```
#include <ros/ros.h>

#include <stdio.h>
#include <string.h>
#include <math.h>
#include <iostream>
#include <list>
#include <signal.h>
#include <stdlib.h>

#include <sys/ioctl.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/resource.h>

#include <sys/sem.h>
#include <sys/shm.h>
#include <sys/ipc.h>

#include <cobalt/time.h>
#include <cobalt/unistd.h>
#include <cobalt/pthread.h>

#include <fcntl.h>   // File control definitions
#include <errno.h>   // Error number definitions
#include <termios.h> // POSIX terminal control definitions
#include "utility.h"
#include "ethercat.h"



using namespace std;
/*-------------------------Variables----------------------------*/

#pragma region globalVars
// thread  EtherCAT
#define EC_TIMEOUTMON 500
#define stack64k (64 * 1024)
#define NSEC_PER_SEC 1000000000
#define NUM_AXIS_TMP 1

pthread_t thread1; // ethercat
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

```
char IOmap[4096] = {0}; // for XMC E-Cat
int expectedWKC;
boolean needlf;
volatile int wkc;
boolean inOP;
//----------------------------------------------------
#pragma pack(1)
typedef struct DSTRUCT_OUT      //  pdo
{                               // 1c12
    int32_t target_position;   // 0x607a
    int32_t target_velocity;   // 0x60ff
    int16_t max_torque;        // 0x6072
    uint16_t control_word;     // 0x6040 pdo domain 0x1604
    int16_t target_torque;     // 0x6071
    uint8_t mode_of_operation; // 0x6060
} out_ELMOt;

typedef struct DSTRUCT_IN            // pdo
{                                    // 1c13
    int32_t position_actual;         // 0x6064
    int32_t position_follow_err;     // 0x60f4
    int16_t torque_actual;           // 0x6077
    uint16_t status_word;            // 0x6041
    uint8_t mode_of_operation_disp; //
    int32_t velocity_actual;         // 0x606C
} in_ELMOt;

#pragma pack(4)

static out_ELMOt *out_ELMO[NUM_AXIS_TMP] = {NULL};
static in_ELMOt *in_ELMO[NUM_AXIS_TMP] = {NULL};
/*--------------------------------------------------------------*/

// EtherCAT
int ELMOsetupPlatinum(uint16 slave);
bool simpletest(char *ifname);
void add_timespec(struct timespec *ts, int64 addtime);
void ec_sync(int reftime, int cycletime, int *offsettime);



//------------------------------------------------------------
```

```
void *ecatthread(void *ptr)
{
    struct timespec ts, tleft;
    int ht;
    int cycletime;
    int toff = 0;

    struct sched_param p;

    p.sched_priority = sched_get_priority_max(SCHED_FIFO);
    pthread_setschedparam(pthread_self(), SCHED_FIFO, &p);

    clock_gettime(CLOCK_MONOTONIC, &ts);
    ht = (ts.tv_nsec / 1000000) + 1; /* round to nearest ms */
    ts.tv_nsec = ht * 1000000;
    cycletime = *(int *)ptr * 1000; /* cycletime in ns */

    printf("create ecatthread! ₩n");
    printf("pthread priority = %d₩n", p.sched_priority);

    ec_send_processdata();
    //----------------------------------------------------------------------------------
    while (1)
    {
        /* calculate next cycle start */
        add_timespec(&ts, cycletime + toff);
        std::cout << cycletime << "  cycletime  |  toff  " << toff << std::endl;
        /* wait to cycle start */
        clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &ts, &tleft);
        ec_receive_processdata(EC_TIMEOUTRET);

        if (ec_slave[0].hasdc)
        {
            ec_sync(ec_DCtime, cycletime, &toff);
        }
        ec_send_processdata();
    }
}

