Introduction to Motor3508 Code Analysis

Austin Yang ∈ Illini RoboMaster
[austiny2@illinois.edu, https://www.illinirobomaster.com]

October 18, 2025

Contents

1 Introduction

1(a) Plan

This tutorial will go through the `motor.h` and `motor.cc` files line-by-line with a focus on analyzing the `Motor3508` class implementation. You can find the files covered at
https://github.com/illini-robomaster/iRM_Embedded_2026/blob/embedded_tutorial/shared/libraries/motor.h
https://github.com/illini-robomaster/iRM_Embedded_2026/blob/embedded_tutorial/shared/libraries/motor.cc
https://github.com/illini-robomaster/iRM_Embedded_2026/blob/embedded_tutorial/shared/bsp/bsp_can.h
You can find the example at
https://github.com/illini-robomaster/iRM_Embedded_2026/blob/main/examples/motor/m3508_speed.cc

1(b) A note on CAN IDs

The 3508 motor uses CAN bus communication for:

- Receiving: Send motor current commands from controller to motor
- Feedback: Receive telemetry data from motor to controller

This should be familiar if you have read the previous tutorial. The IDs we use are chosen for a reason. Motors send feedback on IDs `0x201`–`0x208` (or `0x1FF + id`), and commands are received by *all motors in the group* on ID `0x200` (or `0x1FF`).

Remark | Being able to use a single CAN frame to control multiple motors helps with, for example, synchronization. This will be covered in 4.

2 Understanding the Motor3508 Header File

We will start by examining the Motor3508 class definition in `motor.h`.

2(a) Class Hierarchy and Inheritance

Let's look at the Motor3508 class declaration:

```
190 /**
191  * @brief DJI 3508 motor class
192  */
193 class Motor3508 : public MotorCANBase {
194  public:
195   /* constructor wrapper over MotorCANBase */
196   Motor3508(bsp::CAN* can, uint16_t rx_id);
197   /* implements data update callback */
198   void UpdateData(const uint8_t data[]) override final;
199   /* implements data printout */
200   void PrintData() const override final;
201   /* override base implementation with max current protection */
202   void SetOutput(int16_t val) override final;
203
204   int16_t GetCurr() const override final;
205
206   uint16_t GetTemp() const override final;
207
208
209
210  private:
211   volatile int16_t raw_current_get_ = 0;
212   volatile uint8_t raw_temperature_ = 0;
213 };
```

The Motor3508 class inherits from `MotorCANBase`, which provides the base functionality for our DJI motors. You should know what inheritance is in programming languages. Here it helps us:

- Reuse common functionality
- Maintain consistent interfaces across different motor types (important!)
- Implement different behavior for different motor types

2(b) MotorCANBase

The base class `MotorCANBase` provides the following default functionality:

- CAN object/CAN ID storage

- Common data (`theta_`, `omega_`)
- Standard interface methods (GetTheta(), GetOmega(), etc.)
- `TransmitOutput`

Take a look:

```cpp
//===================================... (truncated)
// MotorCanBase(DJI Base)
//===================================... (truncated)

/**
 * @brief base class for CAN motor representation
 */
class MotorCANBase : public MotorBase {
 public:
  /**
   * @brief base constructor
   *
   * @param can    CAN instance
   * @param rx_id  CAN rx id
   */
  MotorCANBase(bsp::CAN* can, uint16_t rx_id);

  /**
   * @brief base constructor for non-DJI motors
   * @param can     CAN instance
   * @param rx_id   CAN rx id
   * @param type    motor type
   */
  MotorCANBase(bsp::CAN* can, uint16_t rx_id, uint16_t type);

  /**
   * @brief update motor feedback data
   * @note only used in CAN callback, do not call elsewhere
   *
   * @param data[]  raw data bytes
   */
  virtual void UpdateData(const uint8_t data[]) = 0;

  /**
   * @brief print out motor data
   */
  virtual void PrintData() const = 0;

