HEX 4310

Safety Instructions

Important Safety Information

- It is strictly prohibited to hot-plug electrical appliances (plugging in electrical appliances while they are powered on).
- Ensure the device is stationary before power off unless it is an emergency. Do not power off during movement.
- It is strictly prohibited to use the device during charging.
- Confirm the device's operating voltage, power, and installation parameters before use. Issues caused by exceeding these parameters are not covered under warranty.
- Assess the IP rating, temperature, and suitability of the environment. Issues arising from harsh conditions are not covered under warranty.
- This device does not provide collision, fall, or proximity warnings. Conduct safety evaluations for integrated products and ensure compliance with relevant regulations and certifications to avoid major safety hazards.
- Read maintenance requirements to prevent irreversible damage from incorrect operations, such as battery over-discharge, low tire pressure causing hub damage, or lack of lubrication leading to axle wear.
- For first-time use, place the device in a safe, open area without loads. Follow the operation instructions and test all functions. Contact customer service if issues arise.
- In case of abnormalities or accidents, use the emergency stop or power off immediately to prevent further damage. Contact technical support and do not disassemble the device yourself. Note: Damage from unauthorized disassembly, modification, misuse, or force majeure events is not covered under warranty. The product is not liable for safety incidents with user-integrated devices. Users must conduct risk and reliability assessments.

1. Motor Introduction

The HEX 4310 geared motor is specifically designed for humanoid robot joints, desktop collaborative robotic arms, and similar applications. Its compact and lightweight structure makes it suitable for tight installation spaces. Functionally, it offers high positioning precision and high torque density, enabling precise joint control and handling of variable-load scenarios, with built-in overload protection for enhanced safety. Manufactured with high-strength aluminum and carefully designed gears, key components are strictly selected and undergo factory precision calibration, torque testing, and other inspections to ensure stable and reliable performance, supporting efficient equipment operation.

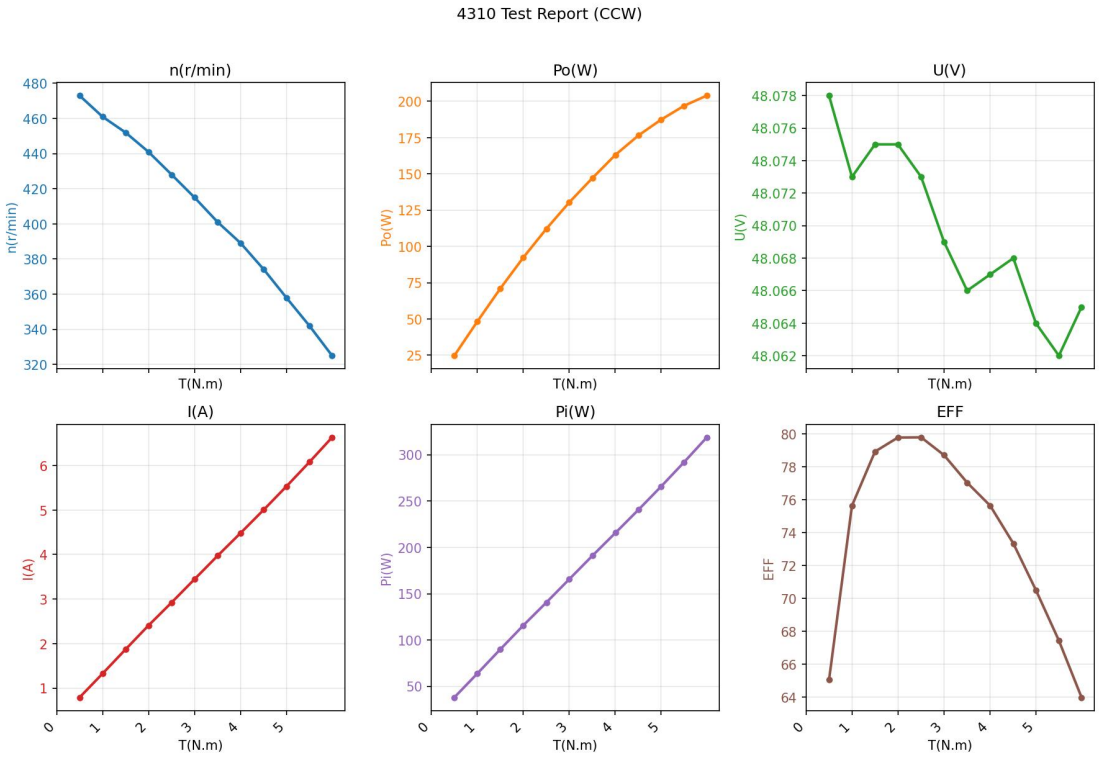
2. Specifications and Model Data

The HEX43xx series motors are integrated joint motors designed by HexFellow specifically for the embodied robotics industry. The series uses a planetary reduction system and offers advantages such as impact resistance, high torque transparency, low friction, and high

cost-effectiveness. The motors feature integrated drive and control with CAN-FD communication, and support hardware timestamping, synchronization, MIT control, one-to-many control, and other functions.

