BoxFish机器鱼单片机代码文档

1 简介

1.1 基本功能

本代码用于Box Fish的控制任务，主要基于STM32F407VGTx系列单片机，开发环境为Keil 5。

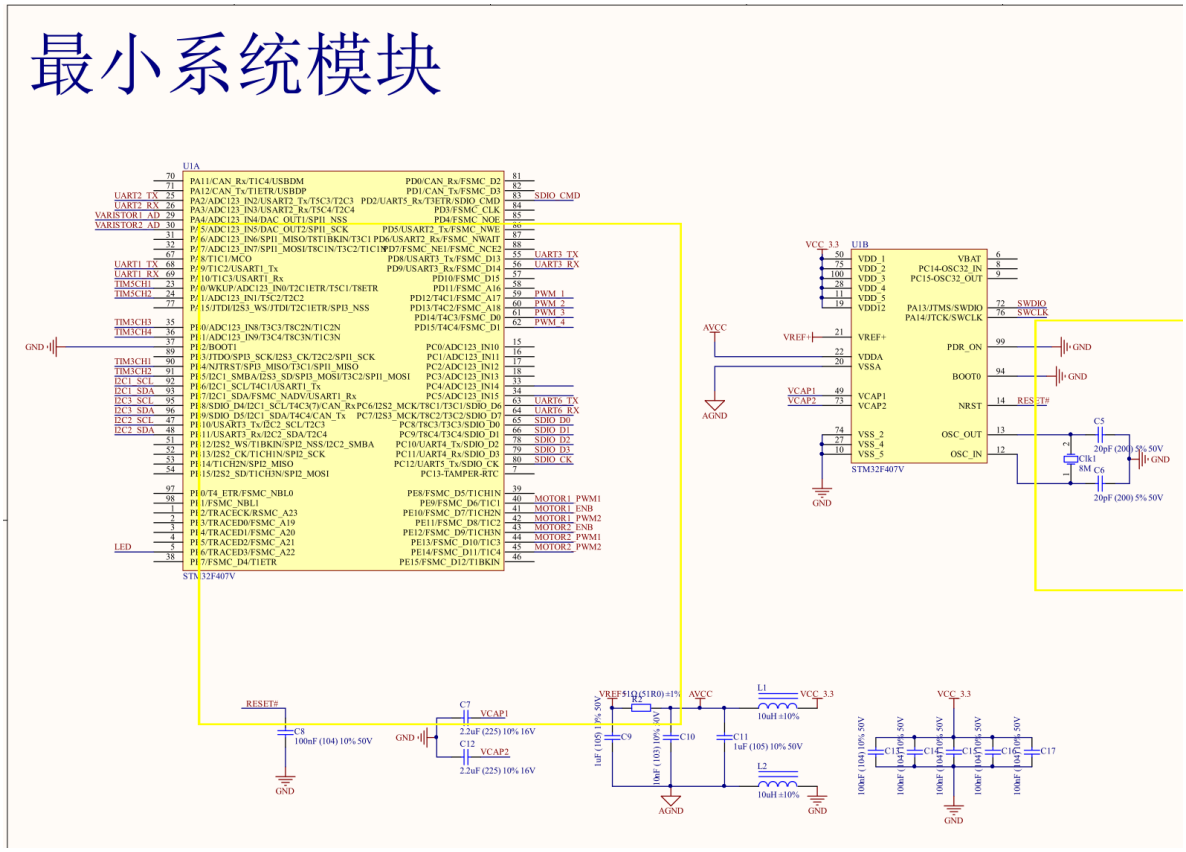
目前，本代码可实现的功能包括：

- 基本的舵机驱动，例如BLS3511S舵机（平台使用的）
- 基本的传感器驱动，包括IMU（MPU6050、MPU9150），深度传感器（MS5837）
- 基本的游动控制功能，基于CPG模型的尾巴拍动控制
- 上位机通讯功能，设计了一款RFLink通讯协议，具有良好的可扩展性
- SD卡读写功能，可在实验过程中方便的存储数据
- 数据回传功能，借助上位机，可将SD卡上的数据发送至电脑
- 数据显示功能，借助上位机，可实时显示传感器的数据变化
- 云台控制功能（暂未开发）