// Main Thread
int main(int argc, char **argv)
{
```

```
ros::init(argc, argv, "soem_node");
ros::NodeHandle nh;
char IOmap[1028] = {0};
char ch = 0;
int cnt = 0;
int ctime = 1000; // us()
char *eth_name = "rteth0";
pthread_attr_t thread_attr;
struct sched_param param;

try
{
    if (!simpletest(eth_name)) //
        throw "Failed initialization for master ₩n";
    if (ec_slavecount < NUM_AXIS_TMP)
        throw "Number of axis exceeded to ecat slaves ₩n";

    for (int i = 0; i < NUM_AXIS_TMP; i++)
    {
        out_ELMO[i] = (out_ELMOt *)ec_slave[i + 1].outputs;
        in_ELMO[i] = (in_ELMOt *)ec_slave[i + 1].inputs;

        out_ELMO[i]->target_position = in_ELMO[i]->position_actual;
    }

    pthread_attr_init(&thread_attr);
    pthread_create(&thread1, NULL, ecatthread, (void *)&ctime);
}
catch (const char *str)
{
    perror(str);
    abort();
}

mlockall(MCL_CURRENT | MCL_FUTURE);

while (ros::ok())
{
    cnt = (cnt + 1) % 10;
    if (cnt == 0) //
    {

    }
```

```
//==================================================================
================================================
        usleep(50000); //
    }
    return 0;
}

/*-------------------------------------------------------------*/

int ELMOsetupGOLD(uint16 slave)
{
    int wkc = 0;
    uint32_t sdoObj = 0x00000000;
    uint8_t diable_bits = 0x00;
    uint8_t enable_bits = 0x01;
    uint16_t objAddr = 0x0000;
    uint16_t TxAddr = 0x1607;
    uint16_t RxAddr = 0x1A07;

    wkc += ec_SDOwrite(slave, TxAddr, 0x00, FALSE, sizeof(diable_bits), &(diable_bits),
EC_TIMEOUTSAFE); // 0 disable
    sdoObj = 0x607A0020;
// Target position
    wkc += ec_SDOwrite(slave, TxAddr, 0x01, FALSE, sizeof(sdoObj), &(sdoObj),
EC_TIMEOUTSAFE);
    sdoObj = 0x60FF0020; // Target velocity
    wkc += ec_SDOwrite(slave, TxAddr, 0x02, FALSE, sizeof(sdoObj), &(sdoObj),
EC_TIMEOUTSAFE);
    sdoObj = 0x60720010; // Max torque
    wkc += ec_SDOwrite(slave, TxAddr, 0x03, FALSE, sizeof(sdoObj), &(sdoObj),
EC_TIMEOUTSAFE);
    sdoObj = 0x60400010; // Controlword
    wkc += ec_SDOwrite(slave, TxAddr, 0x04, FALSE, sizeof(sdoObj), &(sdoObj),
EC_TIMEOUTSAFE);
    sdoObj = 0x60710010; // Target torque
    wkc += ec_SDOwrite(slave, TxAddr, 0x05, FALSE, sizeof(sdoObj), &(sdoObj),
EC_TIMEOUTSAFE);
    sdoObj = 0x60600008; // Modes of operation
    wkc += ec_SDOwrite(slave, TxAddr, 0x06, FALSE, sizeof(sdoObj), &(sdoObj),
EC_TIMEOUTSAFE);
    enable_bits = 0x06;
    wkc += ec_SDOwrite(slave, TxAddr, 0x00, FALSE, sizeof(enable_bits), &(enable_bits),
EC_TIMEOUTSAFE);

    wkc += ec_SDOwrite(slave, RxAddr, 0x00, FALSE, sizeof(diable_bits), &(diable_bits),
EC_TIMEOUTSAFE); // 0 disable
    sdoObj = 0x60640020;
// Position actual value
    wkc += ec_SDOwrite(slave, RxAddr, 0x01, FALSE, sizeof(sdoObj), &(sdoObj),
EC_TIMEOUTSAFE);
    sdoObj = 0x60F40020; // Following error actual value
    wkc += ec_SDOwrite(slave, RxAddr, 0x02, FALSE, sizeof(sdoObj), &(sdoObj),
EC_TIMEOUTSAFE);
    sdoObj = 0x60770010; // Torque actual value
    wkc += ec_SDOwrite(slave, RxAddr, 0x03, FALSE, sizeof(sdoObj), &(sdoObj),
EC_TIMEOUTSAFE);
    sdoObj = 0x60410010; // Statusword
    wkc += ec_SDOwrite(slave, RxAddr, 0x04, FALSE, sizeof(sdoObj), &(sdoObj),
EC_TIMEOUTSAFE);
    sdoObj = 0x60610008; // Modes of operation display
    wkc += ec_SDOwrite(slave, RxAddr, 0x05, FALSE, sizeof(sdoObj), &(sdoObj),
EC_TIMEOUTSAFE);
    sdoObj = 0x20A00020; // Additional position actual value
    wkc += ec_SDOwrite(slave, RxAddr, 0x06, FALSE, sizeof(sdoObj), &(sdoObj),
EC_TIMEOUTSAFE);
    sdoObj = 0x606C0020; // Velocity actual value
    wkc += ec_SDOwrite(slave, RxAddr, 0x07, FALSE, sizeof(sdoObj), &(sdoObj),
EC_TIMEOUTSAFE);
    enable_bits = 0x07;
    wkc += ec_SDOwrite(slave, RxAddr, 0x00, FALSE, sizeof(enable_bits), &(enable_bits),
EC_TIMEOUTSAFE);

