Application Note

基于 PCIe 主站卡 C_C++搭建 IO 及电机控制应用

E 系列高速 IO 模块

AN01010101 1.00 Date:2025/11/18

类别	内容
关键词	集中式远程 IO 从站、集中式步进电机驱动从站、 PCIe 转 EtherCAT 通讯卡
摘要	致远主站系列产品及 E 系列高速 IO 联调示例

修订历史

版本	日期	原因
V1.00	2025/11/18	创建文档

目 录

1. 适用范围

本文介绍了致远电子开发的 PCIe-EtherCAT 主站卡与 E 系列集中式高速 IO 从站联调，搭建 IO 及电机控制的示例。

从站产品包括 E 系列集中式数字输入模块、数字输出模块以及步进电机控制模块等。

2. 基于 PCIe-EtherCAT 通讯卡 C/C++的搭建

PCIe-EtherCAT 通讯卡提供了 C/C++接口的 SDK 动态库。用户只需要调用动态库提供的接口编写程序，就可以控制板卡运行 EtherCAT 主站。在主站程序中操作 PDO 数据、请求 SDO 读写命令，就可以控制从站设备，实现电机运动。

2.1 测试环境

主站设备：

- HPCIe-2E 通讯卡

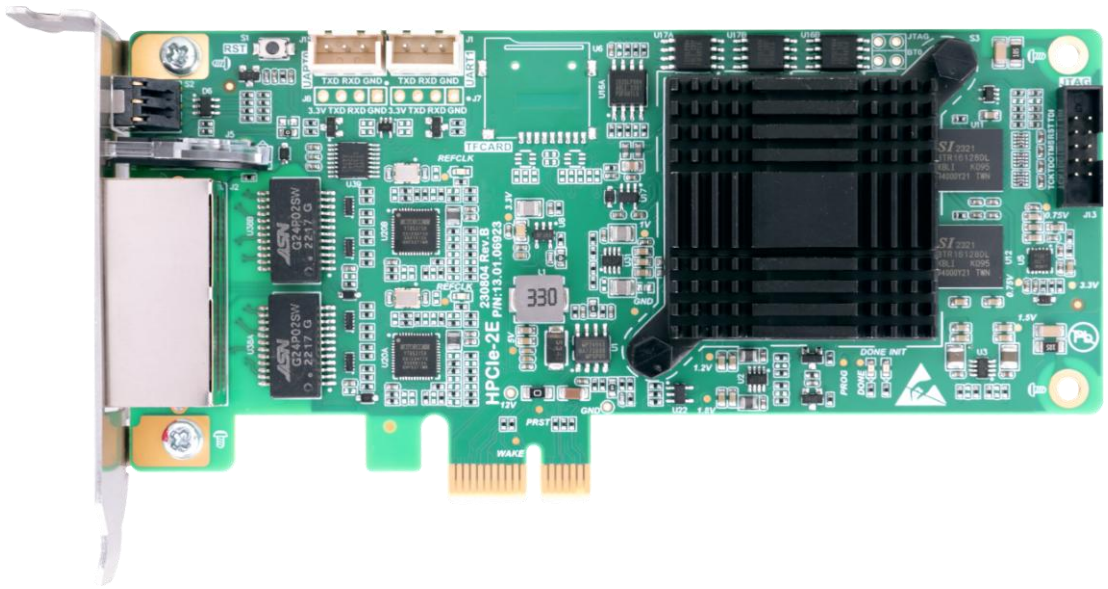

图 2.1 HPCIe-2E 主站通讯卡

从站设备:

- ZPT-8080 耦合器
- ZDM-E0016N 数字输出模块
- ZDM-E1600N 数字输入模块
- ZMTI-EF1200 步进电机驱动模块

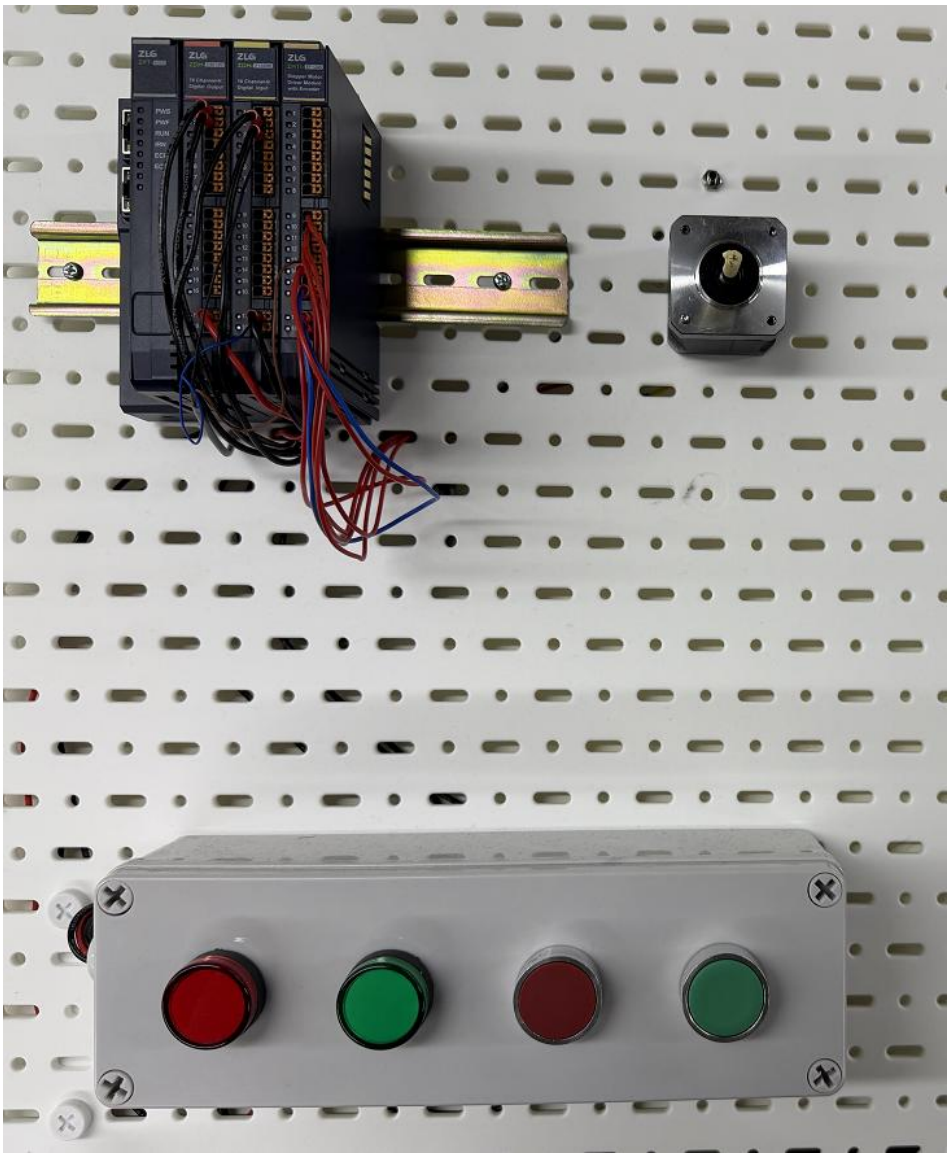

图 2.2　从站设备及其传感器执行器

其他硬件：

- 带 PCIe 卡槽的开发主机（Windows 或 Linux 均可）
- 配置主机（Windows，可以同时作为配置和开发主机）
- 步进电机
- 指示灯（红、绿）
- 控制按钮（红、绿）
- 接近传感器
- 网线若干
- 24V 电源