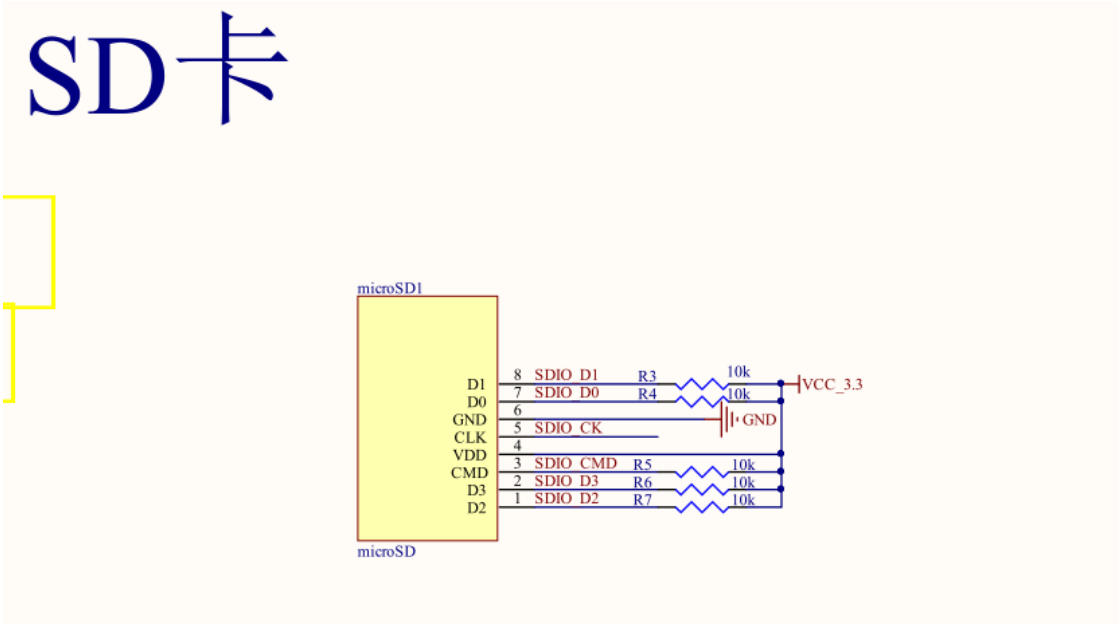
1.2 电路原理图

下面为代码所基于的电路原理图。

1.2.1 单片机最小系统



1.2.2 SD卡电路图

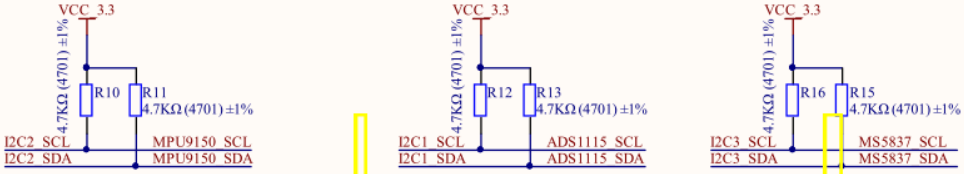


1.2.3 传感器电路

传感器模块

压敏电阻检测电路

I2C上拉电路

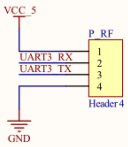


1.2.4 接口端子

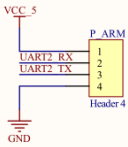
端子

程序下载接口

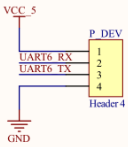
无线通信串口



ARM板通信串口

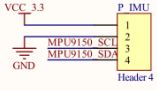


飞轮通信串口



