# 步兵二代 代码说明

Langgo

|  |  |  |
| --- | --- | --- |
| Version | Date | Note |
| 1.0 | 2016-12-22 | 记录基本框架和主要思路。 |
| 1.1 | 2017-1-18 | 删除了kb\_task,改为hook |
|  |  | 修复了若干bug。 |
|  |  | 新增一点点说明。 |
|  |  |  |

软件/库版本

|  |  |  |
| --- | --- | --- |
| Keil5 | -uVision V5.21 | F4-DFP=V2.6 |
| CubePackage | STM32Cube\_FW\_F4\_V1.13.0 |  |
| STM32CubeMX | 4.16 |  |
| MCU | STM32F427IIH6 |  |

代码说明：

Bref：

代码使用ST的STM32Cube软件 + HAL驱动库（hardware abstract library）

+ freertos（作为HAL库的中间层。）为框架。

需要开发者参与的部分主要分为hal\_driver（cube软件生成），bsp，task三部分。

Bsp部分：包含硬件相关的模块和API，

例如pid.c 提供pid\_t类型的pid结构体定义，float pid\_calc(pid\_t\* pid, float fdb, float ref);函数。通过调用

void PID\_struct\_init(

pid\_t\* pid,

uint32\_t mode,

uint32\_t maxout,

uint32\_t intergral\_limit,

float kp,

float ki,

float kd);来初始化pid\_t类型的结构体。

之后就可以使用

float pid\_calc(pid\_t\* pid, float fdb, float ref);实现pid计算得到输出。

其他同理。相关.h文件一般对API说明的很清楚。

以下为各模块的简介。

|  |  |
| --- | --- |
| mpu.c | 提供板载IMU mpu6500和ist8030 IC的读取，主要作为云台角速度的获取。 |
| bsp\_can.c | 提供CAN消息发送API与全局CAN接受中断回调void HAL\_CAN\_RxCpltCallback(CAN\_HandleTypeDef\* \_hcan)。大部分模块的CAN反馈报文在这里读取与预处理。 |
| Bsp\_flash.c | 提供给calibrate.c使用。包含往片内flash写入校准数据的IO API，例如云台校准，imu校准数据 |
| Bt.c | 提供更高一级的uart接口功能API，如蓝牙，遥控器DBUS，裁判系统接口 |
| Arhs.c | 提供imu的姿态解算功能，这部分功能暂时还不能使用。 |
| Calibrate.c | 提供校准设备的Hook API， 一般在读取设备数据之后使用。 |

Task相关

Main函数：

初始化了pwm，tim，usart，dbus，can及滤波器等等，并从flash读取校准数据存到全局结构体变量gAppParam。之后初始化freertos和task。

一共四个task简介

|  |  |
| --- | --- |
| Gimbal | 云台、摩擦轮、拨单电机、控制 |
| Chassis | 底盘移动与3510电机控制 |
| Error | 用于模块离线错误检测与报警的任务。 |
| Kb | 用于PC按键判断与处理 |

以下分别介绍task以及控制流程。

Gimbal Task：

1、在task初始化阶段初始化pid结构体和参数。

2、初始化mpu。

3、相关变量预处理。。

4、之后进入任务循环，循环间隔5ms

While（1）{

1. imu温度的pid控制。保持imu期间的稳定。
2. 读取mpu的gyro，用于gimbal角速度闭环控制。
3. 对pit。Yaw轴的分别积分型位置控制输入和限幅。遥控器+鼠标键盘模式。

pit轴比较容易处理。Yaw要复杂一些。

其中Yaw分为两种模式。

《1》Yaw encoder mode：yaw 以自身的电机编码器为位置反馈做闭环。这种方式最简单，最直观。

《2》Yaw zgyro mode：yaw 以固定在自身的RM-陀螺仪模块为位置反馈。这种方式需要考虑陀螺仪上电校准时不能被移动/碰撞。为了保证步兵车能走直线，主要需要依靠这种模式。

同时，这里涉及一些数据换算，以函数说明。

s16 get\_relative\_pos(s16 raw\_ecd, s16 center\_offset)；获取相对于云台校准角度后的位置。例如pit和yaw都在中间时，获得校准值pit-offset，yaw-offset，则经过该函数返回的编码器值，当云台回中时，得到的数值就是0,然后顺时针为正 逆时针为负。