软件环境：

- PCIe-EtherCAT 通讯卡的驱动程序，安装在开发主机上
- AWStudio 运动控制版 配置上位机
- Visual Studio 2015（Windows）
- gcc cmake make 等编译器和构建工具（Linux）

2.2 设备连接

将 HPCIe-2E 通讯卡插入到开发主机的 PCIe 卡槽中，开机，安装板卡驱动。

用网线连接 HPCIe-2E 通讯卡的 ECAT 口和 ZPT-8080 耦合器的 IN 口，将剩余模块顺次通过背板总线连接到耦合器。24V 电源分别 ZPT-8080 耦合器。

指示灯的信号从 ZDM-E0016N 数字输出模块的输出引脚引出。按钮的信号接入到 ZDM-E1600N 数字输入模块的输入引脚。电机的驱动线按次序接到 ZMTI-EF1200 的 9~12 的端口，如果有编码器信号也接到 ZMTI-EF1200 上。此外，ZMTI-EF1200 还需要额外的 1 路电源输入作为电机驱动电源。

2.3 运行效果

启动程序后，主站切换到 Operation 操作状态。电机系统进入停止状态未启动，绿灯闪烁。

按绿色运行键之后，电机开始转动，绿灯常亮。按红色停止键，电机停止转动，绿灯恢复闪烁。

用任意金属物体靠近传感器，电机系统进入紧急状态，电机停止转动，红灯常亮。按绿色运行键后，电机系统恢复运行。

2.4 ENI 文件生成

本节介绍通过 AWStudio 运动控制版配置上位机生成 ENI 文件。

2.4.1 新建解决方案

AWStudio 启动后默认没有解决方案，需要先新建或打开一个解决方案。

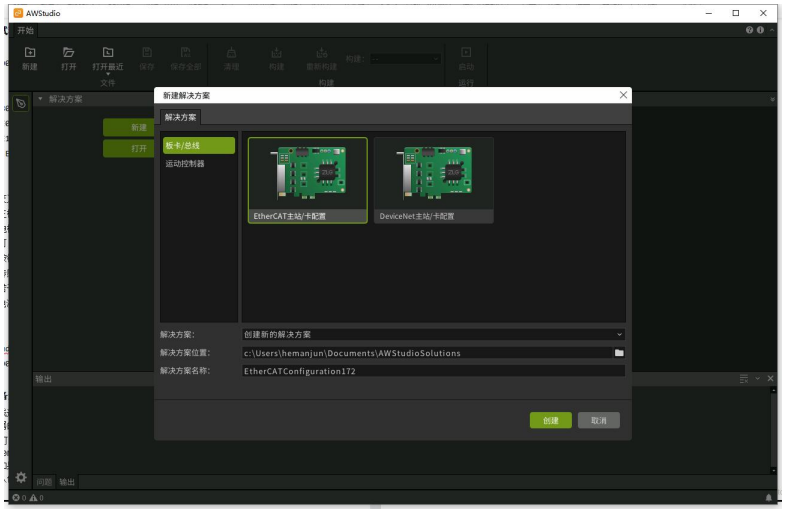

图 2.3 新建 EtherCAT 解决方案

2.4.2 添加从站

添加从站可以通过在线扫描或手动添加的方式。

图 2.4　拓扑扫描

添加从站后如图 2.5。

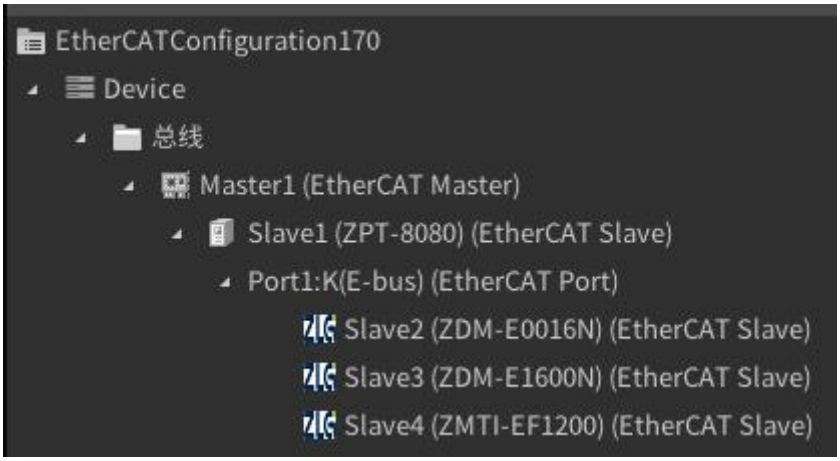

图 2.5　EtherCAT 网络拓扑

2.4.3 配置电机的 PDO 过程数据

双击从站 ZMTI-EF1200，打开“从站配置-变量页”，可以查看厂商 ESI 默认配置下的 PDO 变量，如图。默认变量不一定能满足我们的控制需求。ZMTC-EF1200 默认是 CSP 模式的过程变量。有需要时可以自行配置电机的 PDO 变量。

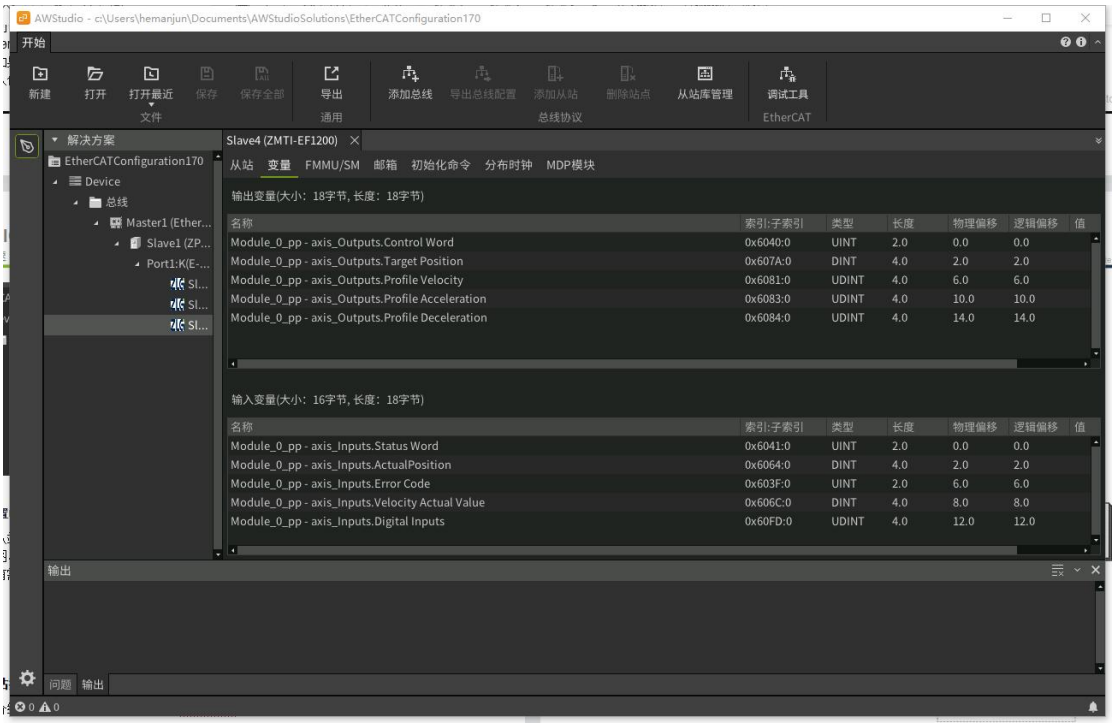

图 2.6　从站的 PDO 过程数据

打开“从站配置-MDP 模块”，可以看到画面分为两个区域，左边“插槽配置”为当前启用的配置项，右侧模块库为我们预定义的可用模块，可以看到是各模块是按 CiA402 电机控制模式去区分的。默认为 CSP 模式。