Technical Specifications

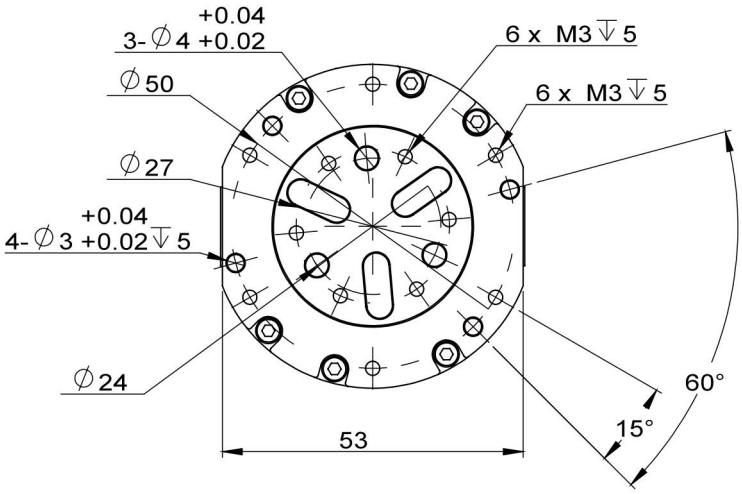
Motor Characteristics

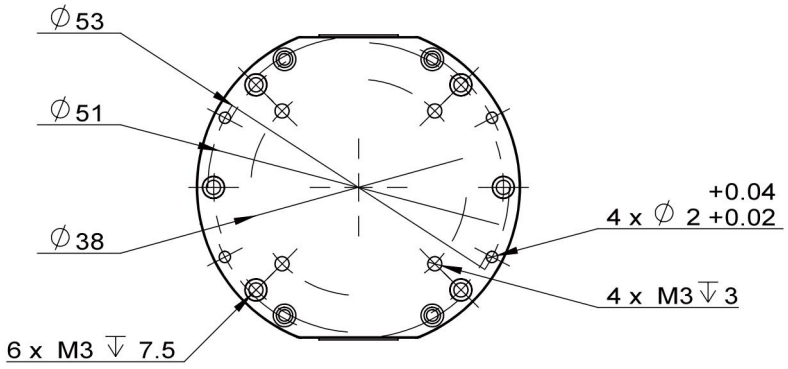


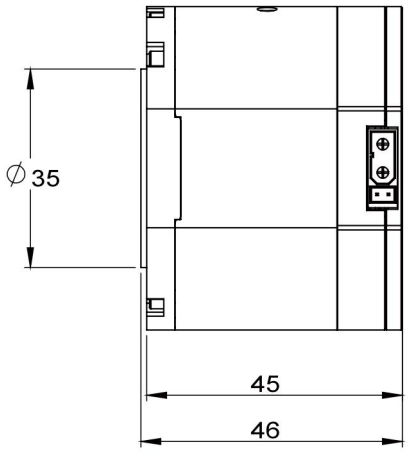
Motor Features

- Dual encoders with single-turn absolute position on the output shaft, no loss of absolute position on power failure; integrated motor and driver design with compact structure and high integration.
- Feedback of motor speed, position, torque, motor temperature, and other information via CAN-FD@5Mbps.
- Support for sending control commands to multiple motors simultaneously with a single CAN message, enabling one-to-many control.
- Customizable return messages (via CANopen PDO mechanism).
- Support for multiple control modes including position mode, velocity mode, torque mode, MIT mode, and more.

Motor Dimensions







Motor Parameters

Parameter	Data

Parameter	Data
Rated Voltage	48 V
Rated Current	3.7 A
Rated Speed	410 rpm
Rated Torque	3 Nm
Peak Torque	7 Nm
Peak Current	7.5 A
No-load Speed	500 rpm

Key Performance Parameters

Parameter	Data
Pole Pairs	28
Phase Inductance	0.55 mH
Phase Resistance	1.05 Ω
Reduction Ratio	10

Sensors

Parameter	Data
Encoder Bits	16-bit
Encoder	2

Parameter	Data
Count	
Encoder Type	Magnetic (single-turn)

Communication Interface

Parameter	Data
Control Interface	CANFD@5Mbps

Control Modes

- MIT Mode
- Velocity Mode
- Position Mode
- Torque Mode
- Direct VS Mode (Profile Torque Mode)

Protection Features

Protection Type	Trigger Condition
Driver Over-temperature	Driver temperature > 120° C
Motor Over-temperature	Motor coil temperature > 110° C
Over-voltage	DC bus voltage > 65 V
Phase Over-current	Phase current > 14 A (PEAK)

Protection Type	Trigger Condition
Under-voltage	DC bus voltage < 14 V
Stall Protection	Driver detects motor stall

Protection Behavior

All protections above will cause the motor to exit "Enabled Mode" when triggered.

Motor Dimensions and Weight

Parameter	Data
Outer Diameter	57 mm
Height	45 mm
Weight	310 g
Operating Temperature	-20~40° C

3. Unpacking

Packing List

No.	Item	Quantity
1	Motor (with driver)	1 pc
2	M4 Dowel Pin	3 pcs

Interface and Wiring

The motor has two power ports (Power Port 1 and Power Port 2), both using XT30(2+2)-F connectors with integrated CAN communication terminals.