1. 根据遥控器（也就是rc）的右拨动开关选择yaw mode。如果是zgyro mode，需要加一些特殊处理，例如从其他mode切换进来时要先进入zgyro校准模式。
2. 根据鼠标和左拨码开关来处理摩擦轮和拨弹电机。

其中拨弹电机使用速度+位置双闭环控制。

1. 计算gimbal double axis PID， double loop control。

思路是：根据位置反馈计算得到pid-pit.pos\_out,然后将pos\_out作为速度环的参考，计算pid\_pit\_omg，最终得到的输出当做iq，发送到电机去。

根据目前控制模式和各模块状态，判断是否输出到gimbal motor。

//end of gimbal task loop，goto next time loop

}

Chassis task：

先介绍一些用到的函数。

|  |  |
| --- | --- |
| reset\_zgyro() | 复位陀螺仪。复位之后要注意必须使陀螺仪静止至少2s |
| move\_measure\_hook() | 用于根据麦轮电机上的编码器测量步兵底盘移动的坐标。 |
| mecanum\_calc（。。。） | 输入底盘移动的vx，vy，omega，得到解算后4个轮子的速度。 |
| get\_chassis\_mode\_set\_ref | 根据遥控器来获得chassis mode和vx vy。（之后这里会改） |

具体实现以及参数输入输出都在API上写了。请看代码。

Task任务：（10ms控制周期）

1. 先初始化底盘速度闭环的pid结构体数组pid\_spd[4]。
2. 初始化底盘跟随云台的角度闭环pid结构体pid\_chassis\_angle。
3. 设置底盘跟随的pid的一些特殊参数，例如死区，超出误差范围就切断输出，这是一个保护措施，防止步兵车因为超大误差引起的疯狂自旋。
4. 复位陀螺仪 等待一段时间 然后就可以开始进入任务循环。

While（1）{

1．获得chassis mode 和 输入vx，vy，o mega等速度指令。

2．插入一个测量底盘移动的hook。不过暂时用不到。

3．判断底盘mode， 目前其实只有一种mode，就是底盘跟随云台的编码器0值，即CHASSIS\_FOLLOW\_GIMBAL\_ENCODER。这样云台转到哪儿，底盘就跟到哪儿。按照这个思路，

chassis.omega = pid\_calc(&pid\_chassis\_angle, yaw\_relative\_pos, 0);这里通过控制chassis omega使底盘跟随云台编码器0实现角度闭环。

4. 调用mecanum\_calc(chassis.vx, chassis.vy, chassis.omega,

MAX\_WHEEL\_SPEED, chassis.wheel\_speed.s16\_fmt);

获取4个轮子的电机速度。

5．计算底盘电机的速度闭环pid，速度反馈由820R电调得到，输入速度即为经过麦轮解算函数mecanum\_calc得到的速度。计算得到输出到电调的iq。

6.判断当前各个模块是否正常是否在线，判断当前mode是否可以输出，

如果存在错误或者处于保护/静止模式，就把iq清零。

7通过can发送iq到820R驱动电机。

}

Error task：

这部分主要通过检测上一次收到模块数据的时间与当前系统时间之间间隔delta\_time来判断模块是否离线/timeout err/TOE。有几个主要的API如下。

|  |  |
| --- | --- |
| U32 HAL\_GetTick(); | 获取system time bask。 |
| void err\_detector\_callback(int err\_id) | 这是一个callback。应该放在收到数据后调用，用于记录模块收到数据的system time。一般在CAN-RX-IT中调用，输入参数为模块id即可，例如GimbalYawTOE。 |
| global\_err\_detector\_init(); | 用于初始化全局错误检测变量的函数。 |

以下是主要的错误检测结构体。4字节对齐。使用了bit filed。

#pragma pack(4)

typedef struct{

volatile uint32\_t last\_time; //记录上次时间

volatile uint32\_t err\_exist :1; //1 = err\_exist, 0 = everything ok

volatile uint32\_t enable :1; //是否使能。

volatile uint32\_t warn\_pri :6; //错误优先级

volatile uint32\_t beep\_mask :24; //蜂鸣器或者led显示错误的序列。

volatile uint32\_t delta\_time :16; //当前与上次直接的时间间隔。

volatile uint32\_t set\_timeout:16; //设定超时时间

void (\*f\_err\_deal)(void);//暂时没用到。保留作为错误处理函数。

}ErrorFlagTypedef;