    // Tx PDO disable
    wkc += ec_SDOwrite(slave, 0x1c12, 0x00, FALSE, sizeof(diable_bits), &(diable_bits),
EC_TIMEOUTSAFE);
    // Tx PDO dictionary mapping
    objAddr = TxAddr;
    wkc += ec_SDOwrite(slave, 0x1c12, 0x01, FALSE, sizeof(objAddr), &(objAddr),
EC_TIMEOUTSAFE);

    // Rx PDO disable
    wkc += ec_SDOwrite(slave, 0x1c13, 0x00, FALSE, sizeof(diable_bits), &(diable_bits),
EC_TIMEOUTSAFE);
    // Rx PDO dictionary mapping
    objAddr = RxAddr;
    wkc += ec_SDOwrite(slave, 0x1c13, 0x01, FALSE, sizeof(objAddr), &(objAddr),
EC_TIMEOUTSAFE);

    enable_bits = 0x01;
    wkc += ec_SDOwrite(slave, 0x1c12, 0x00, FALSE, sizeof(enable_bits), &(enable_bits),
EC_TIMEOUTSAFE);
    wkc += ec_SDOwrite(slave, 0x1c13, 0x00, FALSE, sizeof(enable_bits), &(enable_bits),
EC_TIMEOUTSAFE);
    printf("wkc : %d₩n", wkc);

    uint8 u8val = 8;                                                  // 8:position
9:velocity 10:torque
    ec_SDOwrite(slave, 0x6060, 0x00, FALSE, sizeof(u8val), &u8val, EC_TIMEOUTRXM); //
  cyclic sychronous position mode

    printf("supported drive modes: %d₩n", u8val);


///////////////////////////////////////////////////////////////////////////////////////////////
////////

///////////////////////////////////////////////////////////////////////////////////////////////
////////

    // uint16 map_1c12[4] = {0x0003, 0x1604, 0x160C, 0x160B};
    // uint16 map_1c13[4] = {0x0003, 0x1A04, 0x1A1E, 0x1A11};
    // uint8 u8val = 8; //8:position 9:velocity 10:torque
    // //    uint32 u32val;

    // ec_SDOwrite(slave, 0x1c12, 0x00, TRUE, sizeof(map_1c12), &map_1c12,
EC_TIMEOUTSAFE); //pdo
    // ec_SDOwrite(slave, 0x1c13, 0x00, TRUE, sizeof(map_1c13), &map_1c13,
EC_TIMEOUTSAFE); //pdo 설정
    // ec_SDOwrite(slave, 0x6060, 0x00, FALSE, sizeof(u8val), &u8val, EC_TIMEOUTRXM);
     //    cyclic sychronous position mode

    // printf("ELMO SETUP SUCCESSFULLY > supported drive modes: %d₩n", u8val);
}

int ELMOsetupPlatinum(uint16 slave)
{
    int wkc = 0;
    uint8_t diable_bits = 0x00;
    uint8_t enable_bits = 0x01;
    uint16_t objAddr = 0x0000;
    uint32_t sdoObj = 0x00000000;