Function	Description
Power	Connect power via XT30(2+2)-F plug, rated voltage 48V
CAN Bus	Connect to external controller to receive commands and report motor status
Daisy-chain	Either port can be used independently or daisy-chained for convenient wiring

LED Status Indicators

State	Indicator	Description
Normal	Green blinking	Motor operating normally
Abnormal	Red blinking	Motor fault — read fault type via CANOpen protocol object 0x603F

Precautions

Before Use

- Operating temperature: -20~40°C. Motor winding temperature must not exceed 100°C (NTC detection). Operating outside specifications will cause damage.
- Prevent foreign objects from entering the motor, which may cause jamming or short-circuit damage.
- Check that all items in the packing list are complete and intact before use.

- Mount using the screw holes shown in the dimension drawing and wire according to the interface definitions.
- Motor surface temperature rises during prolonged loaded operation — do not touch.
- Encoders are factory-calibrated. Do not disassemble the motor; unauthorized disassembly voids warranty support.

4. API Control

CAN Configuration

Hardware Configuration

Termination resistors must use Split termination. Star connection is prohibited. You must ensure that there are exactly two sets of termination resistors on the same CAN line, and both are at the two ends of the line.

Software Configuration

Our motors use CAN-FD. The arbitration segment is fixed at 1Mbps with a sample point of 0.8. The data segment baud rate is fixed at 5Mbps with a sample point of 0.75.

Recommended CAN configuration command:

```
ip link set can0 type can bitrate 1000000 sample-point 0.8 dbitrate 5000000 sample-point 0.75
sjw 5 dsjw 3 fd on
```

Note

The sample point must be set correctly, otherwise sending long frames will definitely cause communication failure. Some USB-to-CAN devices may not correctly support setting these parameters. If you encounter this situation, please use an SOC with hardware CAN-FD support such as RK3576, or consider a Maver-L4 set that includes a main controller.

Configure Motor ID

Our motors use the CANOpen(CiA301) protocol. CANOpen is a widely used protocol in industrial automation with many tutorials available online, so we won't elaborate here. Some minor modifications have been

made to PDO, allowing PDO to have a maximum length of 64 bytes, making full use of CAN-FD bandwidth.

CAN-ID is set through object dictionary 2001h, 1h.

Be Careful About ID Conflict

Each motor module has two motors. Please be careful not to set the same ID for two motors in the same module, otherwise you will not be able to distinguish between the two motors.

For example, using the canopend tool for configuration, change the ID of the original 1st motor to 3rd:

> For how to use the canopend tool, please refer to its Github introduction, here we will not explain it.

```
root@hexfellow-ffa9635b86b733d7 ~# ./canopend can0 -i 0x2f -c "stdio"
./canopend[2291]: CANopen device, Node ID = 0x00, starting
./canopend[2291]: CANopen command interface on "standard IO" started
./canopend[2291]: CAN Interface "can0" RX buffer set to 477 messages (212992 Bytes)
./canopend[2291]: CANopen NMT state changed to: "initializing" (0)
./canopend[2291]: CANopen device, Node ID = 0x2F, communication reset
./canopend[2291]: CANopen device, Node ID = 0x2F, running ...
./canopend[2291]: CANopen NMT state changed to: "pre-operational" (127)
./canopend[2291]: CANopen Emergency message from node 0x2F: errorCode=0x5000, errorRegister=0x01, errorBit=0x2F, infoCode=0x00000004
1 w 0x2001 01 u8 3
[0] OK
1 w 0x1010 01 vs "save"
[0] OK
```

New ID Takes Effect After Power Off

After setting a new ID, it will only take effect after power off. Please be careful not to set the wrong ID.

Quick start: complete control demo

If you want a working end-to-end path from cold boot to steady rotation first, see the open-source repo hex-motor-control-test. It ships two minimal C demos (not a library) that drive **hexfellow** motors over CAN-FD using CANopen (CiA301 + CiA402) with 64-byte PDO extensions:

Program	Description
`demo_mit`	MIT mode (`6060h = 5`), spins every motor at a constant

	velocity
`demo_velocity`	Profile-velocity mode (`6060h = 3`), constant velocity with per-frame `6072h` torque limit

The demos cover sequential per-motor SDO init, NMT to Operational, CiA402 enable, a master heartbeat plus shared RPDO control loop, and TPDO telemetry. They assume CANopen IDs **1…N** (contiguous, up to **8** motors) and firmware version **08**.

Configure the CAN interface for CAN-FD as described above, then build and run—for example, four motors on `can0` at 0.5 rev/s:

```
make./build/demo_mit can0 4 0.5./build/demo_velocity can0 4 1.0 200
```

See the repository README for CLI options, cross-compilation, heartbeat/RPDO ordering, and other details.

Demo only

This repository is **for learning and bench verification only**, not production use. It has no robust error recovery or safety interlocks. Audit it yourself and write a product-grade driver before any deployment where motor misbehavior could cause harm or loss.