#pragma pack()

全局错误检测结构体。包含各个模块的list，当前错误的string，id以及指针。

typedef struct{

volatile ErrorFlagTypedef\* err\_now;

volatile ErrorFlagTypedef err\_list[ErrorListLength];

char \*str; //for oled display

TOE\_ID\_e id;

}GlbRxErrTypeDef;

模块Id枚举：

typedef enum{

NoTimeOutErr = 0,

ChassisGyroTOE,

ChassisMoto1TOE,

ChassisMoto2TOE,

ChassisMoto3TOE,

ChassisMoto4TOE,

CurrentSampleTOE,

DbusTOE,

JudgeTOE,

GimbalYawTOE,

GimbalPitTOE,

PokeMotoTOE, //拨弹电机。

ErrorListLength，}TOE\_ID\_e;

Task介绍：（50ms周期）

1. 初始化全局结构体gRxErr的列表，

例如gRxErr.err\_list[DbusTOE].err\_exist = 0;

gRxErr.err\_list[DbusTOE].warn\_pri = 10; //max !

gRxErr.err\_list[DbusTOE].beep\_mask = Bi\_Bi;

gRxErr.err\_list[DbusTOE].set\_timeout = 100; //ms

gRxErr.err\_list[DbusTOE].enable = 1;

2、（到各个模块的RX IT中插入err\_detector\_callback）

开始进入循环检测。

while(1){

1，遍历err-list[ErrorListLength], 如果存在delta time > set timeout。就说明存在错误了。然后依次按照错误提示优先级对错误进行排序。取得当前错误指针。

2，按照指针指向通过蜂鸣器/led来提示错误。

}

Kb task：（10ms周期）

这部分非常简单。首先

介绍一个根据有限状态机设计的按键单击检测。例如鼠标左键按下再松开才算完整的一次按下，避免抖动引起的意图单击实际连击的情况。

u8 KeyStateMachine(u8\* psta, u8 condition)。每10ms扫描一次，根据函数返回值得到按键单击与否。具体实现就是保存按键状态，根据当前按键输入决定状态转移与输出。

key\_mouse\_task：

1. 延时1s。等待系统稳定

2进入循环

While（1）

{

1. 根据遥控器结构体变量rc.kb.code得到一个map到16个按键的变量。
2. 根据WSAD键来积分，同时有限幅。不能直接给定一个vx或vy是因为手感问题，所以需要加上一个平缓的曲线，防止底盘行动时抖动。

//end of loop

}

代码中值得一谈的其他部分：

关于uart的空闲中断，通过dma接收数据的思路。

首先重写了一个

static int UART\_Receive\_DMA\_No\_IT(UART\_HandleTypeDef\* huart, u8\* pData, u32 Size)

这个函数开启串口空闲中断（即接收完一串数据后硬件会发生IDLE中断）。有一点要注意的是，这里只开启UART 的全局中断，对应的IRQhandler是void USART1\_IRQHandler(void)。

不开启UART的DMA channel的DMA IT。仅仅只是使用DMA来传输数据，不触发DMA接收完毕的IT。void DMA1\_Stream1\_IRQHandler(void)这类函数handler也是不会进入的。这是为了减少dma频繁中断造成的系统任务切换，提高系统运行效率。

使用idle it需要在真正的uart IRQhandler（）里面加入一个自定义的数据帧接收解析和重启DMA的函数，例如LanggoUartFrameIRQHandler(&huart1);这个函数完成对接收缓冲区（一个数组而已）的数据解析，符合协议则使用，否则丢弃。这里同时也提高系统接收数据的鲁棒性，防止接收的数据不完整造成的整体解析出错。解析数据之后应该记得要清楚idle flag，同时重启DMA。将接收指针重新指向接收缓冲区的起始位置。这些操作调用API

uart\_reset\_idle\_rx\_callback(huartx);实现。

另外还有一点就是串口直接如果需要处理float 类型的数据，就使用联合体或者直接内存拷贝4个字节（注意大小端）然后发送。接收的时候重新组合就行了。