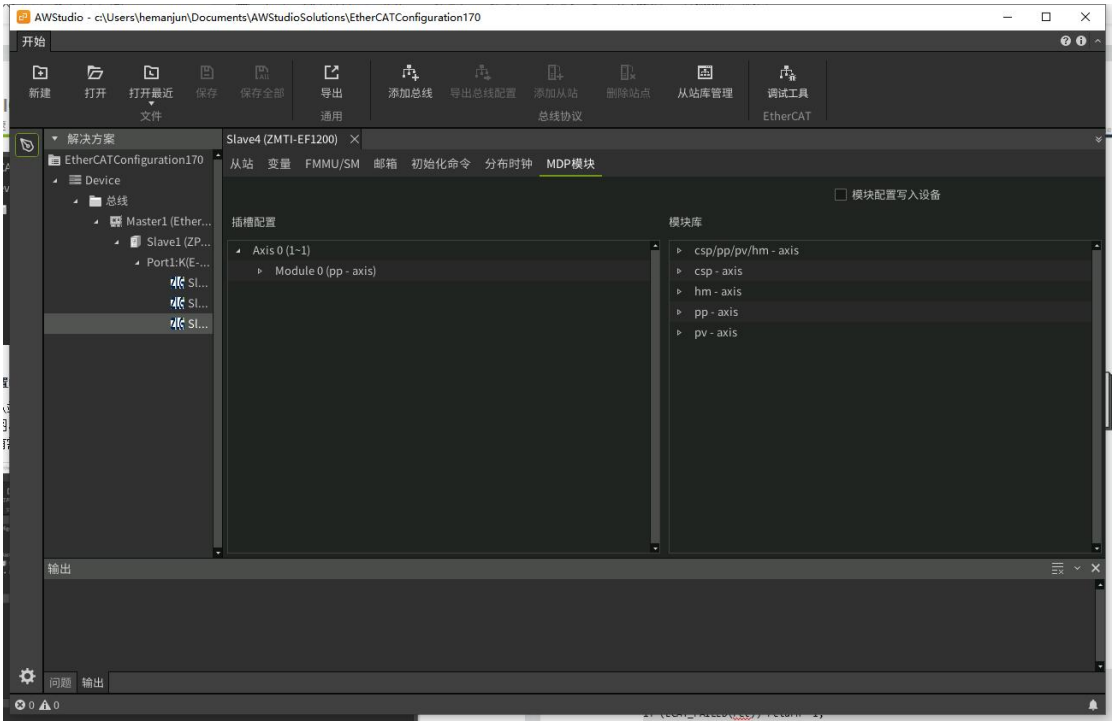

图 2.7　MDP 模块配置

2.4.4 导出 ENI

同样配置其他主站和从站参数，完成后导出 ENI 文件。

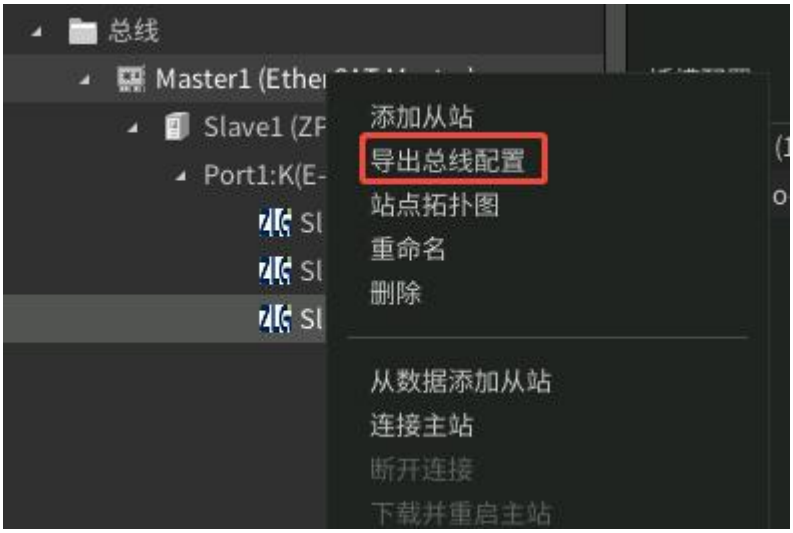

图 2.8　导出总线配置到 ENI 文件

2.5　过程数据 PDO

根据主站配置和网络拓扑结构，我们需要定义当前拓扑下使用的 PDO 数据结构。

打开“主站配置-变量”页，查看当前网络拓扑下的数据结构。

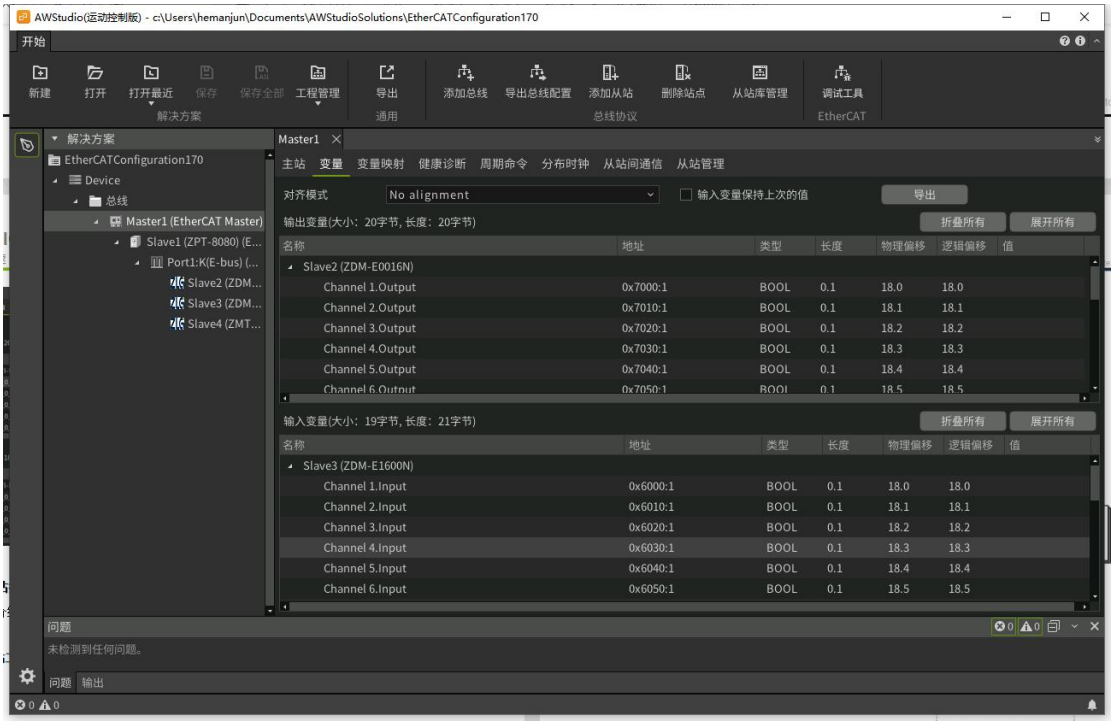

图 2.9　EtherCAT 总线拓扑的 PDO 过程数据

结合从站手册的定义，可以定义为以下的结构。

```
////////////////////////////////////////////////////////////////////////////////
// 过程映像
#pragma pack(push, 1)

typedef union {
    struct {
        uint16_t switch_on: 1;
        uint16_t enable_voltage: 1;
        uint16_t quick_stop: 1;
        uint16_t enable_operation: 1;
        uint16_t pad0: 3;
        uint16_t fault_reset: 1;
        uint16_t halt: 1;
        uint16_t pad1: 1;
        uint16_t pad2: 6;
    } b;
    uint16_t v;
} servo_ctrl_word;

typedef union {
    struct {
        uint16_t ready_to_switch_on: 1;
        uint16_t switched_on: 1;
        uint16_t operation_enabled: 1;
        uint16_t fault: 1;
        uint16_t voltage_enabled: 1;
        uint16_t quick_stop: 1;
        uint16_t switch_on_disabled: 1;
        uint16_t warning: 1;
        uint16_t pad0: 1;
        uint16_t remote: 1;
        uint16_t target_reached: 1;
        uint16_t internal_limit_active: 1;
        uint16_t pad1: 4;
    } b;
    uint16_t v;
} servo_stat_word;
```

```
typedef struct {
    servo_ctrl_word ctrl;
    int32_t p;
    uint8_t pad[6];
} servo_ctrl;
typedef struct {
    servo_stat_word stat;
    int32_t p;
    uint16_t error;
    uint32_t inputs;
} servo_stat;

typedef struct {
    servo_ctrl servos[NUM_SERVOS];
    uint16_t outputs[NUM_DO];
} proc_img_o;

typedef struct {
    servo_stat servos[NUM_SERVOS];
    uint16_t inputs[NUM_DI];
} proc_img_i;

#pragma pack(pop)
```

2.6 主站控制

本节介绍如何调用 PCIe 主站卡的 SDK 接口来控制 EtherCAT 总线。本示例电机采用 CSP 模式控制，因此可以用主站的同步队列接口交换数据。同步队列的通信原理如图。

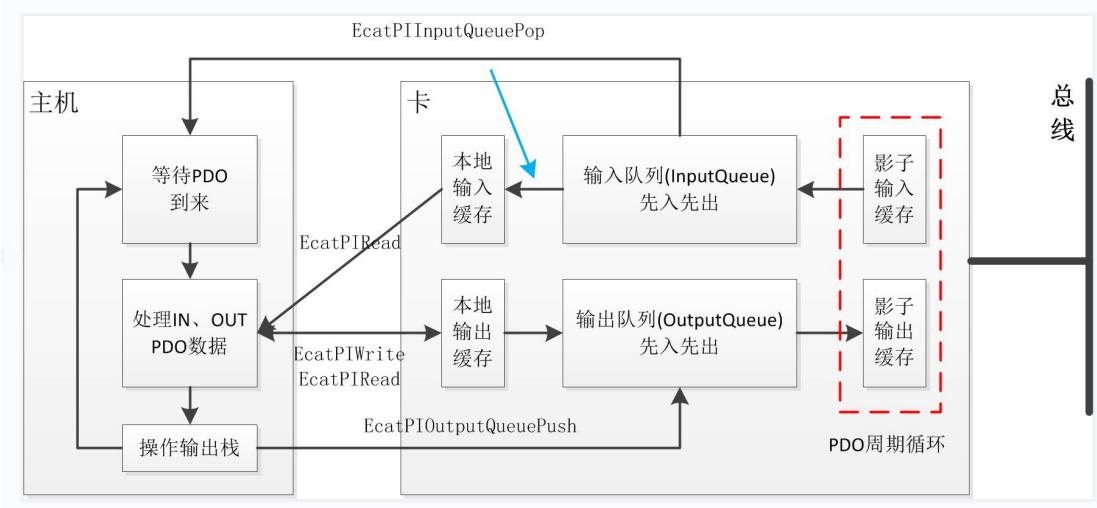

图 2.10 同步队列通信原理

2.6.1 初始化主站