Parse CAN Messages

The default TPDO mapping is as follows:

```
// TPDO1: Position(0x6064, 0, i32(length 0x20)), Timestamp(0x1013 0, u32(length 0x20)),
Torque(0x6077, 0, i16(length 0x10)), Error Code(0x603F, 0, u16(length 0x10)). 12 bytes,
1000Hz.// TPDO2: Status Word(2 bytes), Driver Temperature(2 bytes), Motor Temperature(2 bytes),
Control Word(2 bytes), Error Code(2 bytes). 10 bytes(actually sent 12 bytes), 50Hz.
```

In summary, a typical complete motor initialization and control flow is as follows:

1. The host computer program reads **once** object 1018h 03h to obtain the motor firmware version (must be version 08) to ensure compatibility between the host computer program and the motor firmware version.

2. The host computer program reads **once** object 6076h 00h to obtain the motor's peak torque for subsequent calculation and conversion to international unit torque.
3. The host computer program reads **once** object 2003h 07h to obtain the MIT KP and KD scale factors for conversion to KP and KD in international unit dimensions. (Only for firmware version 08, and only if MIT mode needs to be used)
4. The host computer program configures heartbeat detection object 1016h 00h to ensure the motor automatically exits enable when the host computer program ends. (Optional, timing starts from the first received heartbeat from the target node)
5. The host computer configures TPDO and RPDO mappings (if parameters have been saved through object 1010h, this step is not needed)
6. The host computer program generates a CAN receive processing process to parse and handle the motor's TPDO, HEARTBEAT, and optional EMCY
7. The host computer sends NMT commands to set the motor to Operational state.
8. The host computer correctly configures the CiA402 state machine, selects the mode to enter, and enables the motor.
9. The host computer starts sending its own heartbeat (optional, if heartbeat timeout protection is needed. Own heartbeat is recommended to be sent at low frequency.), and starts sending control commands through PDO.

Here is a code example for the **parsing** part (i.e., step 6):

There are a few details

- Speed is calculated on the host computer using a deque and hardware timestamp deltas, eliminating potential time errors and allowing users to choose their own filtering algorithm.
- Since 6077h has no unit, it is 0-1000, provided in thousandths. Therefore, it needs to be multiplied by 6076h to get the torque value in international units. You should read **once** 6076h at motor startup (constant for the same motor). Use it for multiplication afterwards.

```
for frame in frames {    let id = frame.id();    if let socketcan::Id::Standard(id) = id {
let id = id.as_raw();        let canopen_id = (id as u8) & 0x7F;        if canopen_id !=
```

```
self.canopen_id { continue; } if let Ok(communication_object_code) =
canopen::canopen_types::CommunicationObject::try_from(id) { let constants
= self.runtime_constants.get(); match communication_object_code {
canopen::canopen_types::CommunicationObject::TPDO1 => { if
frame.data().len() != 12 { continue; }
// TPDO1: Position (4 bytes F32, single-turn rev [-0.5, 0.5)), Timestamp (4 bytes), Torque
(2 bytes), Error Code (2 bytes). 12 bytes, 1000Hz. let position_f32 =
f32::from_le_bytes([ frame.data()[0],
frame.data()[1], frame.data()[2],
frame.data()[3], ]); let timestamp =
u32::from_le_bytes([ frame.data()[4],
frame.data()[5], frame.data()[6],
frame.data()[7], ]); let speed = {
let dq = &mut self.runtime_data.lock().unwrap().raw_position; let prev
= dq.back().map(|(p, _)| *p).unwrap_or_default(); let new_pos =
prev.update(position_f32); dq.push_back((new_pos, timestamp));
if dq.len() > self.speed_average_window { let last_data =
dq.pop_front().unwrap(); let time_diff_us =
timestamp.overflowing_sub(last_data.1).0; let time_diff =
time_diff_us as f64 / 1_000_000.0f64; let position_diff_rev =
new_pos.diff(last_data.0); position_diff_rev as f64 *
std::f64::consts::TAU / time_diff } else {
0.0f64 } }; let torque = if let
Some(peak_torque) = self.runtime_constants.get() { let peak_torque =
peak_torque.peak_torque; let torque =
i16::from_le_bytes([frame.data()[8], frame.data()[9]]); // Convert
mapped torque to Nm torque as f64 * peak_torque as f64 /
1000.0f64 } else { 0.0f64 };
let error_code = u16::from_le_bytes([frame.data()[10],
frame.data()[11]]); { let runtime_data = &mut
self.runtime_data.lock().unwrap(); // Raw position is already updated,
so no need to update it here runtime_data.last_contact_time =
std::time::Instant::now(); runtime_data.smoothed_speed = speed;
runtime_data.torque = torque; runtime_data.raw_error_code =
error_code; } }
canopen::canopen_types::CommunicationObject::TPDO2 => { if
frame.data().len() != 12 { continue; }
// TPDO2: Status Word(2 bytes), Driver Temperature(2 bytes), Motor Temperature(2 bytes),
Control Word(2 bytes). 8 bytes, 50Hz. let status_word =
u16::from_le_bytes([frame.data()[0], frame.data()[1]]); let
driver_temperature = i16::from_le_bytes([frame.data()[2],
frame.data()[3]]) as f32 * 0.1f32; let
motor_temperature = i16::from_le_bytes([frame.data()[4],
frame.data()[5]]) as f32 * 0.1f32; let
```

```
control_word = u16::from_le_bytes([frame.data()[6], frame.data()[7]]);
{ let runtime_data = &mut self.runtime_data.lock().unwrap();
runtime_data.status_word = status_word;
runtime_data.driver_temperature = Some(driver_temperature);
runtime_data.motor_temperature = Some(motor_temperature);
runtime_data.read_control_word = control_word; } } }
canopen::canopen_types::CommunicationObject::Heartbeat => { if
frame.data().len() != 1 { continue; }
let data = frame.data()[0]; if let Ok(nmt_state) = NMTState::try_from(data)
{ if nmt_state != NMTState::Operational { {
let mut runtime_data = self.runtime_data.lock().unwrap(); if
runtime_data.last_nmt_send_time.elapsed().as_millis() >
250 {
runtime_data.last_nmt_send_time =
std::time::Instant::now(); } }
ret.push(canopen::set_nmt_state( NMTState::Operational,
self.canopen_id, )); }
self.runtime_data.lock().unwrap().nmt_state =
Some(nmt_state); } } _ =>
{} } } }}
```