外部传感器接口

外置MPU9150

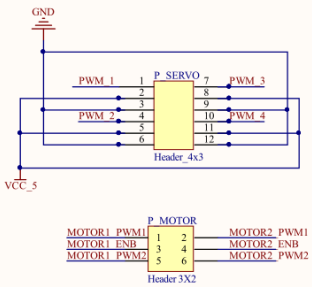


外置AD采样模块

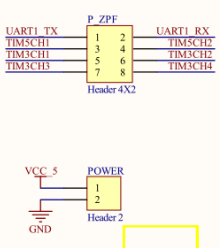
外置深度传感器模块

外置压敏电阻

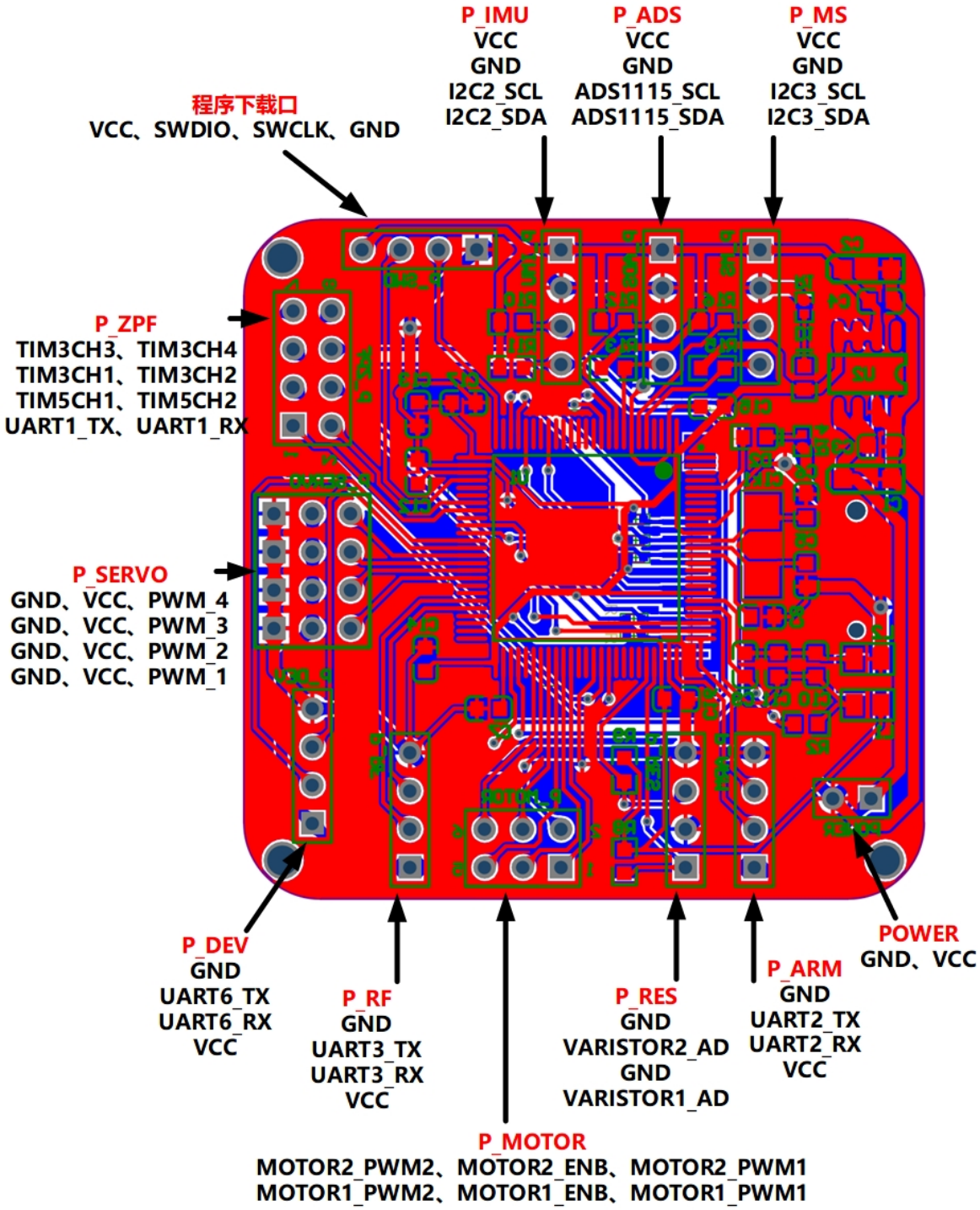
舵机输出



预留接口



1.2.5 PCB板



1.3 传感器以及执行器

舵机：BLS3511S舵机

IMU：MPU9250

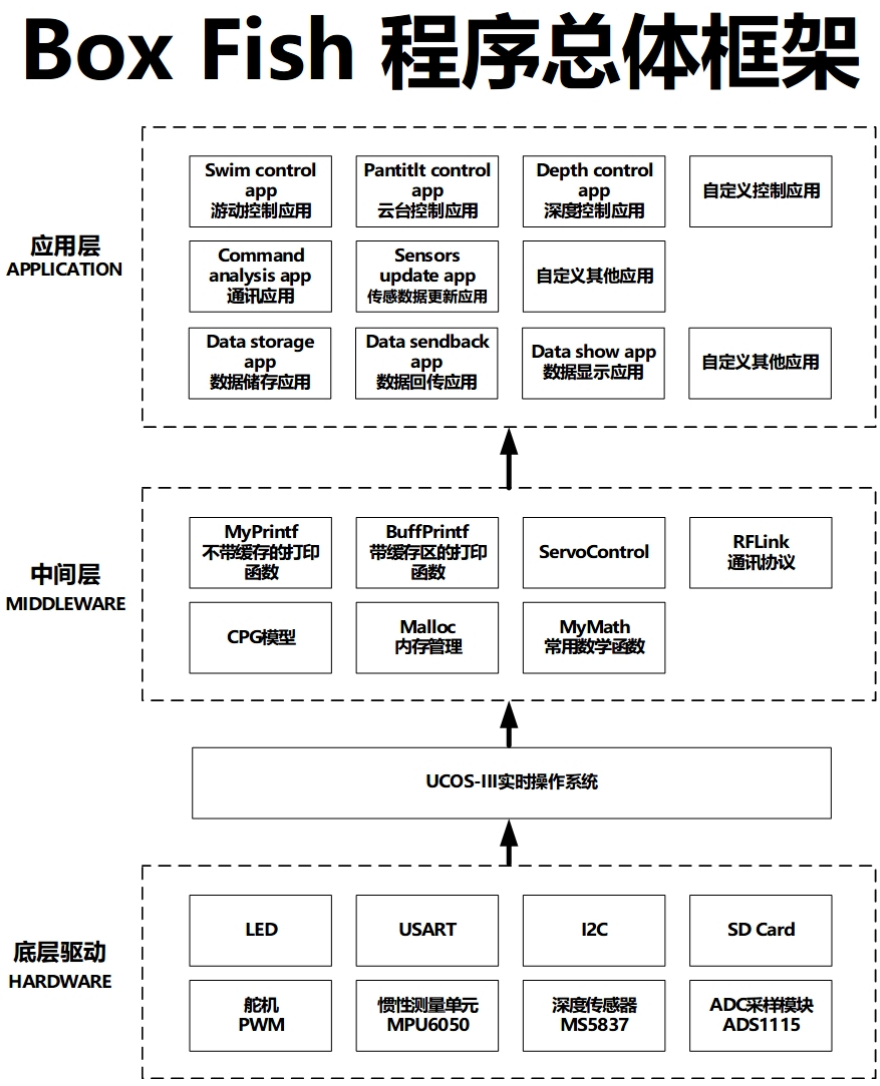
深度传感器：MS5837

2 代码介绍

2.1 代码框架

下图为Box Fish代码整体框架。其特点包括：

- 代码基于UCOS-III实时操作系统编写
- 代码提供包括LED、USART、I2C、SDCard等等在内的多种底层驱动，当然这是基于我们所提供的电路板的
- 代码中间层提供了打印输出函数，舵机控制，RFLink通讯协议等多种功能，用户只需直接调用即可
- 代码应用层已经提供了包括基础的游动控制、通讯、传感器数据更新、数据储存、传输等多种功能，用户也再次基础上定制自身所需要的其他功能



2.2 RFLink通讯协议

2.2.1 消息格式

为了提高控制指令的扩展性，我们设计了一种RFLink（Robotic Fish Link）通讯协议。

这个通讯协议基于串口进行发送，每一帧消息都由多个字节构成。RFLink通讯协议的一帧消息格式如下：

0xFF	0xFF	Sender ID	Receiver ID	Message length High 8 bit	Message length Low 8 bit	Command ID	Message	...	Message	Check number

