新超电

控制逻辑

在电路上，超电是对底盘用电的一种补偿。举个例子，假设底盘需要的功率为220w，但此时裁判系统功率功率限制为50w，那么剩下的170w就由超电提供。如果底盘此时仅用了30w功率，而裁判系统功率限制为50w，那么超电就会自动充电。(仅供理解，实际可能有所差别)

在实际使用中，超电是一直开着的，超电会根据底盘的用电情况来自动放电充电，控制上仅直接需修改对底盘的功率限制即可。

目前仅能在键鼠模式下控制超电的开关，ctrl+C开超电，ctrl+shift+C关超电。

以上是大概说明，接下来说明代码(rm_control/rm_common/include/decision/power_limit.h)

```
typedef enum
{
  CHARGE = 0, //超电充电
  BURST = 1,  //使用超电
  NORMAL = 2, //不使用超电
  ALLOFF = 3, //超电关闭
  TEST = 4,
} Mode;
```

上面5种状态应从对底盘的功率限制上去理解。charge、burst、normal默认超电是开的。

核心代码如下:

`expect_state`: 所期望的超电状态(charge、burst、normal、alloff)　　`cap_state`: 超电实际状态(开、关)

```
void setLimitPower(rm_msgs::ChassisCmd& chassis_cmd, bool is_gyro)
{
  //工程不需要使用超电
  if (robot_id_ == rm_msgs::GameRobotStatus::BLUE_ENGINEER || robot_id_ ==
rm_msgs::GameRobotStatus::RED_ENGINEER)
    chassis_cmd.power_limit = 400;
  else
  {  // standard and hero
    if (referee_is_online_)  //如果裁判系统不在线，底盘功率限制为safety_power
    {
      if (capacity_is_online_)//如果超电不在线，则底盘功率进入normal模式
      {
        if (expect_state_ == ALLOFF)  //如果设置超电关闭，则底盘功率进入normal模式
          normal(chassis_cmd);
        else
        {
          if (chassis_power_limit_ > burst_power_)
            chassis_cmd.power_limit = burst_power_;
          else
          {
            switch (is_new_capacitor_ ? expect_state_ : cap_state_)
           //新旧超电控制上的差异在于：新超电直接设置底盘功率限制；旧超电根据超电状态来设置底盘
功率限制
```

```
            //根据不同的状态，进入相应的模式
             {
               case NORMAL:
                 normal(chassis_cmd);
               break;
               case BURST:
                 burst(chassis_cmd, is_gyro);
               break;
               case CHARGE:
                 charge(chassis_cmd);
               break;
               default:
                 zero(chassis_cmd);
               //如果没收到这3种状态，则底盘功率限制为0，这就会导致底盘动不了，云台能正常运动。
               break;
             }
           }
         }
       }
       else
         normal(chassis_cmd);
     }
     else
       chassis_cmd.power_limit = safety_power_;
   }
}
```

接下来说明burst模式

```
void burst(rm_msgs::ChassisCmd& chassis_cmd, bool is_gyro)
{
  if (cap_state_ != ALLOFF) //为防止超电关闭，需加入这个判断
  {
    if (cap_energy_ > capacitor_threshold_) //若超电剩余容量少于阈值，则进入normal模
式；阈值由电路给出，根据实际测试适当调整
    {
      if (is_gyro)
        chassis_cmd.power_limit = chassis_power_limit_ + extra_power_;
        //键鼠模式下小陀螺会进入normal模式(防止操作手爽爽小陀螺把超电用完了)，因此功率计算有所
不同。新超电下，extra_power需给0。
      else
        chassis_cmd.power_limit = burst_power_;
    }
    else
      expect_state_ = NORMAL;
  }
}
```

normal模式：

```
void normal(rm_msgs::ChassisCmd& chassis_cmd)
{
  double buffer_energy_error = chassis_power_buffer_ - buffer_threshold_;
  double plus_power = buffer_energy_error * power_gain_;
  chassis_cmd.power_limit = chassis_power_limit_ + plus_power;
  // TODO:Add protection when buffer<5
  //裁判系统存在缓冲功率，此处是为了使用缓冲功率。新超电下，不使用缓冲功率，power_gain需给0。
}
```

不妨思考下，使用新超电时，extra_power、power_gain为什么要给0?