```

```
103   /**
104    * @brief get rotor (the cap spinning on the back of the motor) angle, in [rad]
105    *
106    * @return radian angle, range between [0, 2PI]
107    */
108   virtual float GetTheta() const;
109
110   /**
111    * @brief get angle difference (target - actual), in [rad]
112    *
113    * @param target  target angle, in [rad]
114    *
115    * @return angle difference, range between [-PI, PI]
116    */
117   virtual float GetThetaDelta(const float target) const;
118
119   /**
120    * @brief get angular velocity, in [rad / s]
121    *
122    * @return angular velocity
123    */
124   virtual float GetOmega() const;
125
126   /**
127    * @brief get angular velocity difference (target - actual), in [rad / s]
128    *
129    * @param target  target angular velocity, in [rad / s]
130    *
131    * @return difference angular velocity
132    */
133   virtual float GetOmegaDelta(const float target) const;
134
135   virtual int16_t GetCurr() const;
136
137   virtual uint16_t GetTemp() const;
138
139   virtual float GetTorque() const;
140
141   /**
142    * @brief transmit CAN message for setting motor outputs
143    *
144    * @param motors[]    array of CAN motor pointers
145    * @param num_motors  number of motors to transmit
146    */
147   static void TransmitOutput(MotorCANBase* motors[], uint8_t num_motors);
```

```
148   /**
149    * @brief set ServoMotor as friend of MotorCANBase since they need to use
150    *        many of the private parameters of MotorCANBase.
151    */
152   friend class ServoMotor;
153
154   volatile bool connection_flag_ = false;
155
156  protected:
157   volatile float theta_;
158   volatile float omega_;
159
160  private:
161   bsp::CAN* can_;
162   uint16_t rx_id_;
163   uint16_t tx_id_;
164 };
```

2(c) Virtual Functions and Polymorphism

Notice the `virtual` keyword, both in 3508 and the base. The `virtual` keyword in the base class enables *polymorphism*, allowing different motor types to be treated through the same interface while providing different implementations.

Setting something as `virtual` forces all its child classes to provide an implementation. Since C++ is strictly typed, we also have to match the typing no matter what, which makes it useful for defining interfaces.

Tip | If you are familiar with Python, this is similar to `ABC` and `abstractmethod` from the `abc` module. If you provide annotations, this can 'enforce' interfaces (at lint time, that is).

The `override final` keywords in the `Motor3508` indicate:

- `override`: These functions override virtual functions from the base class.
- `final`: No further classes can override these functions.

Tip | Again, in Python, you can use the decorators `@override` and `@final` to achieve similar results.

Remark | Python 3.14 (πthon) provides many quality of life changes. Update now!

2(d) Volatile Variables

Notice the `volatile` keyword on the member variables:

```
214 private:
215  volatile int16_t raw_current_get_ = 0;
216  volatile uint8_t raw_temperature_ = 0;
```

The `volatile` keyword tells the compiler that these variables can change unexpectedly (e.g., from CAN interrupts). This prevents certain optimizations that might interfere with real-time data updates.

Remark | In embedded systems, `volatile` is commonly used for variables that are shared between main code and interrupt handlers, or that represent hardware registers.

Remark | When you compile code, your compiler automatically performs optimizations (you can control this). Typing helps optimization: the stronger the restrictions, the more the compiler can do to optimize. We usually assume things are not volatile, and the compiler does not either; we must use `volatile` to explicitly mark them as such.

3 Understanding the Motor3508 Implementation

Now let us examine the `Motor3508` implementation in motor.cc.

```
//====================================... (truncated)
// Motor3508
//====================================... (truncated)

Motor3508::Motor3508(CAN* can, uint16_t rx_id) : MotorCANBase(can, rx_id) {
  can->RegisterRxCallback(rx_id, can_motor_callback, this);
}

void Motor3508::UpdateData(const uint8_t data[]) {
  const int16_t raw_theta = data[0] << 8 | data[1];
  const int16_t raw_omega = data[2] << 8 | data[3];
  raw_current_get_ = data[4] << 8 | data[5];
  raw_temperature_ = data[6];

  constexpr float THETA_SCALE = 2 * PI / 8192;  // digital -> rad
  constexpr float OMEGA_SCALE = 2 * PI / 60;    // rpm -> rad / sec
  theta_ = raw_theta * THETA_SCALE;
  omega_ = raw_omega * OMEGA_SCALE;

  connection_flag_ = true;
}

void Motor3508::PrintData() const {
  print("theta: % .4f ", GetTheta());
  print("omega: % .4f ", GetOmega());
  print("raw temperature: %3d ", raw_temperature_);
  print("raw current get: % d \r\n", raw_current_get_);
}

void Motor3508::SetOutput(int16_t val) {
  constexpr int16_t MAX_ABS_CURRENT = 14690;  // ~20A
  output_ = clip<int16_t>(val, -MAX_ABS_CURRENT, MAX_ABS_CURRENT);
}

int16_t Motor3508::GetCurr() const { return raw_current_get_; }

uint16_t Motor3508::GetTemp() const { return raw_temperature_; }
```

3(a) Constructor Implementation

```
Motor3508::Motor3508(CAN* can, uint16_t rx_id) : MotorCANBase(can, rx_id) {
```

```
117   can->RegisterRxCallback(rx_id, can_motor_callback, this);
118 }
```

The constructor does two things:

1. Calls the base class constructor with a pointer to the CAN instance (`can`) and the receive ID (`rx_id`).
2. Registers a callback function to handle incoming CAN data.