首先启动 EtherCAT，其中 ALIAS 需要填入主站卡的别名编码器值，“protal3.xml”需要替换为用户通过 AWStudio 导出到 ENI.xml 文件。

```
//PCIe-2E 只有一个通道,PCIe-4E 有两个通道
RETURN_IF_FAILED(EcatOpen(&ecat_dev, BOARD_ALIAS(index_buff, index), channel));
RETURN_IF_FAILED(EcatBusRun(ecat_dev, "protal3.xml"));
```

2.6.2 EtherCAT 状态切换

请求 EtherCAT 总线切换状态。

```
// 请求切换状态
uint8_t state;
RETURN_IF_FAILED(EcatRequestMasterState(ecat_dev, EcatStateO));
for (uint32_t i = 0; ; i++)
{
    RETURN_IF_FAILED(EcatGetMasterState(ecat_dev, &state));
    _INF_("request is: 8, curr is: %d", state);
    usleep(500*1000);
    if (state == EcatStateO) break;
}
```

2.6.3 获取 PI 镜像指针

读写 PDO 变量，首先需要通过接口获取 PI 镜像。为了加快用户对输入输出缓存操作的速度，可使用内存映射的方式来直接操作缓存，以减少 IO 的操作时间。然后通过 pi_get_input 和 pi_get_output 获取变量指针。

```
// 清空并预计填入数据
proc_img_i *pi;
proc_img_o *po;
RETURN_IF_FAILED(EcatPINMap(ecat_dev, PI_AREA_LOCAL_INPUT, (void **)&pi));
 _INF_("EcatPINMap done");

RETURN_IF_FAILED(EcatPINMap(ecat_dev, PI_AREA_LOCAL_OUTPUT, (void **)&po));
 _INF_("EcatPINMap done");

 _INF_("pi: 0x%llx", pi);
 _INF_("po: 0x%llx", po);
```

2.6.4 使能 PI 总线数据交换

在控制电机和初始化电机状态之前，需要先使能 PI 镜像交换

```
RETURN_IF_FAILED(EcatPIEnable(ecat_dev));
```

2.6.5 预填充 PDO 队列

为了减少用户逻辑和系统抖动对 EtherCAT 总线造成的影响，可以向 PDO 队列里预填充几帧数据作为缓冲。

```
for (uint32_t i = 0; i < 2; i++)
{
    EcatPIOutputQueuePush(ecat_dev, false, 100);
}
```

2.6.6 EtherCAT 周期处理主循环

主循环中，EcatPIInputQueuePop 等待队列接收可用数据。用户逻辑更新数据后，通过 EcatPIOutputQueuePush 函数将数据填入队列中。

```
for (uint32_t i = 0; i < 2; i++)
{
    EcatPIOutputQueuePush(ecat_dev, false, 100);
}
_INF_("EcatPINMap done");
RETURN_IF_FAILED(EcatPIEnable(ecat_dev));
while (1)
{
    while (EcatPIInputQueuePop(ecat_dev, false, 100) != 0);
    // 用户逻辑
    if (EcatPIOutputQueuePush(ecat_dev, 0, 100) != 0)
    {
        _ERR_("push pi output error");
        break;
    }
}
```

2.6.7 退出

当应用程序需要退出时，需要销毁 ECAT 主站。

```
// 退出
RETURN_IF_FAILED(EcatBusStop(ecat_dev));
RETURN_IF_FAILED(EcatClose(ecat_dev));
```