    wkc += ec_SDOwrite(slave, 0x1601, 0x00, FALSE, sizeof(diable_bits), &(diable_bits),
EC_TIMEOUTSAFE); // 0 disable
    sdoObj = 0x607A0020;
// Target position
    wkc += ec_SDOwrite(slave, 0x1601, 0x01, FALSE, sizeof(sdoObj), &(sdoObj),
EC_TIMEOUTSAFE);
    sdoObj = 0x60FF0020; // Target velocity
    wkc += ec_SDOwrite(slave, 0x1601, 0x02, FALSE, sizeof(sdoObj), &(sdoObj),
EC_TIMEOUTSAFE);
    sdoObj = 0x60720010; // Max torque
    wkc += ec_SDOwrite(slave, 0x1601, 0x03, FALSE, sizeof(sdoObj), &(sdoObj),
EC_TIMEOUTSAFE);
    sdoObj = 0x60400010; // Controlword
    wkc += ec_SDOwrite(slave, 0x1601, 0x04, FALSE, sizeof(sdoObj), &(sdoObj),
EC_TIMEOUTSAFE);
    sdoObj = 0x60710010; // Target torque
    wkc += ec_SDOwrite(slave, 0x1601, 0x05, FALSE, sizeof(sdoObj), &(sdoObj),
EC_TIMEOUTSAFE);
    sdoObj = 0x60600008; // Modes of operation
    wkc += ec_SDOwrite(slave, 0x1601, 0x06, FALSE, sizeof(sdoObj), &(sdoObj),
EC_TIMEOUTSAFE);
    enable_bits = 0x06; // nums of sdo // enable
    wkc += ec_SDOwrite(slave, 0x1601, 0x00, FALSE, sizeof(enable_bits), &(enable_bits),
EC_TIMEOUTSAFE);

    wkc += ec_SDOwrite(slave, 0x1A01, 0x00, FALSE, sizeof(diable_bits), &(diable_bits),
EC_TIMEOUTSAFE); // 0 disable
    sdoObj = 0x60640020;
// Position actual value
    wkc += ec_SDOwrite(slave, 0x1A01, 0x01, FALSE, sizeof(sdoObj), &(sdoObj),
EC_TIMEOUTSAFE);
    sdoObj = 0x60F40020; // Following error actual value
    wkc += ec_SDOwrite(slave, 0x1A01, 0x02, FALSE, sizeof(sdoObj), &(sdoObj),
EC_TIMEOUTSAFE);
    sdoObj = 0x60770010; // Torque actual value
    wkc += ec_SDOwrite(slave, 0x1A01, 0x03, FALSE, sizeof(sdoObj), &(sdoObj),
EC_TIMEOUTSAFE);
    sdoObj = 0x60410010; // Statusword
    wkc += ec_SDOwrite(slave, 0x1A01, 0x04, FALSE, sizeof(sdoObj), &(sdoObj),
EC_TIMEOUTSAFE);
    sdoObj = 0x60610008; // Modes of operation display
    wkc += ec_SDOwrite(slave, 0x1A01, 0x05, FALSE, sizeof(sdoObj), &(sdoObj),
EC_TIMEOUTSAFE);
    // sdoObj = 0x30CD02;//0x36E40220 ; // Additional position actual value
0x66110020; 0x30CD0220;
    // wkc += ec_SDOwrite(slave, 0x1A01, 0x06, FALSE, sizeof(sdoObj), &(sdoObj),
EC_TIMEOUTSAFE);
    sdoObj = 0x606C0020; // Velocity actual value
    wkc += ec_SDOwrite(slave, 0x1A01, 0x06, FALSE, sizeof(sdoObj), &(sdoObj),
EC_TIMEOUTSAFE);
    enable_bits = 0x06; // nums of sdo // enable
    wkc += ec_SDOwrite(slave, 0x1A01, 0x00, FALSE, sizeof(enable_bits), &(enable_bits),
EC_TIMEOUTSAFE);

    // Tx PDO disable
    wkc += ec_SDOwrite(slave, 0x1c12, 0x00, FALSE, sizeof(diable_bits), &(diable_bits),
EC_TIMEOUTSAFE);
    // Tx PDO dictionary mapping
    objAddr = 0x1601;
    wkc += ec_SDOwrite(slave, 0x1c12, 0x01, FALSE, sizeof(objAddr), &(objAddr),
EC_TIMEOUTSAFE);
    // Rx PDO disable
    wkc += ec_SDOwrite(slave, 0x1c13, 0x00, FALSE, sizeof(diable_bits), &(diable_bits),
EC_TIMEOUTSAFE);
    // Rx PDO dictionary mapping
    objAddr = 0x1A01;
    wkc += ec_SDOwrite(slave, 0x1c13, 0x01, FALSE, sizeof(objAddr), &(objAddr),
EC_TIMEOUTSAFE);