详细说明：

- 上图中，每一个方格代表一个字节，即8 bit的二进制位
- 一帧消息的数据头是两个0xFF
- 发送者本人的ID，也是一个8位的无符号整型数
- 接收方的ID，也是一个8位的无符号整型数，由此，可以指定某人进行接收
- 消息总长度的高八位和低八位
- 命令的ID，我们可以定义多个命令，每一条命令都有独属的ID
- 消息的主体内容，多个8位的无符号整型数
- 最后是一个检验数，是将前面的所有数据加和，然后除以255取余得到的。通过这个检验数，我们可以丢弃产生接收错误的消息。

2.2.2 RFLink通讯协议的实现

主要实现包括三点：

- rflink.h文件
  - 在这个文件中，我们定义了发送者和接受者的ID
  - 定义了Comand的枚举类型，定义了多条指令，如果想要添加新的命令，就在这个枚举类型中添加即可。但是，需要注意，所有的使用RFLink通讯的程序，Command枚举类型要同步。例如上位机的Command列表和下位机的Comand列表要完全一致，否则会出现错误。

```
typedef enum Command
{
    SHAKING_HANDS=1,
    SET_SWIM_RUN,
    SET_SWIM_STOP,
    SET_SWIM_FORCESTOP,
    SET_SWIM_SPEEDUP,
    ..... 此处省略多条指令
    PRINT_SYS_MSG,
    LAST_COMMAND_FLAG
}Command;
```

- rflink.c文件
  - 实现了RFLink_receiveStates_1函数，利用状态机机制来接收并判断消息是否正确，是否正常接收

    - 实现了RFLink_commandAnalysis函数，判断是否和上位机握手成功，判断消息来自于哪里
    - 实现了RFLink_sendStruct函数，用于发送Command和Message
    - 实现了RFLink_message函数，用于格式化输出语句，类似printf的功能
- BuffPrintf.c文件
    - 实现了RFLinkPrintf函数，这是RFLink发送消息的底层实现，是一个基于缓冲区的消息发送函数，这个函数中体现了关于消息结构的内容。总之，非常重要。

3 使用方法

3.1 新建控制任务App的方法——以深度控制为例

3.1.1 第一步

在APPLICATION下创建一个新的文件夹，比如DepthControl

3.1.2 第二步

新建头文件depth_control_app.h，保存在DepthControl文件夹下，并增加如下内容：

```
/*********************************************************************
*************
Copyright: CASIA 仿生机器鱼实验室
File name:
Description:
Author: VincentFEI
Version: 1.0.0
Date: 2020.09.28
History:
*********************************************************************
************/
#ifndef _DEPTH_CONTROL_APP_H_
#define _DEPTH_CONTROL_APP_H_

// 系统头文件
#include "stm32f4xx.h"
#include "includes.h"
#include "os_cfg_app.h"
// RFLink通讯相关的头文件
#include "rflink.h"
```

```
#include "uart6_printf.h"
#include "BuffPrintf.h"
// 机器人状态头文件
#include "robotstate.h"
// 控制器参数头文件
#include "control_param.h"

// 定义任务
#define DEPTH_CONTROL_APP_TASK_PRIO 10  // 设置任务优先级,一般最少都有256个优先级，数字越小代表的优先级越高
#define DEPTH_CONTROL_APP_STK_SIZE 256 // 设置任务堆栈大小（这里的1个代表4字节）
#define DEPTH_CONTROL_APP_TIMER_PERIOD_MS 10 // 设置运行周期，这里说明期望这个任务每隔10ms运行一次
#define DEPTH_CONTROL_APP_TIMER_PERIOD_TICKS (((uint32_t)DEPTH_CONTROL_APP_TIMER_PERIOD_MS*1000)/OS_CFG_TMR_TASK_RATE_HZ) /* 定时器时钟频率被设置为了 1000Hz */

// 任务初始化函数
void depth_control_app_init(void);
// 任务暂停函数
void depth_control_app_stop(void);
// 任务恢复运行函数
void depth_control_app_resume(void);

#endif
```

3.1.3 第三步

新建源文件depth_control_app.c，保存在DepthControl文件夹下，并增加如下内容：

```
/*********************************************************************************************
Copyright: CASIA 仿生机器鱼实验室
File name:
Description:
Author: VincentFEI
Version: 1.0.0
Date: 2020.09.28
History:
*********************************************************************************************/
#include "depth_control_app.h"
extern BOXFISH boxfishstate;
```