2.7 按键及指示灯控制

该应用中包括运行键（绿）、停止键（红）、运行状态灯（绿）、紧急状态灯（红）、接近传感器等外设，这些外设功能如下。

1. 运行状态灯用于指示电机运行，停止状态下为绿灯闪烁，运行状态下为绿灯常亮，紧急状态下绿灯熄灭。

2. 紧急状态灯用于指示系统进入紧急状态。正常状态下红灯熄灭。紧急状态下红灯常亮，绿灯熄灭。

3. 运行键用于启动电机，进入运行状态。停止键用于停止电机运行，进入停止状态。

4. 接近传感器检测到金属物体接近时，会输出低电平。此时系统需要进入紧急状态。进入紧急状态后，必须用运行键重启电机系统。

以上控制逻辑实现在 EtherCAT 控制程序的主循环中。

```
while (1)
{
    while (EcatPIInputQueuePop(ecat_dev, false, 100) != 0);

    // 绿色按钮按下时，才启动
    btn_run = !((pi->inputs[0] >> BTN_RUN) & 0x1); // 输入低有效
    if(btn_run) {
        run_cmd = 1;
        stop_cmd = 0;
        halt_cmd = 0;
    }
    // 红色按钮按下，电机停止运行
    btn_stop = !((pi->inputs[0] >> BTN_STOP) & 0x1); // 输入低有效
    if(btn_stop) {
        stop_cmd = 1;
        run_cmd = 0;
    }
    else {
        stop_cmd = 0;
    }

    // 接近传感器
    btn_close = !((pi->inputs[0] >> BTN_CLOSE) & 0x1); // 输入低有效
    if(btn_close) {
        halt_cmd = 1;
        run_cmd = 0;
    }

    // 绿灯状态
    if(run_cmd) {
        // 常亮
        po->outputs[0] |= (1 << LED_GREEN);
    }
    else {
        // 闪烁 2Hz
        if (loops % 500 == 0) {
            po->outputs[0] ^= (1 << LED_GREEN);
        }
    }
```

```
        if(halt_cmd) {
            po->outputs[0] |= (1 << LED_RED);       // 红灯常亮
            po->outputs[0] &= ~(1 << LED_GREEN);    // 绿灯熄灭
        }
        else {
            po->outputs[0] &= ~(1 << LED_RED);      // 红灯熄灭
        }
        loops++;
        /** 电机控制*/
    }
```

2.8 电机控制

2.8.1 CiA402 状态机简介

以下为 CiA402 规范定义的状态机转换模型。

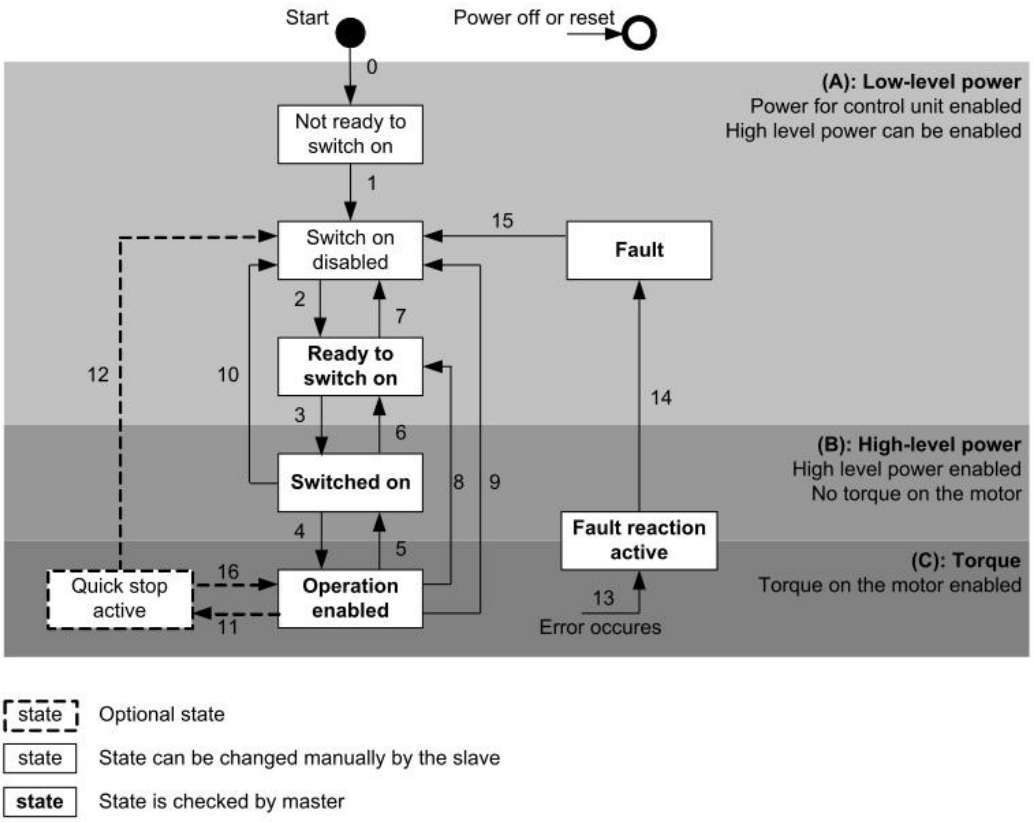

图 2.11 CiA402 状态机

CiA402 控制字用于发送状态转换请求，其位定义如下。

The bits of the *controlword* are defined as follows:

15 11	10 9	8	7	6 4	3	2	1	0
manufacturer specific	reserved	halt	Fault reset	Operation mode specific	Enable operation	Quick stop	Enable voltage	Switch on
O	O	O	M	O	M	M	M	M

MSB LSB

0 - Optional M - Mandatory

BITS 0 – 3 AND 7:

Device control commands are triggered by the following bit patterns in the *controlword*:

Command	Bit of the *controlword*					Transitions
	Fault reset	Enable operation	Quick stop	Enable voltage	Switch on	
Shutdown	0	X	1	1	0	2,6,8
Switch on	0	0	1	1	1	3*
Switch on	0	1	1	1	1	3**
Disable voltage	0	X	X	0	X	7,9,10,12
Quick stop	0	X	0	1	X	7,10,11
Disable operation	0	0	1	1	1	5
Enable operation	0	1	1	1	1	4,16
Fault reset	↑	X	X	X	X	15

Table 4: Device control commands **(bits marked X are irrelevant, * ... In the state SWITCHED ON the drive executes the functionality of this state., ** ... It exists no functionality in the state SWITCHED ON. The drive does not do any in this state.)**

图 2.12 CiA402 标准控制字定义

发送状态请求后，通过状态字来确认状态转换是否成功，成功后才能继续转换状态，状态字的位定义如下。

Bit	Description	M /O
0	Ready to switch on	M
1	Switched on	M
2	Operation enabled	M
3	Fault	M
4	Voltage enabled	M
5	Quick stop	M
6	Switch on disabled	M
7	Warning	O
8	Manufacturer specific	O
9	Remote	M
10	Target reached	M
11	Internal limit active	M
12 - 13	Operation mode specific	O
14 - 15	Manufacturer specific	O

Table 6: Bits in the *statusword*

图 2.13 CiA402 标准状态字定义

BITS 0 – 3, 5 AND 6:

The following bits indicate the status of the device:

Value (binary)	State
xxxx xxxx x0xx 0000	Not ready to switch on
xxxx xxxx x1xx 0000	Switch on disabled
xxxx xxxx x01x 0001	Ready to switch on
xxxx xxxx x01x 0011	Switched on
xxxx xxxx x01x 0111	Operation enabled
xxxx xxxx x00x 0111	Quick stop active
xxxx xxxx x0xx 1111	Fault reaction active
xxxx xxxx x0xx 1000	Fault