When using MIT mode, the compact MIT control parameters are recommended; see object 2004h in the version-08 object dictionary. Below is sample Rust code that packs MIT control targets into the MIT control word:

```
impl Default for MitTargetMapping { fn default() -> Self { Self {
position_min: -0.5, position_max: 0.5, velocity_min: -10.0,
velocity_max: 10.0, torque_min: -10.0, torque_max: 10.0,
kp_min: 0.0, kp_max: 100.0, kd_min: 0.0, kd_max:
20.0, } }} fn float_to_uint(x: f32, x_min: f32, x_max: f32, bits: u32) -> u32 {
let span = x_max - x_min; let offset = x_min; let scale = ((1 << bits) - 1) as
f32; ((x - offset) * scale / span) as u32 } pub fn to_le_bytes(self, mapping:
&MitTargetMapping) -> [u8; 8] { // Clamp values to mapping range so overflowing values
are wrapped to min/max let position =
self .position .clamp(mapping.position_min, mapping.position_max);
let velocity = self .velocity .clamp(mapping.velocity_min,
mapping.velocity_max); let torque = self.torque.clamp(mapping.torque_min,
mapping.torque_max); let kp = self.kp.clamp(mapping.kp_min, mapping.kp_max);
let kd = self.kd.clamp(mapping.kd_min, mapping.kd_max); let pos =
Self::float_to_uint(position, mapping.position_min, mapping.position_max, 16); let
vel = Self::float_to_uint(velocity, mapping.velocity_min, mapping.velocity_max, 12);
let torque_u = Self::float_to_uint(torque, mapping.torque_min, mapping.torque_max, 12);
let kp_u = Self::float_to_uint(kp, mapping.kp_min, mapping.kp_max, 12); let kd_u =
Self::float_to_uint(kd, mapping.kd_min, mapping.kd_max, 12); let lower_u32 = torque_u
```

```
| (kd_u << 12) | ((kp_u & 0xFF) << 24);        let upper_u32 = (kp_u >> 8) | (vel << 4) | (pos << 16);        let lower = lower_u32.to_le_bytes();        let upper = upper_u32.to_le_bytes();        [            lower[0], lower[1], lower[2], lower[3], upper[0], upper[1], upper[2], upper[3],        ]    }
```