    enable_bits = 0x01;
    wkc += ec_SDOwrite(slave, 0x1c12, 0x00, FALSE, sizeof(enable_bits), &(enable_bits),
EC_TIMEOUTSAFE);
    enable_bits = 0x01;
    wkc += ec_SDOwrite(slave, 0x1c13, 0x00, FALSE, sizeof(enable_bits), &(enable_bits),
EC_TIMEOUTSAFE);
    printf("wkc : %d₩n", wkc);

    uint8 u8val = 8;                                                    // 8:position
9:velocity 10:torque
    ec_SDOwrite(slave, 0x6060, 0x00, FALSE, sizeof(u8val), &u8val, EC_TIMEOUTRXM); //
  cyclic sychronous position mode
```

```
    printf("supported drive modes: %d₩n", u8val);
}

bool simpletest(char *ifname)
{
    int i, j, oloop, iloop, chk, slc;
    u_int mmResult;

    needlf = FALSE;
    inOP = FALSE;

    printf("Starting simple test₩n");

    /* initialise SOEM, bind socket to ifname */
    if (ec_init(ifname)) // Network Interface Card NIC
    {
        printf("ec_init on %s succeeded.₩n", ifname);
        /* find and auto-config slaves */

        if (ec_config_init(FALSE) > 0) // Mailbox Setup,  Slave -> PRE_OP Mode, All data
read and configured are stored in a global array
        {
            printf("%d slaves found and configured.₩n", ec_slavecount);
            if ((ec_slavecount >= 1)) // ec_slabecount : number of slaves found on the
network
            {
                for (slc = 1; slc <= ec_slavecount; slc++) // ec_slave[0] : reserved for the
master
                {
                    printf("Name: %s EEpMan: %d eep_id: %d configadr: %d aliasadr: %d
State %d₩n₩r", ec_slave[slc].name, ec_slave[slc].eep_man,
                           ec_slave[slc].eep_id, ec_slave[slc].configadr, ec_slave[slc].aliasadr,
ec_slave[slc].state);
                    if (ec_slave[slc].eep_man == 154 && ec_slave[slc].eep_id == 198948) //
ELMO GOLD's man & id
                    {
                        printf("slave%d -> ELMO GOLD detected₩n", slc - 1);
                        ec_slave[slc].PO2SOconfig = &ELMOsetupGOLD;
                    }
                    else if (ec_slave[slc].eep_man == 154 && ec_slave[slc].eep_id ==
17825794) // ELMO PLATINUM's man & id
                    {
                        printf("slave%d -> ELMO PLATINUM detected₩n", slc - 1);
                        ec_slave[slc].PO2SOconfig = &ELMOsetupPlatinum;
                        ec_slave[slc].blockLRW = 1; // Platinum does not support LRW
function
                        ec_slave[0].blockLRW++; // Platinum does not support LRW function
                        // ecx_context.grouplist[slc-1].blockLRW = 1;
                    }
                    else if (ec_slave[slc].eep_man == 3138 && ec_slave[slc].eep_id == 0) //
XMC's man & id
                    {
                        printf("slave%d -> XMC detected₩n", slc - 1);
                    }
                }
            }
            ec_config_map(&IOmap); // 32 Map all PDOs from slaves to IOmap with
Outputs/Inputs in sequential order (legacy SOEM way).
            ec_configdc();

            printf("Slaves mapped, state to SAFE_OP.₩n");
            /* wait for all slaves to reach SAFE_OP state , When the mapping is done
SOEM requests slaves to enter SAFE_OP.*/

            ec_statecheck(0, EC_STATE_SAFE_OP, EC_TIMEOUTSTATE * 4);
            // Operation Mode

            oloop = ec_slave[0].Obytes;
            if ((oloop == 0) && (ec_slave[0].Obits > 0)) oloop = 1;
            if (oloop > 8) oloop = 8;
            iloop = ec_slave[0].Ibytes;
            if ((iloop == 0) && (ec_slave[0].Ibits > 0)) iloop = 1;
            if (iloop > 8) iloop = 8;

            printf("segments : %d : %d %d %d %d₩n", ec_group[0].nsegments,
ec_group[0].IOsegment[0], ec_group[0].IOsegment[1], ec_group[0].IOsegment[2],
ec_group[0].IOsegment[3]);