Remark | This is different from the callback function presented in the first tutorial; that one was for Linux CAN. However, their functionality is similar.

Exercise: Look through the code for `RegisterRxCallback` and figure out how it is implemented. Starting point: look at what it does in 3(a).1.

3(a).1 Callback Registration

The `RegisterRxCallback` function sets up an *interrupt handler* that will automatically call `can_motor_callback` whenever a CAN message with the specified ID is received.

```
49 static void can_motor_callback(const uint8_t data[], void* args) {
50   MotorCANBase* motor = reinterpret_cast<MotorCANBase*>(args);
51   motor->UpdateData(data);
52 }
```

The concept of callbacks (or sometimes a "hook") is *extremely* important for embedded systems. Imagine if everything had to poll everything else for what they need. That would be very messy and ugly.

- `reinterpret_cast` converts the generic void pointer back to our `MotorCANBase*` (remember what a void pointer is?).
- From the motor we just recast, we call `UpdateData` on the data we received.

This works for any motor type that inherits from `MotorCANBase`. If you read the first tutorial, this should seem very familiar!

Tip | You will get an introduction to interrupts near the end of ECE 120, if you take it.

3(b) Data Parsing Implementation

The `UpdateData` function is where the metaphorical magic happens. We parse (decode) the received data based on whatever was written on the datasheet of our motor.

```cpp
void Motor3508::UpdateData(const uint8_t data[]) {
  const int16_t raw_theta = data[0] << 8 | data[1];
  const int16_t raw_omega = data[2] << 8 | data[3];
  raw_current_get_ = data[4] << 8 | data[5];
  raw_temperature_ = data[6];

  constexpr float THETA_SCALE = 2 * PI / 8192;  // digital -> rad
  constexpr float OMEGA_SCALE = 2 * PI / 60;    // rpm -> rad / sec
  theta_ = raw_theta * THETA_SCALE;
  omega_ = raw_omega * OMEGA_SCALE;

  connection_flag_ = true;
}
```

You can find the datasheet for the C620 controller (used to control the M3508) here: https://rm-static.djicdn.com/tem/17348/RoboMaster %20C620%20Brushless%20DC%20Motor%20Speed%20Controller%20V1.01.pdf.

3(b).1 CAN Data Structure

From examining the code, or reading the data sheet (page 17), we can figure out that the 3508 motor sends 8 bytes of data per CAN frame in the following format:

Bytes	Decoded as	Description
0	`raw_theta`	Rotor angle (upper 8 bits)
1		Rotor angle (lower 8 bits)
2	`raw_omega`	Rotational speed (upper 8 bits)
3		Rotational speed (lower 8 bits)
4	`raw_current_get_`	Torque current (upper 8 bits)
5		Torque current (lower 8 bits)
6	`raw_temperature_`	Motor temperature (8 bits)
7	Null	Unused

Table 1: C620 data frame format

Remeber what a CAN data frame is from tutorial 2? From the datasheet,

Data	Format/Units
`raw_theta`	`0` to `8192` (0° to 360°)
`raw_omega`	RPM
`raw_current_get_`	`-16384` to `16384` (−20A to 20A)
`raw_temperature_`	°C

Table 2: C620 data frame field units

3(b).2 Bit Shifting

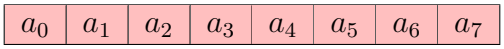
We use bit shifting operations combine individual bytes into 2-byte values:

```
121 const int16_t raw_theta = data[0] << 8 | data[1];
```

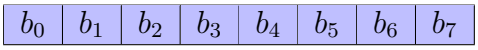
This is equivalent to (and better than): `raw_theta = (data[0] * 256) + data[1]`

- `data[0] << 8`: Shift the high byte left by 8 bits (multiply by 256)
- `| data[1]`: Combine with the low byte using bitwise OR