Equivalent C example (same logic as above; 8 bytes little-endian):

```
#include <stdint.h>typedef struct {    float position;    float velocity;    float torque;    float kp;    float kd;} MitTarget;typedef struct {    float position_min;    float position_max;    float velocity_min;    float velocity_max;    float torque_min;    float torque_max;    float kp_min;    float kp_max;    float kd_min;    float kd_max;} MitTargetMapping;static MitTargetMapping mit_target_mapping_default(void) {    MitTargetMapping m = {        .position_min = -0.5f,        .position_max = 0.5f,        .velocity_min = -10.0f,        .velocity_max = 10.0f,        .torque_min = -10.0f,        .torque_max = 10.0f,        .kp_min = 0.0f,        .kp_max = 100.0f,        .kd_min = 0.0f,        .kd_max = 20.0f,    };    return m;}static float clampf(float x, float lo, float hi) {    if (x < lo) {        return lo;    }    if (x > hi) {        return hi;    }    return x;}static uint32_t float_to_uint(float x, float x_min, float x_max, uint32_t bits) {    float span = x_max - x_min;    float offset = x_min;    float scale = (float)((1u << bits) - 1u);    return (uint32_t)(((x - offset) * scale) / span);}static void store_u32_le(uint8_t dst[4], uint32_t v) {    dst[0] = (uint8_t)(v & 0xFFu);    dst[1] = (uint8_t)((v >> 8) & 0xFFu);    dst[2] = (uint8_t)((v >> 16) & 0xFFu);    dst[3] = (uint8_t)((v >> 24) & 0xFFu);}/* Pack MIT target into 8 bytes (lower_u32 then upper_u32, both little-endian). */void mit_target_to_le_bytes(const MitTarget *self, const MitTargetMapping *mapping,                            uint8_t out[8]) {    float position = clampf(self->position, mapping->position_min, mapping->position_max);    float velocity = clampf(self->velocity, mapping->velocity_min, mapping->velocity_max);    float torque = clampf(self->torque, mapping->torque_min, mapping->torque_max);    float kp = clampf(self->kp, mapping->kp_min, mapping->kp_max);    float kd = clampf(self->kd, mapping->kd_min, mapping->kd_max);    uint32_t pos = float_to_uint(position, mapping->position_min, mapping->position_max, 16);    uint32_t vel = float_to_uint(velocity, mapping->velocity_min, mapping->velocity_max, 12);    uint32_t torque_u = float_to_uint(torque, mapping->torque_min, mapping->torque_max, 12);    uint32_t kp_u = float_to_uint(kp, mapping->kp_min, mapping->kp_max, 12);    uint32_t kd_u = float_to_uint(kd, mapping->kd_min, mapping->kd_max, 12);    uint32_t lower_u32 = torque_u | (kd_u << 12) | ((kp_u & 0xFFu) << 24);    uint32_t upper_u32 = (kp_u >> 8) | (vel << 4) | (pos << 16);    store_u32_le(out, lower_u32);    store_u32_le(out + 4, upper_u32);}
```

Motor Control

Control Command Timeout

CANOpen has a built-in timeout mechanism. It is disabled by default, and you need to enable it manually. Please refer to the introduction of object 1016h for details.

Motor Disable Brake

By adjusting object 2040h 00h, you can set whether the motor windings are shorted to maintain braking state after the motor is disabled or a fault occurs.

One-to-Many Control

In CANOpen, PDO is not limited to using specified COB-IDs, but can be freely set. This means you can achieve one-to-many functionality by setting the COB-IDs of RPDO of multiple machines to the same value.

Does it sound abstract? Let's take four motors using force-position hybrid control as an example.

The CANOpenIDs of two motors are 0x01 and 0x02 respectively. Each motor controls their 6072h (Max Torque) and 60FFh (Target Velocity). Set the COB-ID of RPDO1 of both motors to the COB-ID of TPDO1 of node 0x10.

Then each motor has two valid objects. Other bytes can be filled. We assume 6072h is placed first, so for motor one, the mapping should be: `60FFh(4bytes)|6072h(2bytes)|fill(4bytes)|fill(4bytes)|fill(4bytes)|fill(4bytes)|fill(2bytes)`. Since there are 4 motors in total, 4 * (4 + 2) = 24 bytes are needed to control all motors in one frame, so 18 bytes of fill objects need to be placed after.

Similarly for motor two, it should be: `fill(4bytes)|fill(2bytes)|60FFh(4bytes)|6072h(2bytes)|fill(4bytes)|fill(4bytes)|fill(4bytes)`. And so on.

Save PDO Mapping

PDO Mapping can be saved to avoid re-setting every time on startup. Please refer to object 1010h for details. After saving, you only need to send NMT commands on each startup to directly start control.

MIT Control and Velocity Control

For `Powered Castor Wheel`, being pushable by humans is basically a must. In many cases, you want the chassis to be freely pushable under specific

conditions. At this time, you only need to limit the motor's maximum output torque to 0. All of this only requires using velocity control. But if you want to add some force control, such as dynamic/static friction compensation, etc., you can only use MIT control mode.

It's also worth mentioning that there's no restriction that all motors must use the same mode, so you can freely combine velocity control and MIT control, as long as you ensure your PDO mapping is normal.

Velocity Mode:

- Short data, effectively reduces CAN bus occupancy.
- Combined with 6072h, can achieve force-position hybrid control, allowing the chassis to be pushed. In force-position hybrid control, a single motor requires 6 bytes, and 64 bytes can easily drive 4 steering wheel modules (8 motors)
- Cannot achieve force control

MIT Control Mode:

- Longer data, will significantly increase CAN bus occupancy. If CAN bus signal quality is too poor, it will cause communication failure.
- No need to use with 6072h, just set Kd to 0 directly.
- Can achieve force control, such as dynamic/static friction compensation, etc.
- Length can be 16 bytes or 10 bytes depending on the situation. In either case, 2 COB-IDs are needed to drive all 4 steering wheel modules. Nevertheless, using 10-byte length can effectively reduce CAN bus occupancy, so it is still recommended.

About MIT Control

A single MIT object is 16 bytes long, of which 4 bytes of target position and 2 bytes of Kp, if not used, it is recommended that you directly remove them from the mapping. This can further reduce CAN bus occupancy.

Stall Error

When the motor continuously outputs at peak torque, it will trigger a stall error. If you want to completely avoid stall errors, you need to limit the motor's maximum output torque to 80% of peak torque. That is, write 800 to 6072h.