Table 7: Device state bits (x ... irrelevant for this state)

图 2.14 状态字判断

2.8.2 CiA402 运行模式&电机数量

```
////////////////////////////////////////////////////////////////////////////
// 全局配置参数
#define ECAT_ETH0           "zecm0"
#define RPC_SERVER_PORT     5000                // 开放与上位机通讯的端口号
#define NUM_SERVOS          1                   // 总电机数
#define NUM_DO              1
#define NUM_DI              1

#define LED_GREEN           0                   // 绿灯
#define LED_RED             1                   // 红灯

#define BTN_CLOSE           0                   // 接近传感器
#define BTN_RUN             1                   // 绿色按钮
#define BTN_STOP            2                   // 红色按钮

enum {
    OP_MODE_PP = 1,
    OP_MODE_PV = 3,
    OP_MODE_PT = 4,
    OP_MODE_HM = 6,
    OP_MODE_CSP = 8,
    OP_MODE_CSV = 9,
    OP_MODE_CST = 10,
};
```

```
uint32_t g_op_mode = OP_MODE_CSP;
uint32_t g_run_task = 1;
```

g_op_mode 设置电机运行模式，本示例为 CSP 模式。

2.8.3 SDO 命令设置 CiA402 电机运行模式

motor_mode_init 初始化运行模式，主要是通过 SDO 命令写邮箱 0x6060 对象，设置运行模式。

```
static int motor_config(ECAT_HANDLE ecat_dev)
{
    _INF_("motor_config Enter\r\n");
    ECAT_RESULT ret = ECAT_S_OK;
    uint32_t len;
    unsigned i;
    int cnt = 0;
    ecat_sdo_t items[10] = {
        { 0, 0x6060, 0x00, 1, &g_op_mode }, // 控制模式
    };
    for (uint16_t s = 0; s < NUM_SERVOS; s++)
    {
        _INF_("motor_config slave: %d\r\n", s + 1);

        for (i = 0; i < sizeof(items) / sizeof(ecat_sdo_t); i++)
        {
            _INF_("ecat_sdo_t: %d\r\n", i);
            ecat_sdo_t *item = &items[i];
            len = item->len;
            item->slave = s + 1;
            _INF_("ecat_sdo_t line: %d\r\n", __LINE__);
            ret = EcatCoeSDODownload(
                ecat_dev,
                item->slave,
                item->index,
                item->subindex,
                len,
                (void*)item->val,
                5000);
            _INF_("ecat_sdo_t line: %d\r\n", __LINE__);
            if (ret != ECAT_S_OK) {
                _DBG_("EcatCoeSDODownload failed: ret=%d, slave=%d, index=0x%04x,
subindex=0x%02x, value=0x%08x(%d)", ret, item->slave, item->index, item->subindex,
*(uint16_t*)item->val, *(uint16_t*)item->val);
                return ret;
            }
```

```
            _DBG_("EcatCoeSDODownload succeeded: slave=%d, index=0x%04x,
subindex=0x%02x, value=0x%08x(%d)", item->slave, item->index, item->subindex,
*(uint16_t*)item->val, *(uint16_t*)item->val);
        }
    }
    return ret;
}
```

2.8.4 CiA402 状态转换处理

CiA402 协议需要维护状态机的切换，在 motor_control 函数中实现，包括电机启动和错误响应及处理。

CiA402 状态机的转换，需要在 PDO 周期中运行。电机的启动、急停、错误检测和清除等，都依赖 CiA402 状态机实现。意味着 motor_control 需要在主循环中反复调用。

```
static bool motor_control(int axis, servo_stat_word *s0, const servo_stat_word *s1,
servo_ctrl_word *c, bool show)
{
    servos_t *servo = &__servos[axis];
    do {
        c->v = 0;
        c->b.enable_voltage = 1;
        c->b.quick_stop = 1;
        c->b.enable_operation = s1->b.switched_on || s1->b.operation_enabled;
        c->b.switch_on = s1->b.ready_to_switch_on;

        if (s1->b.fault && !servo->fault_reset_delay)
        {
            // 延时 1 秒再尝试错误恢复，防止伺服在错误和恢复中反复快速切换损坏电路
            _DBG_("axis %d: delay fault reset, ctrl:%d, stat:%x", axis, c->v,
(int)s1->v);
            servo->fault_reset_delay = 2000; // 2s(即 2000 个周期)
            servo->run_delay = 2000; // 当错误恢复成功伺服出力后，延时 2s 再开始加速(根据
抱闸动作时间适当调整)
            break;
        }
        // 伺服发生错误，且在延时等待处理中
        if (servo->fault_reset_delay)
        {
            servo->fault_reset_delay--;
            if (!servo->fault_reset_delay)
            {
                // 延时减到 0，尝试进行错误恢复
                _DBG_("axis %d: start fault reset", axis);
```

```
                c->b.fault_reset = 1;
            }
            break;
        }

    } while (0);

    s0->v = s1->v;

    return s1->b.operation_enabled;
}
```

2.8.5 EtherCAT 周期数据处理控制电机转动

控制电机往复运动的主要逻辑在 main 函数中循环实现。在 CSP 模式下，程序通过计算得到电机位置后，写到 PDO 变量中。然后主站卡在下一次周期将更新后的电机位置发送给从站，从站就能控制电机转动。

```
for (int i = 0; i < NUM_SERVOS; i++)
{
    if(halt_cmd | stop_cmd) {
        // 电机 halt
        po->servos[i].ctrl.b.halt = 1;
    }
    else {
        po->servos[i].ctrl.b.halt = 0;
    }

    do {
        servos_t *servo = &__servos[i];
        if(!run_cmd || (i > 0 && run[i-1] == 0)) {
            break;
        }

        run[i] = motor_control(i, &servo->ss, pi->servos[i].stat, &servo->sc, false);

        if (pi->servos[i].stat.b.fault) { // 检测到报错
            _DBG_("Motor fault, servo[%d] error code = 0x%04X", i, pi->servos[i].error);
        }

        if (loops % 1000 == 0)
        {
            _INF_("servo[%d]: p=%10d, ctrl:%08x stat=%08x", i, pi->servos[i].p,
po->servos[i].ctrl.v, pi->servos[i].stat.v);
        }
```

```
        // 伺服未出力，或已出力但延时未结束，应停止转动
        if (!run || servo->run_delay != 0) {
            servo->out_p = pi->servos[i].p;
            servo->zero_p = (float)servo->out_p;
            servo->offset_p = 0;
            servo->out_v = 0;
            servo->out_t = 0;
            servo->reverse = false;
        }

        // 伺服已出力，延时计数递减
        if (run && servo->run_delay)
            servo->run_delay--;

        // 伺服已出力，延时结束，进行正常控制
        if (run && !servo->run_delay) {

            switch (g_op_mode)
            {
            case OP_MODE_CSP:
                {
                    // 不同轴单位可能不一样，下表定义各轴位置范围及增量
                    I32 range_min[] = { -50000 };
                    I32 range_max[] = {  50000 };
                    float step_size[] = { 50.f };
                    if (servo->offset_p < range_min[0]) servo->reverse = false;
                    if (servo->offset_p > range_max[0]) servo->reverse = true;
                    servo->offset_p += (servo->reverse ? -1 : 1) * step_size[0];
                    servo->out_p = (I32)(servo->zero_p + servo->offset_p);
                }
                break;
            }
        }

        // 更新输出变量
        po->servos[i].ctrl = servo->sc;
        switch (g_op_mode)
        {
        case OP_MODE_CSP: po->servos[i].p = servo->out_p; break;
        }
    }while(0);
}
```