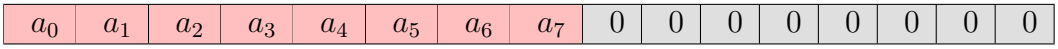
Let us act it out, step by step (binary $a_i, b_j \in \mathbb{Z}/2\mathbb{Z}$):



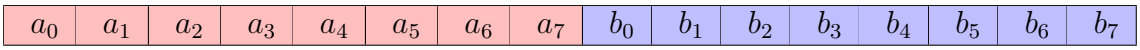
Dummy `data[0]` (8-bit)



Dummy `data[1]` (8-bit)



`data[0] << 8`



`data[0] << 8 | data[1]` (16-bit)

Don't worry too much about intermediate types: C++ automatically converts your data when using bitshift operators, and everything is implicitly converted back to `uint16_t` on assignment.

One thing we have to worry about, though, is *sign extension* – you will encounter this in ECE 110, if you take it (search it up if not). This is why we store data as `uint8_t`.

3(b).3 Unit Conversion and Scaling

The `constexpr` variables define conversion factors:

```
126 constexpr float THETA_SCALE = 2 * PI / 8192;  // digital -> rad
127 constexpr float OMEGA_SCALE = 2 * PI / 60;    // rpm -> rad / sec
```

These convert raw motor data to the units we want:

- Position: 8192 encoder counts = 2π rad (one rotation)
- Velocity: 1 RPM = $2\pi/60$ rad s^{-1}

Remark | Using `constexpr` ensures these calculations are performed at compile time, saving computational resources on the embedded system.

Remark | Remeber what we mentioned in 2(d)? This is an example of declaring a restriction to help the compiler make optimizations.

3(c) Current Limiting for Safety

```
141 void Motor3508::SetOutput(int16_t val) {
142   constexpr int16_t MAX_ABS_CURRENT = 14690;  // ~20A
143   output_ = clip<int16_t>(val, -MAX_ABS_CURRENT, MAX_ABS_CURRENT);
144 }
```

This critical safety function prevents damage to the motor:

- Clips the output command to ± 14690 units (ignore the comment, it wasn't changed when changing the limit).
- Uses the `clip` to enforce limits (notice the template – you should be familiar from the first tutorial!).

Tip | Always include safety limits! Motors cost money, and the smoke does not smell good.

3(d) Debug Output Function

```
134 void Motor3508::PrintData() const {
135   print("theta: % .4f ", GetTheta());
136   print("omega: % .4f ", GetOmega());
137   print("raw temperature: %3d ", raw_temperature_);
138   print("raw current get: % d \r\n", raw_current_get_);
139 }
```

This function provides formatted output for debugging:

- `% .4f`: Print float with 4 decimal places, signed.
- `%3d`: Print integer with minimum 3 characters width.
- `% d`: Print signed integer with space for positive numbers.
- `\r\n`: Both types of newlines.

4 CAN Communication for Motor Control

Let's examine how multiple motors are controlled simultaneously through CAN.

4(a) TransmitOutput Implementation

The `TransmitOutput` static method (from MotorCANBase) sends commands to multiple motors in a single CAN frame:

```
79 void MotorCANBase::TransmitOutput(MotorCANBase* motors[], uint8_t num_motors) {
80   uint8_t data[8] = {0};
81
82   RM_ASSERT_GT(num_motors, 0, "Meaningless empty can motor transmission");
83   RM_ASSERT_LE(num_motors, 4, "Exceeding maximum of 4 motor commands per CAN message");
84   for (uint8_t i = 0; i < num_motors; ++i) {
85     RM_ASSERT_EQ(motors[i]->tx_id_, motors[0]->tx_id_, "tx id mismatch");
86     RM_ASSERT_EQ(motors[i]->can_, motors[0]->can_, "can line mismatch");
87     const uint8_t motor_idx = (motors[i]->rx_id_ - 1) % 4;
88     const int16_t output = motors[i]->output_;
89     data[2 * motor_idx] = output >> 8;
90     data[2 * motor_idx + 1] = output & 0xFF;
91   }
92   motor_val = motors[0]->output_;
93   motors[0]->can_->Transmit(motors[0]->tx_id_, data, 8);
94 }
```

4(a).1 Multi-motor Message Packing

This function packs up to 4 motor commands into one 8-byte CAN message:

CAN Bytes	Motor ID	Data
0-1	Motor 1 (ID 0x201)	Current command (16-bit)
2-3	Motor 2 (ID 0x202)	Current command (16-bit)
4-5	Motor 3 (ID 0x203)	Current command (16-bit)
6-7	Motor 4 (ID 0x204)	Current command (16-bit)

Table 3: CAN Command Message Format for Multiple Motors

You can find this on page 15 of the manual (at the bottom of 3(a).1).

