware stores ([SM] **Calibration record**;) and the online estimates it prints on request ([EST]). Inertia and friction travel in micro-units on both, because the tracer prints three decimals. The identification

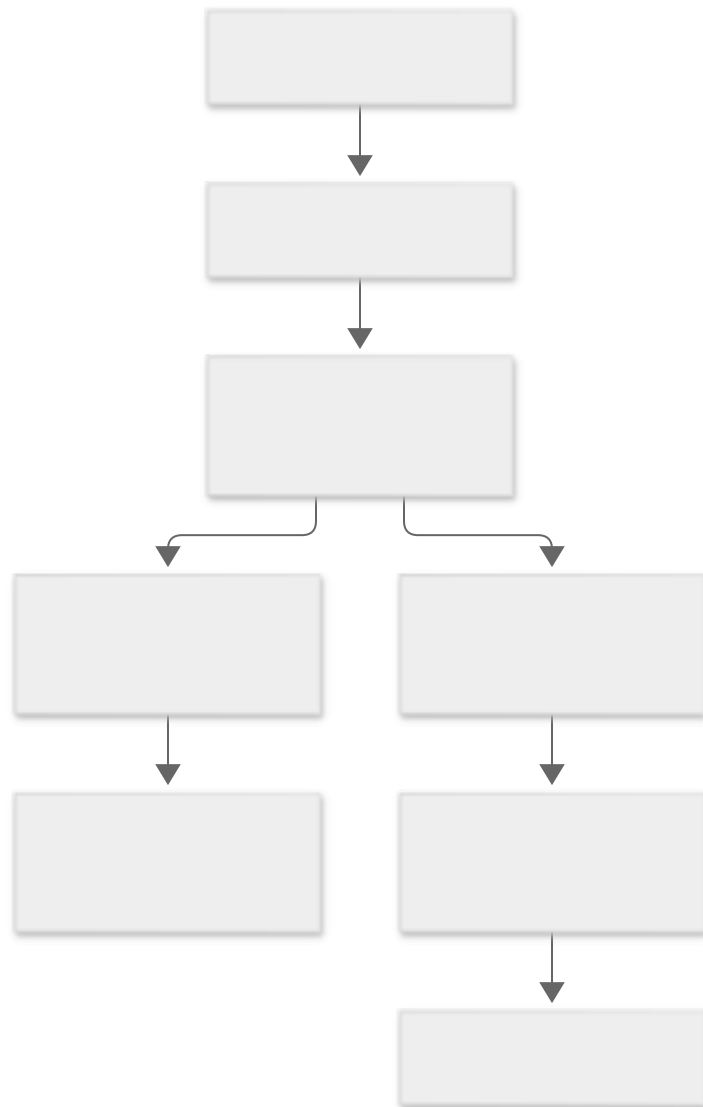


Figure 77: Block diagram

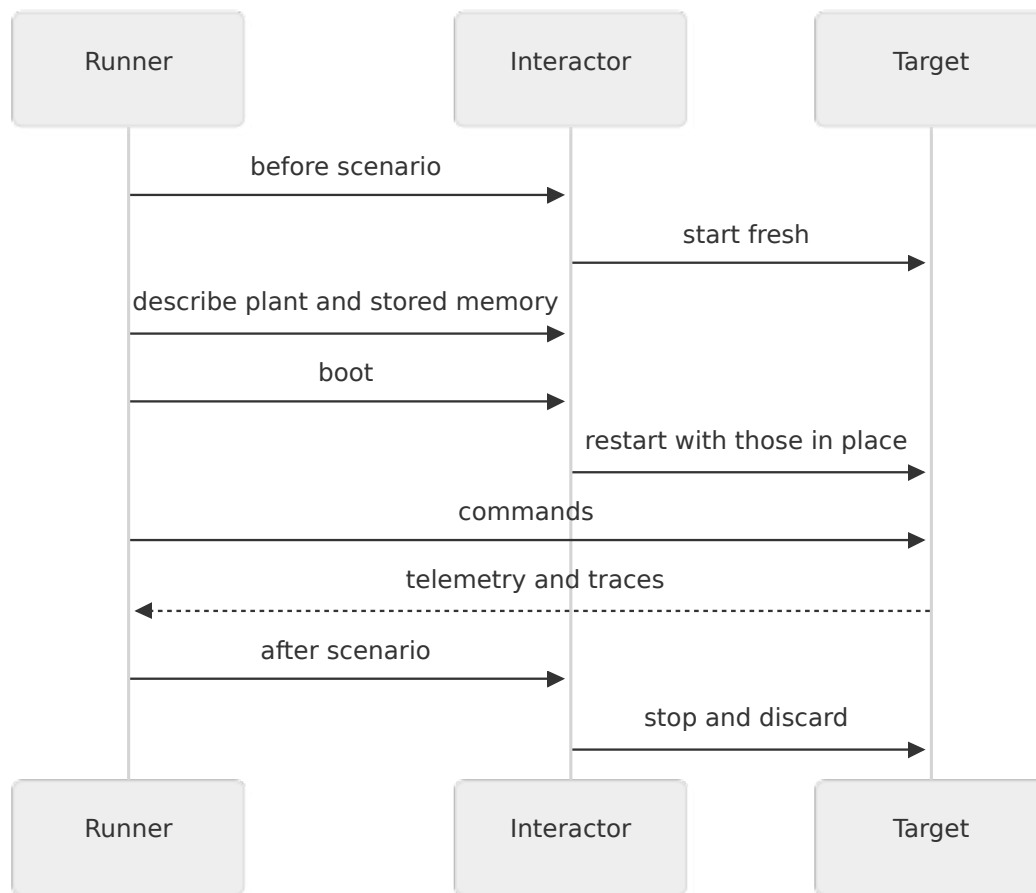


Figure 78: Sequence diagram

scenarios compare them, and the CAN electrical-parameters response, against the plant the scenario configured; the terminal command that asks for the estimates is the product's own, sent on the simulated target through the same socket as the CAN frames.

Trace output only drains while the target's event loop has work, so an assertion that waits for a trace polls telemetry rather than waiting passively. Trace lines are retained across the frames the command path reads, and cleared per scenario and per restart, so an assertion cannot match a line from an earlier boot.

## Isolation

Scenarios ran into each other through three shared resources, all now per session: the working directory holding the target's files, the socket carrying frames, and the captured output.

## Interfaces

### Provided to step definitions

| Capability        | Purpose                                                                     |
|-------------------|-----------------------------------------------------------------------------|
| Lifecycle         | Start, stop and restart the target                                          |
| Command transport | Send a category command and await its acknowledgement                       |
| Telemetry         | Await a state, a fault code, or read the measured position                  |
| Serial capture    | Drain and search the target's trace output                                  |
| Simulation        | Describe the plant and the stored calibration and configuration             |
| Response          | Capture the plant trajectory and measure step and disturbance metrics on it |

### Required from the system under test

- A CAN command set covering mode selection, calibration, alignment, enable, disable, set-points, telemetry requests and fault clearing
- Telemetry reporting state, fault code, measured speed and measured position
- Traces naming the algorithm each control loop is running
- For a simulated target, that it read its plant description at boot

No test-only command is added to the product to make the suite work.

## Feature-to-Requirements Mapping

Each scenario carries the requirement identifiers it verifies as tags alongside its runner tag. The traceability matrix is generated from those tags during release, reading the feature directory directly, so the mapping lives with the scenarios rather than in this document where it would rot.

**e-foc**

Field-Oriented Control for BLDC/PMSM Motors

*Real-time · Embedded · No heap allocation*

Generated from **documentation/** with Pandoc and Mermaid.

Source, requirements, and design documents at the project repository.