            printf("Request operational state for all slaves₩n");
            expectedWKC = (ec_group[0].outputsWKC * 2) + ec_group[0].inputsWKC;
            printf("Calculated workcounter %d₩n", expectedWKC);
            ec_slave[0].state = EC_STATE_OPERATIONAL;
//////////////////////////////// 0 or 1
            /* send one valid process data to make outputs in slaves happy*/
            ec_send_processdata();
            ec_receive_processdata(EC_TIMEOUTRET);
            /* request OP state for all slaves */
            ec_writestate(0); //////////////////////////////// 0 or 1
            chk = 200;
            /* wait for all slaves to reach OP state */
            do
            {
                ec_send_processdata();
                ec_receive_processdata(EC_TIMEOUTRET);
                ec_statecheck(0, EC_STATE_OPERATIONAL, 50000);
            } while (chk-- && (ec_slave[0].state != EC_STATE_OPERATIONAL));
            //////////////////////////////////////////////////
//////////////////////////////////////////////////
            if (ec_slave[0].state == EC_STATE_OPERATIONAL)
            {
                printf("Operational state reached for all slaves.₩n");
                inOP = TRUE;

                /* cyclic loop, reads data from RT thread */
                int PCL = 30; // Processdata Cycle Loop
                for (i = 1; i <= PCL; i++)
                {
                    ec_send_processdata();
                    wkc = ec_receive_processdata(EC_TIMEOUTRET);

                    if (wkc >= expectedWKC)
                    {
                        printf("Processdata cycle %4d, WKC %d , O:", i, wkc);

                        for (j = 0; j < oloop; j++)
                        {
                            printf(" %2.2x", *(ec_slave[0].outputs + j));
                        }

                        printf(" I:");
                        for (j = 0; j < iloop; j++)
                        {
                            printf(" %2.2x", *(ec_slave[0].inputs + j));
                        }
                        printf("₩nOutput Byte: %d  / input Byte: %d₩n", oloop, iloop);
                        // printf(" T:%lld₩r", ec_DCtime);
                        needlf = TRUE;
                    }
                    osal_usleep(50000);
                }
                inOP = FALSE;
            }
            else
            {
                printf("Not all slaves reached operational state.₩n");
                ec_readstate();
                for (i = 1; i <= ec_slavecount; i++)
                {
                    if (ec_slave[i].state != EC_STATE_OPERATIONAL)
                    {
                        printf("Slave %d State=0x%2.2x StatusCode=0x%4.4x : %s₩n",
                               i, ec_slave[i].state, ec_slave[i].ALstatuscode,
ec_ALstatuscode2string(ec_slave[i].ALstatuscode));
                    }
                }
                return false;
            }
            printf("₩nRequest init state for all slaves₩n");
            ec_slave[0].state = EC_STATE_INIT;
            /* request INIT state for all slaves */
            ec_writestate(0);

        }
        else
        {
            printf("No slaves found!₩n");
            return false;
        }
        // printf("End simple test₩n");
    }
    else
    {
        printf("No socket connection on %s₩nExcecute as root₩n", ifname);
        return false;
    }
    return true;
}

void add_timespec(struct timespec *ts, int64 addtime)
{
    int64 sec, nsec;
    nsec = addtime % NSEC_PER_SEC;
    sec = (addtime - nsec) / NSEC_PER_SEC;
    ts->tv_sec += sec;
    ts->tv_nsec += nsec;
    if (ts->tv_nsec > NSEC_PER_SEC)
    {
        nsec = ts->tv_nsec % NSEC_PER_SEC;
        ts->tv_sec += (ts->tv_nsec - nsec) / NSEC_PER_SEC;
        ts->tv_nsec = nsec;
    }
}
/* PI calculation to get linux time synced to DC time */
void ec_sync(int reftime, int cycletime, int *offsettime)
{
    static int integral = 0;
    static int delta;

    /* set linux sync point 50us later than DC sync, just as example */
    delta = (reftime - 50000) % cycletime;
    if (delta > (cycletime / 2))
    {
        delta = delta - cycletime;
    }
    if (delta > 0)
    {
        integral++;
    }
    if (delta < 0)
    {
        integral--;
    }
    *offsettime = -(delta / 100) - (integral / 20); // 100 20
    // gl_delta = delta;
}
```