以上是超电的控制逻辑，接下来说明超电通信。

超电通信

如上是超电的接线图，裁判系统接超电，再接到电脑。如果说读不到裁判系统数据，原因之一是超电出现问题。

通信实际上是数据交换，我们需要明白nuc往超电发什么、怎么发(控制），超电往nuc发什么(电路)。

接下来先介绍nuc往超电发的数据。

NUC发送数据

前面的控制逻辑提到一个变量：`expect_state`，这个是控制逻辑的核心。如果需要修改`expect_state`，则要用到如下函数

```
void updateState(uint8_t state)
{
  if (!capacitor_is_on_)
    expect_state_ = ALLOFF; //如果超电是关的，不管state是什么，expect_state_都是ALLOFF
  else
    expect_state_ = state;
}
```

在manual中，会调用上面这个函数对`expect_state`进行修改。

manual_base中有`manual_to_referee_pub_`这样一个发布者，发布的数据为`manual_to_referee_pub_data_`，这个数据在`ChassisGimbalShooterManual::checkReferee()`中赋值，`expect_state`就这样被发布出去。

referee_base中，有`manual_data_sub_`这样一个接收者：

```
RefereeBase::manual_data_sub_ =
    nh.subscribe<rm_msgs::ManualToReferee>("/manual_to_referee", 10,
&RefereeBase::manualDataCallBack, this);
```

当收到数据时，会调用如下函数：

```
void RefereeBase::manualDataCallBack(const rm_msgs::ManualToReferee::ConstPtr&
data)
{
  if (chassis_trigger_change_ui_)
    chassis_trigger_change_ui_->updateManualCmdData(data); //对于超电而言，仅这个用上

  if (shooter_trigger_change_ui_ && !is_adding_)
    shooter_trigger_change_ui_->updateManualCmdData(data);
  if (gimbal_trigger_change_ui_ && !is_adding_)
    gimbal_trigger_change_ui_->updateManualCmdData(data);
  if (target_trigger_change_ui_ && !is_adding_)
    target_trigger_change_ui_->updateManualCmdData(data);
  if (cover_flash_ui_ && !is_adding_)
    cover_flash_ui_->updateManualCmdData(data, ros::Time::now());
}

void ChassisTriggerChangeUi::updateManualCmdData(const
rm_msgs::ManualToReferee::ConstPtr data)
{
  power_limit_state_ = data->power_limit_state;  //更新数据
}
```

上面函数涉及到一个类：`chassis_trigger_change_ui`，可以理解为：与底盘相关的ui。

ui实际上是往裁判系统发送数据，这个过程由ChassisTriggerChangeUi::update( )实现。通过跳转可以看到，当chassis_cmd_sub_收到"/cmd_chassis"时，chassis_trigger_change_ui就会更新，调用ChassisTriggerChangeUi::update( )，这个函数的核心是`updateConfig`

```
void ChassisTriggerChangeUi::updateConfig(uint8_t main_mode, bool main_flag,
uint8_t sub_mode, bool sub_flag, bool extra_flag)
{
  static ros::Time trigger_time;
  static int expect;
  static bool delay = false;
  if (main_mode == 254)
  {
    graph_->setContent("Cap reset");
    graph_->setColor(rm_referee::GraphColor::YELLOW);
    return;
  }
  graph_->setContent(getChassisState(main_mode));
  if (sub_mode == 1)
    graph_->setColor(rm_referee::GraphColor::PINK);
  else
  {
    if (base_.capacity_recent_mode_ == rm_common::PowerLimit::ALLOFF && !delay)
    {
      trigger_time = ros::Time::now();
      expect = power_limit_state_;
      delay = true;
    }
    else if (delay)
    {
      if (expect != power_limit_state_)
      {
        trigger_time = ros::Time::now();
        expect = power_limit_state_;
```

```
      }
      else if ((ros::Time::now() - trigger_time).toSec() > 0.2)
      {
        if (main_flag)
          graph_->setColor(rm_referee::GraphColor::ORANGE);
        else if (sub_flag)
          graph_->setColor(rm_referee::GraphColor::GREEN);
        else if (extra_flag)
          graph_->setColor(rm_referee::GraphColor::WHITE);
        else
          graph_->setColor(rm_referee::GraphColor::BLACK);
        delay = false;
      }
    }
    else
    {
      if (main_flag)
        graph_->setColor(rm_referee::GraphColor::ORANGE);
      else if (sub_flag)
        graph_->setColor(rm_referee::GraphColor::GREEN);
      else if (extra_flag)
        graph_->setColor(rm_referee::GraphColor::WHITE);
      else
        graph_->setColor(rm_referee::GraphColor::BLACK);
    }
  }
}
```