2.9 完整示例代码

```
#include <stdio.h>

#include "pci_errno.h"
#include "pci_zecm.h"
#include "pci_dbg.h"

#ifdef VXWORKS
#define main SYNC_CONTROL_TEST
#endif

#define RETURN_IF_FAILED(expr)      { int ret = (expr); if (ret != 0) { _ERR_(#expr "
failed: err=%d", ret); return -1; } else { _INF_(#expr " succeeded"); } }
```

```
/*************************motor config************************/
#define NUM_SERVOS  1                          // 总电机数
#define NUM_DO  1
#define NUM_DI  1

#define LED_GREEN   0   // 绿灯
#define LED_RED     1   // 红灯

#define BTN_CLOSE   0   // 接近传感器
#define BTN_RUN     1   // 绿色按钮
#define BTN_STOP    2   // 红色按钮

typedef  uint32_t   U32;
typedef  uint16_t   U16;
typedef  uint8_t    U8;
typedef  int32_t    I32;
typedef  int16_t    I16;

enum {
    OP_MODE_PP = 1,
    OP_MODE_PV = 3,
    OP_MODE_PT = 4,
    OP_MODE_HM = 6,
    OP_MODE_CSP = 8,
    OP_MODE_CSV = 9,
    OP_MODE_CST = 10,
};
```

```
static U32 g_op_mode = OP_MODE_CSP;
static U32 g_run_task = 1;

////////////////////////////////////////////////////////////////////////////////
// 过程映像
#pragma pack(push, 1)

typedef union {
    struct {
        uint16_t switch_on: 1;
        uint16_t enable_voltage: 1;
        uint16_t quick_stop: 1;
        uint16_t enable_operation: 1;
        uint16_t pad0: 3;
        uint16_t fault_reset: 1;
        uint16_t halt: 1;
        uint16_t pad1: 1;
        uint16_t pad2: 6;
    } b;
    uint16_t v;
} servo_ctrl_word;

typedef union {
    struct {
        uint16_t ready_to_switch_on: 1;
        uint16_t switched_on: 1;
        uint16_t operation_enabled: 1;
        uint16_t fault: 1;
        uint16_t voltage_enabled: 1;
        uint16_t quick_stop: 1;
        uint16_t switch_on_disabled: 1;
        uint16_t warning: 1;
        uint16_t pad0: 1;
        uint16_t remote: 1;
        uint16_t target_reached: 1;
        uint16_t internal_limit_active: 1;
        uint16_t pad1: 4;
    } b;
    uint16_t v;
} servo_stat_word;
```

```
typedef struct {
    servo_ctrl_word ctrl;
    int32_t p;
    uint8_t pad[6];
} servo_ctrl;

typedef struct {
    servo_stat_word stat;
    int32_t p;
    uint16_t error;
    uint32_t inputs;
} servo_stat;

typedef struct {
    servo_ctrl servos[NUM_SERVOS];
    uint16_t outputs[NUM_DO];
} proc_img_o;

typedef struct {
    servo_stat servos[NUM_SERVOS];
    uint16_t inputs[NUM_DI];
} proc_img_i;

#pragma pack(pop)

typedef struct {
    servo_ctrl_word sc;    // 新控制字
    servo_stat_word ss;    // 旧状态字
    int run_delay;         // 延时启动
    int fault_reset_delay; // 延时再错误恢复
    float zero_p;          // 零点位置
    float offset_p;        // 偏移输出位置
    I32 out_p;             // 输出位置
    I32 out_v;             // 输出转速
    I16 out_t;             // 输出扭矩
    I32 out_acc;             // 输出扭矩
    I32 out_dec;             // 输出扭矩
    bool reverse;          // 当前方向
}servos_t;

static servos_t __servos[NUM_SERVOS] = {0};
```

```
#ifdef _WIN32
#include <windows.h>
void usleep(int64_t usec){
    HANDLE timer;
    LARGE_INTEGER ft;
    ft.QuadPart = -(10*usec);
    timer = CreateWaitableTimer(NULL, TRUE, NULL);
    SetWaitableTimer(timer, &ft, 0, NULL, NULL, 0);
    WaitForSingleObject(timer, INFINITE);
    CloseHandle(timer);
}
#else
#include <unistd.h>
#endif

/************************ motor config end ***************************/
static bool motor_control(int axis, servo_stat_word *s0, const servo_stat_word s1,
servo_ctrl_word *c, bool show)
{
    servos_t *servo = &__servos[axis];
    do {
        c->v = 0;
        c->b.enable_voltage = 1;
        c->b.quick_stop = 1;
        c->b.enable_operation = s1.b.switched_on || s1.b.operation_enabled;
        c->b.switch_on = s1.b.ready_to_switch_on;

        if (s1.b.fault && !servo->fault_reset_delay)
        {
            // 延时 1 秒再尝试错误恢复，防止伺服在错误和恢复中反复快速切换损坏电路
            _DBG_("axis %d: delay fault reset, ctrl:%d, stat:%x", axis, c->v, (int)s1.v);
            servo->fault_reset_delay = 2000; // 2s(即 2000 个周期)
            servo->run_delay = 2000; // 当错误恢复成功伺服出力后，延时 2s 再开始加速(根据
抱闸动作时间适当调整)
            break;
        }
        // 伺服发生错误，且在延时等待处理中
        if (servo->fault_reset_delay)
        {
            servo->fault_reset_delay--;
            if (!servo->fault_reset_delay)
```

```
            {
                // 延时减到 0，尝试进行错误恢复
                _DBG_("axis %d: start fault reset", axis);
                c->b.fault_reset = 1;
            }
            break;
        }

    } while (0);

    s0->v = s1.v;

    return s1.b.operation_enabled;
}

typedef struct {
    int16_t slave;
    int16_t index;
    int8_t subindex;
    int32_t len;
    void* val;
} ecat_sdo_t;

static int motor_config(ECAT_HANDLE ecat_dev)
{
    _INF_("motor_config Enter\r\n");
    ECAT_RESULT ret = ECAT_S_OK;
    uint32_t len;
    unsigned i;
    int cnt = 0;
    ecat_sdo_t items[] = {
        { 0, 0x6060, 0x00, 1, &g_op_mode }, // 控制模式
    };
    for (uint16_t s = 0; s < NUM_SERVOS; s++)
    {
        _INF_("motor_config slave: %d\r\n", s + 1);

        for (i = 0; i < sizeof(items) / sizeof(ecat_sdo_t); i++)
        {
            _INF_("ecat_sdo_t: %d\r\n", i);
            ecat_sdo_t *item = &items[i];
            len = item->len;
            item->slave = s + 3; // 前面接了 3 个电机
```

```
            _INF_("ecat_sdo_t line: %d\r\n", __LINE__);
            ret = EcatCoeSDODownload(
                ecat_dev,
                item->slave,
                item->index,
                item->subindex,
                len,
                (void*)item->val,
                5000);
            _INF_("ecat_sdo_t line: %d\r\n", __LINE__);
            if (ret != ECAT_S_OK) {
                _DBG_("EcatCoeSDODownload failed: ret=%d, slave=%d, index=0x%04x,
subindex=0x%02x, value=0x%08x(%d)", ret, item->slave, item->index, item->subindex,
*(uint16_t*)item->val, *(uint16_t*)item->val);
                return ret;
            }
            _DBG_("EcatCoeSDODownload succeeded: slave=%d, index=0x%04x,