Control Testing

Please Operate with Caution

Before starting motor control, please ensure the surrounding environment is safe. Since repeatedly sending heartbeat packets for a node in the terminal is not particularly convenient, both examples provided here do not use heartbeat timeout detection.

SDO Control Testing

About SDO Control Testing

In fact, you should use PDO to write objects such as 60FFh, but here we first use SDO for CAN communication demonstration.

Please Operate with Caution

Before starting motor control, please ensure the surrounding environment is safe

Before starting, we power on the motor again to ensure the motor is in its initial state. Assume the motors to be controlled are 0x01 and 0x02. We will use velocity mode for demonstration. Motor 1's torque will be limited to 5% of peak torque, and motor 2 will be limited to 80% of peak torque.

```
1 w 0x6060 0 i8 32 w 0x6060 0 i8 31 w 0x60FF 0 i32 02 w 0x60FF 0 i32 01 w 0x6040 0 u16 6 2
w 0x6040 0 u16 6 1 w 0x6072 0 u16 502 w 0x6072 0 u16 800
```

After all the above writes are completed, write control word 7

```
1 w 0x6040 0 u16 72 w 0x6040 0 u16 7
```

Finally, write 0x0f

```
1 w 0x6040 0 u16 0x0f2 w 0x6040 0 u16 0x0f
```

At this point, the motor is enabled, with a target of 0 Rev/s. Write the speed in Float32 rev/s units. Here we prepare to test at 0.1 Rev/s.

```
1 w 0x60FF 0 r32 0.12 w 0x60FF 0 r32 0.1
```

At this point, both motors rotate at 0.1 Rev/s. Because their max torque limits differ, a small opposing torque stops one motor sooner, while the other needs a larger torque to stop.

```
1 w 0x6060 00 i8 3
[0] OK
2 w 0x6060 00 i8 3
[0] OK
1 w 0x60ff 00 r32 0.0
[0] OK
2 w 0x60ff 00 r32 0.0
[0] OK
1 w 0x6040 00 u16 6
[0] OK
2 w 0x6040 00 u16 6
[0] OK
1 w 0x6072 00 u16 50
[0] OK
2 w 0x6072 00 u16 800
[0] OK
1 w 0x6040 00 u16 7
[0] OK
2 w 0x6040 00 u16 7
[0] OK
1 w 0x6040 00 u16 0x0f
[0] OK
2 w 0x6040 00 u16 0x0f
[0] OK
1 w 0x60ff 00 r32 0.1
[0] OK
2 w 0x60ff 00 r32 0.1
[0] OK
```

PDO Control Testing

Obviously, using SDO communication wastes bandwidth significantly. PDO should be used for control whenever possible, but as mentioned earlier, PDO mapping needs to be adjusted according to the actual number of motors being operated. Therefore, this demo still only demonstrates controlling 4 motors. For other cases, please adjust according to the actual situation.

Adjust PDO Mapping

To achieve one-frame control of multiple motors, the original RPDO COB-ID cannot meet the requirements. Therefore, not only do we need to adjust the Mapping of all controlled motors, but also adjust the RPDO COB-ID.

Taking velocity control with torque limit as an example, each motor needs to use 6072h (Max Torque) and 60FFh (Target Velocity). Therefore, each motor needs six bytes. There are 4 motors in total, so 24 bytes are needed.

For the sending program, the order of all 24 bytes is as follows:

4-byte fill object is 3000h 03h, 2-byte fill object is 3000h 02h

Motor 1 Max Torque (2 Bytes) | **Motor 1 Target Velocity (4 Bytes)** | **Motor 2 Max Torque (2 Bytes)** | **Motor 2 Target Velocity (4 Bytes)** | **Motor 3 Max Torque (2 Bytes)** | **Motor 3 Target Velocity (4 Bytes)** | **Motor 4 Max Torque (2 Bytes)** | **Motor 4 Target Velocity (4 Bytes)**

For **Motor 1**, its RPDO Mapping should be:

Max Torque 6072h | **Target Velocity 60FFh** | 4-byte fill | 4-byte fill | 4-byte fill | 4-byte fill | 2-byte fill

For **Motor 2**, its RPDO Mapping should be:

4-byte fill | 2-byte fill | **Max Torque 6072h** | **Target Velocity 60FFh** | 4-byte fill | 4-byte fill | 4-byte fill

For **Motor 3**, its RPDO Mapping should be:

4-byte fill | 4-byte fill | 4-byte fill | **Max Torque 6072h** | **Target Velocity 60FFh** | 4-byte fill | 2-byte fill

For **Motor 4**, its RPDO Mapping should be:

4-byte fill | 4-byte fill | 4-byte fill | 4-byte fill | 2-byte fill | **Max Torque 6072h** | **Target Velocity 60FFh**

About Fill Objects

There's no need to stick to this form. Thanks to the flexible nature of PDO, you can arrange it freely. If the target velocity actually needs higher update

frequency while max torque doesn't, you can completely put all max torques into RPDO2, sending target velocity at high frequency and max torque at low frequency.

There's also no need to force all motors to be in the same control mode. It's perfectly fine to have some motors in velocity control mode and some using MIT control mode. As long as the Mapping is correct.