上面代码实现了以下功能：

- `expect_state`为normal时，GraphColor为WHITE；`expect_state`为charge时，GraphColor为GREEN；`expect_state_`为burst时，GraphColor为BLACK；`expect_state`为ALLOFF时，GraphColor为BLACK。
- 当超电当前状态为关闭时，开关超电存在0.2s延迟（防止操作手误触）

实际上，超电所收到的信息，就是根据GraphColor的颜色。（具体发送了什么，可以看裁判系统串口协议）

以上就是nuc发送数据的大概过程，具体还有一些细节，可以自行看代码。

NUC接收数据

nuc接收数据就相当简单了，如下：

```
case rm_referee::POWER_MANAGEMENT_SAMPLE_AND_STATUS_DATA_CMD:
{
  rm_msgs::PowerManagementSampleAndStatusData sample_and_status_pub_data;
  uint8_t data[sizeof(rm_referee::PowerManagementSampleAndStatusData)];
  memcpy(&data, rx_data + 7,
sizeof(rm_referee::PowerManagementSampleAndStatusData));
  sample_and_status_pub_data.chassis_power = (static_cast<uint16_t>((data[0] <<
8) | data[1]) / 100.);
  sample_and_status_pub_data.chassis_expect_power = (static_cast<uint16_t>
((data[2] << 8) | data[3]) / 100.);
  sample_and_status_pub_data.capacity_recent_charge_power =
      (static_cast<uint16_t>((data[4] << 8) | data[5]) / 100.);
  sample_and_status_pub_data.capacity_remain_charge =
```

```
      (static_cast<uint16_t>((data[6] << 8) | data[7]) / 10000.);
  sample_and_status_pub_data.capacity_expect_charge_power = static_cast<uint8_t>
(data[8]);
  sample_and_status_pub_data.state_machine_running_state =
base_.capacity_recent_mode_ =
      static_cast<uint8_t>(data[9] >> 4);
  sample_and_status_pub_data.power_management_protection_info =
static_cast<uint8_t>((data[9] >> 2) & 0x03);
  sample_and_status_pub_data.power_management_topology = static_cast<uint8_t>
(data[9] & 0x03);
  sample_and_status_pub_data.stamp = last_get_data_time_;

  referee_ui_.capacityDataCallBack(sample_and_status_pub_data,
last_get_data_time_);

  power_management_sample_and_status_data_pub_.publish(sample_and_status_pub_data
);
  break;
}
```

上面的解包是根据超电发送的数据写的，这个应和电路那边约定好。

在此，对各个变量作解释：

- chassis_power：电管输出功率，从裁判系统读到的，后面应该会删去（裁判系统协议V1.7.0版本不再发送这个数据）
- capacity_recent_charge_power：模组充电功率
- capacity_remain_charge： 超电容量百分比
- capacity_expect_charge_power：超电放电功率
- state_machine_running_state：超电状态（开：1　关：3）（这个根据前面定义的状态枚举来设置的）