4(a).2 Motor Index Calculation

The line `const uint8_t motor_idx = (motors[i]->rx_id_ - 1) % 4;` maps motor RX IDs to array indices:

- Motor with RX ID `0x201` →index 0 →CAN bytes 0-1

- Motor with RX ID `0x202` →index 1 →CAN bytes 2-3
- Motor with RX ID `0x203` →index 2 →CAN bytes 4-5
- Motor with RX ID `0x204` →index 3 →CAN bytes 6-7

Tip | Modulo arithmetic (`% 4`) allows the same function to work with multiple motor groups (`0x205`–`0x208` would map to the same byte positions).

Remark | Completely unrelated: modulo arithmetic is related to a whole branch of mathematics.

4(a).3 Data Packing

The bit manipulation to pack 16-bit motor commands into bytes (The comments aren't a part of the original file):

```
89 data[2 * motor_idx] = output >> 8;         // High byte
90 data[2 * motor_idx + 1] = output & 0xFF;   // Low byte
```

This is the reverse of the parsing operation we saw earlier - splitting a 16-bit value into two 8-bit bytes for transmission.

Exercise: draw it out like I did in 3(b).2. Or at least think about it.

4(b) CAN ID Management

The `MotorCANBase` constructor automatically determines the correct TX ID based on the motor's RX ID:

```
56 constexpr uint16_t RX1_ID_START = 0x201;
57 constexpr uint16_t RX2_ID_START = 0x205;
58 constexpr uint16_t RX3_ID_START = 0x209;
59 constexpr uint16_t TX1_ID = 0x200;
60 constexpr uint16_t TX2_ID = 0x1ff;
61 constexpr uint16_t TX3_ID = 0x2ff;
```

This creates three motor groups:

- Group 1: Motors `0x201`–`0x204` send feedback on 0x200.
- Group 2: Motors `0x205`–`0x208` send feedback on 0x1FF.
- Group 3: Motors `0x209`–`0x20C` send feedback on 0x2FF.

5 Example

Let's examine an example: m3508_speed.cc.

```cpp
#include "bsp_print.h"
#include "cmsis_os.h"
#include "controller.h"
#include "main.h"
#include "motor.h"

#define TARGET_SPEED 30

float kp = 0;
float ki = 0;
float kd = 0;

int16_t out = 0;

bsp::CAN* can = nullptr;
control::MotorCANBase* motor = nullptr;

void RM_RTOS_Init() {
  print_use_uart(&huart1);
  can = new bsp::CAN(&hcan1, true);
  motor = new control::Motor3508(can, 0x201);
}

void RM_RTOS_Default_Task(const void* args) {
  UNUSED(args);

  control::MotorCANBase* motors[] = {motor};
  control::PIDController pid(20, 15, 30);

  while (true) {
    float diff = motor->GetOmegaDelta(TARGET_SPEED);
    pid.Reinit(kp, ki, kd);
    out = pid.ComputeConstrainedOutput(diff);
    motor->SetOutput(out);
    control::MotorCANBase::TransmitOutput(motors, 1);
    motor->PrintData();
    osDelay(10);
  }
}
```

You should be able to understand this. If you weren't here for the PID lecture, wait for the next tutorial.

6 Concepts Covered

We analyzed the following concepts in the Motor3508 implementation:

6(a) C++ Programming

- Class inheritance and polymorphism
- `virtual` functions and method overriding
- `volatile` keyword for real-time data
- `constexpr` for compile-time calculations
- Static methods for shared functionality

6(b) Embedded Programming

- More CAN
- Interrupts and callbacks
- Bit manipulation for data packing/unpacking

6(c) Motor Control

- Encoder data parsing
- Current limiting for motor protection
- Multi-motor coordination

7 Conclusion

This tutorial provided an analysis of the Motor3508 code implementation, demonstrating how embedded motor control systems are designed and implemented in practice.

Same as previous tutorials: if there are any concepts you do not completely understand, search them up and ask an LLM for clarification.