About Configuration Automation and Persistence

Remember, essentially canopend just sends some CAN messages. You can completely automate the PDO configuration process. For the specific format of the SDO protocol, please refer to the abundant internet tutorials.

In addition, these configurations will revert to default settings after each power-on unless saved through object 1010h. If you confirm that these configurations won't change, just save them through object 1010h to achieve configuration persistence. Subsequent power-ons won't require reconfiguration, only sending NMT commands and selecting modes, operating control words.

Here we need to choose a COB-ID as the RPDO1 COB-ID for all motors. Here we choose the TPDO1 COB-ID of node 0x10 (i.e., 0x190). You can choose any CAN-ID, as long as you ensure it won't conflict with other nodes. It's recommended to choose the master station's TPDO1, 2, 3, or 4.

- First write `0x8000_0000 | 0x190` to 1400h 01h to disable RPDO1
- Set 1400h 02h transmission type to 255
- Set 1600 00h RPDO valid mapping count to 0
- Set 1600 01h application object 1 to 4-byte fill 3000h 03h
- Set 1600 02h application object 2 to 2-byte fill 3000h 02h
- Set 1600 03h application object 3 to Max Torque 6072h 00h
- Set 1600 04h application object 4 to Target Velocity 60FFh 00h
- Set 1600 05h application object 5 to 4-byte fill 3000h 03h
- Set 1600 06h application object 6 to 4-byte fill 3000h 03h
- Set 1600 07h application object 7 to 4-byte fill 3000h 03h
- Set 1600 00h valid mapping count to 7
- Finally write `0x0000_0000 | 0x190` to 1400h 01h to enable RPDO1

At this point, RPDO configuration is complete. Next, enter `2 start` to send NMT commands, putting the motor into CiA301 Operational state.

But at this point, control via PDO is still not possible because the CiA402 state machine hasn't been configured yet. Since this only needs to be configured once on power-up, we'll also use SDO for demonstration.

- Write `0x80` to 6040h 00h to clear errors
- Write `0x03` to 6060h 00h to set motor to velocity control mode
- Write `0x06` to 6040h 00h Shutdown
- Write `0x07` to 6040h 00h Switch On
- Write `0x0f` to 6040h 00h Operation Enable

```
root@hexfellow-3eab8adae2dc9b17 ~# ./canopend can0 -i 0x2f -c "stdio"
./canopend[973]: CANopen device, Node ID = 0x00, starting
./canopend[973]: CANopen command interface on "standard IO" started
./canopend[973]: CAN Interface "can0" RX buffer set to 477 messages (212992 Bytes)
./canopend[973]: CANopen NMT state changed to: "initializing" (0)
./canopend[973]: CANopen device, Node ID = 0x2F, communication reset
./canopend[973]: CANopen device, Node ID = 0x2F, running ...
./canopend[973]: CANopen NMT state changed to: "pre-operational" (127)
./canopend[973]: CANopen Emergency message from node 0x2F: errorCode=0x5000, errorRegister=0x01, errorBit=0x2F, infoCode=0x00000004
2 w 0x1400 01 u32 0x80000190
[0] OK
2 w 0x1400 02 u8 255
[0] OK
2 w 0x1600 00 u8 0
[0] OK
2 w 0x1600 01 u32 0x30000320
[0] OK
2 w 0x1600 02 u32 0x30000210
[0] OK
2 w 0x1600 03 u32 0x60720010
[0] OK
2 w 0x1600 04 u32 0x60ff0020
[0] OK
2 w 0x1600 05 u32 0x30000320
[0] OK
2 w 0x1600 06 u32 0x30000320
[0] OK
2 w 0x1600 07 u32 0x30000320
[0] OK
2 w 0x1600 00 u8 7
[0] OK
2 w 0x1400 01 u32 0x190
[0] OK
2 start
[0] OK
2 w 0x6040 00 u16 0x80
[0] OK
2 w 0x6060 00 i8 3
[0] OK
2 w 0x6040 00 u16 0x06
[0] OK
2 w 0x6040 00 u16 0x07
[0] OK
2 w 0x6040 00 u16 0x0f
[0] OK
```

At this point, the motor can start to be controlled via PDO. Below is an example of limiting all motors' max torque to 80% and target speed to 1 Rev/s.

```
cansend can0 190##1.20030000803F20030000803F20030000803F20030000803F
```

If all motors are correctly configured, all motors will start running.

Object Dictionary

> Only provide parts different from CiA301 and CiA402. Due to distribution issues, we cannot provide CiA301.pdf and CiA402.pdf. Please use a search engine to find relevant content.

Software Version

By reading 1018h 03h we can obtain the motor firmware version. Object dictionaries may differ between firmware versions, please note.

Extra Readings

- hex-motor-control-test — full C control demos (MIT / profile velocity)
- Hex Motor Example

Version Change History

- 0x07
  - First publicly released version
- 0x08
  - Changed floating-point format from Q21 to Float32
  - Added compact MIT control parameters

0x07

Object Dictionary for Firmware Version 07

0x08

Object Dictionary for Firmware Version 08