subindex=0x%02x, value=0x%08x(%d)", item->slave, item->index, item->subindex,
*(uint16_t*)item->val, *(uint16_t*)item->val);
        }
    }
    return ret;
}

int main(int argc, char* argv[])
{
    char index_buff[10];
    uint32_t index = 0;
    uint64_t loops = 0;
    uint32_t channel = 0;
    char *eni_file;
    ECAT_HANDLE ecat_dev;

    int btn_run = 0;
    int btn_stop = 0;
    int btn_close = 0;

    int run_cmd = 0;
    int run[NUM_SERVOS] = {0};
    int stop_cmd = 0;
    int halt_cmd = 0;
```

```
if (argc != 4)
{
   _INF_("usage: ./program encoder_id chnnal eni.xml\r\n");
   return 0;
}

index = atoi(argv[1]);
channel = atoi(argv[2]);
if(channel > 1) channel = 1;
eni_file = argv[3];

//PCIe-2E 只有一个通道,PCIe-4E 有两个通道
RETURN_IF_FAILED(EcatOpen(&ecat_dev, BOARD_ALIAS(index_buff, index), channel));

RETURN_IF_FAILED(EcatBusRun(ecat_dev, eni_file));

// 请求切换状态
uint8_t state;
RETURN_IF_FAILED(EcatRequestMasterState(ecat_dev, EcatStateO));
for (uint32_t i = 0; ; i++)
{
    RETURN_IF_FAILED(EcatGetMasterState(ecat_dev, &state));
    _INF_("request is: 8, curr is: %d", state);
    usleep(500*1000);
    if (state == EcatStateO) break;
}

_INF_("EcatRequestMasterState done");

RETURN_IF_FAILED(motor_config(ecat_dev));

// 清空并预计填入数据
proc_img_i *pi;
proc_img_o *po;
RETURN_IF_FAILED(EcatPINMap(ecat_dev, PI_AREA_LOCAL_INPUT, (void **)&pi));
_INF_("EcatPINMap done");

RETURN_IF_FAILED(EcatPINMap(ecat_dev, PI_AREA_LOCAL_OUTPUT, (void **)&po));
_INF_("EcatPINMap done");

_INF_("pi: 0x%llx", pi);
_INF_("po: 0x%llx", po);
```

```
for (uint32_t i = 0; i < 2; i++)
{
    EcatPIOutputQueuePush(ecat_dev, false, 100);
}
_INF_("EcatPINMap done");

RETURN_IF_FAILED(EcatPIEnable(ecat_dev));

while (1)
{
    while (EcatPIInputQueuePop(ecat_dev, false, 100) != 0);

    // 绿色按钮按下时，才启动
    btn_run = !((pi->inputs[0] >> BTN_RUN) & 0x1); // 输入低有效
    if(btn_run) {
        run_cmd = 1;
        stop_cmd = 0;
        halt_cmd = 0;
    }

    // 红色按钮按下，电机停止运行
    btn_stop = !((pi->inputs[0] >> BTN_STOP) & 0x1); // 输入低有效
    if(btn_stop) {
        stop_cmd = 1;
        run_cmd = 0;
    }
    else {
        stop_cmd = 0;
    }

    // 接近传感器
    btn_close = !((pi->inputs[0] >> BTN_CLOSE) & 0x1); // 输入低有效
    if(btn_close) {
        halt_cmd = 1;
        run_cmd = 0;
    }

    // 绿灯状态
    if(run_cmd) {
        // 常亮
        po->outputs[0] |= (1 << LED_GREEN);
    }
```

```
        else {
            // 闪烁 2Hz
            if (loops % 500 == 0) {
                po->outputs[0] ^= (1 << LED_GREEN);
            }
        }

        if(halt_cmd) {
            po->outputs[0] |= (1 << LED_RED);       // 红灯常亮
            po->outputs[0] &= ~(1 << LED_GREEN);    // 绿灯熄灭
        }
        else {
            po->outputs[0] &= ~(1 << LED_RED);      // 红灯熄灭
        }

        loops++;
        for (int i = 0; i < NUM_SERVOS; i++)
        {
            if(halt_cmd | stop_cmd) {
                // 电机 halt
                po->servos[i].ctrl.b.halt = 1;
            }
            else {
                po->servos[i].ctrl.b.halt = 0;
            }

            do {
                servos_t *servo = &__servos[i];
                if(!run_cmd || (i > 0 && run[i-1] == 0)) {
                    break;
                }

                run[i] = motor_control(i, &servo->ss, pi->servos[i].stat, &servo->sc,
false);

                if (pi->servos[i].stat.b.fault) { // 检测到报错
                    _DBG_("Motor fault, servo[%d] error code = 0x%04X", i,
pi->servos[i].error);
                }

                if (loops % 1000 == 0)
```

```
                {
                    _INF_("servo[%d]: p=%10d, ctrl:%08x stat=%08x", i, pi->servos[i].p,
po->servos[i].ctrl.v, pi->servos[i].stat.v);
                }

                // 伺服未出力，或已出力但延时未结束，应停止转动
                if (!run || servo->run_delay != 0) {
                    servo->out_p = pi->servos[i].p;
                    servo->zero_p = (float)servo->out_p;
                    servo->offset_p = 0;
                    servo->out_v = 0;
                    servo->out_t = 0;
                    servo->reverse = false;
                }

                // 伺服已出力，延时计数递减
                if (run && servo->run_delay)
                    servo->run_delay--;

                // 伺服已出力，延时结束，进行正常控制
                if (run && !servo->run_delay) {

                    switch (g_op_mode)
                    {
                    case OP_MODE_CSP:
                        {
                            // 不同轴单位可能不一样，下表定义各轴位置范围及增量
                            I32 range_min[] = { -50000 };
                            I32 range_max[] = {  50000 };
                            float step_size[] = { 50.f };
                            if (servo->offset_p < range_min[0]) servo->reverse = false;
                            if (servo->offset_p > range_max[0]) servo->reverse = true;
                            servo->offset_p += (servo->reverse ? -1 : 1) * step_size[0];
                            servo->out_p = (I32)(servo->zero_p + servo->offset_p);
                        }
                        break;
                    }
                }

                // 更新输出变量
                po->servos[i].ctrl = servo->sc;
                switch (g_op_mode)
```

```
                {
                case OP_MODE_CSP: po->servos[i].p = servo->out_p; break;
                }
            }while(0);

        }

        if (EcatPIOutputQueuePush(ecat_dev, 0, 100) != 0)
        {
            _ERR_("push pi output error");
            break;
        }

    }
    RETURN_IF_FAILED(EcatBusStop(ecat_dev));
    RETURN_IF_FAILED(EcatClose(ecat_dev));
    return 0;
}
```

3. 免责声明

本着为用户提供更好服务的原则，广州致远电子股份有限公司（下称“致远电子”）在本手册中将尽可能地为用户呈现详实、准确的产品信息。但介于本手册的内容具有一定的时效性，致远电子不能完全保证该文档在任何时段的时效性与适用性。致远电子有权在没有通知的情况下对本手册上的内容进行更新，恕不另行通知。为了得到最新版本的信息，请尊敬的用户定时访问致远电子官方网站或者与致远电子工作人员联系。感谢您的包容与支持！

诚信共赢，持续学习，客户为先，专业专注，只做第一

广州致远电子股份有限公司

更多详情请访问 **www.zlg.cn**

欢迎拨打全国服务热线 **400-888-4005**