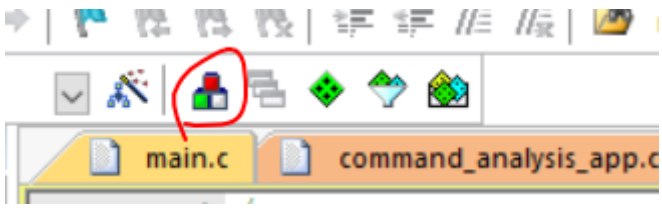
```c
static OS_TMR DepthControlTmr; // 定义一个定时器
static OS_TCB DepthControlTCB; // 定义一个任务控制块
__align(8) CPU_STK
DEPTH_CONTROL_APP_TASK_STK[DEPTH_CONTROL_APP_STK_SIZE]; // 定义任务堆栈

// 定时器回调函数，每当计时到达，就会执行这个函数，这个函数给任务发送信号量，从而唤醒任务
static void depth_control_app_tmrcallback(void)
{
    OS_ERR err;
    OSTaskSemPost(&DepthControlTCB, // 发送信号量给DepthControl任务
            OS_OPT_POST_NONE,
            &err);
}

// 任务主函数，就是会被反复执行的函数
static void depth_control_app_task(void * p_arg)
{
    OS_ERR err;
    CPU_TS ts;
    while(1)
    {
        // 等待信号量，每当信号量到来，就继续往下执行，否则就进入挂起的状态
        OSTaskSemPend(0,
                  OS_OPT_PEND_BLOCKING,
                  &ts,
                  &err);

        // 任务主体代码
        // 这里由大家自由发挥
    }
}

// 任务初始化函数
void depth_control_app_init(void)
{
    CPU_SR_ALLOC();

    OS_ERR err;

    OS_CRITICAL_ENTER();

    // 创建任务
```

```
    OSTaskCreate(&DepthControlTCB,       // 任务控制块
         "Depth Control App",        // 任务名称
         depth_control_app_task,     // 任务主函数
         0,              // 传递给任务的参数
         DEPTH_CONTROL_APP_TASK_PRIO,    // 任务优先级
         &DEPTH_CONTROL_APP_TASK_STK[0], // 任务堆栈基地址
         DEPTH_CONTROL_APP_STK_SIZE/10,  // 任务堆栈深度限位
         DEPTH_CONTROL_APP_STK_SIZE,     // 任务堆栈大小
         0, // 任务任务内部消息队列能够接收的最大消息数目,为0时禁止接收消息
         0, // 当使能时间片轮转时的时间片长度，为 0 时为默认长度
         0, // 用户补充的存储区
         OS_OPT_TASK_STK_CHK|OS_OPT_TASK_STK_CLR, // 任务的特定选项
                                 // OS_OPT_TASK_NONE       表示没有任何
选项
                                 // OS_OPT_TASK_STK_CHK    指定是否允许
检测该任务的堆栈
                                 // OS_OPT_TASK_STK_CLR    指定是否清除
该任务的堆栈
                                 // OS_OPT_TASK_SA VE_FP   指定是否存储
浮点寄存器，CPU需要有浮点运算硬件并且有专用代码保存浮点寄存器。
         &err);

    // 创建定时器
    OSTmrCreate(&DepthControlTmr,     // 定时器指针
        "Depth Control Timer",    // 定时器名字
        0,                        // 初始化定时器的延迟值
        DEPTH_CONTROL_APP_TIMER_PERIOD_TICKS, // 重复周期
        OS_OPT_TMR_PERIODIC, // 定时器运行选项，OS_OPT_TMR_ONE_SHOT
代表单次定时器；OS_OPT_TMR_PERIODIC代表周期定时器
        (OS_TMR_CALLBACK_PTR)depth_control_app_tmrcallback, // 定
时器回调函数
        0, // 回调函数参数
        &err);

    OSTmrStart(&DepthControlTmr, &err); // 启动定时器

    depth_control_app_stop();

    OS_CRITICAL_EXIT();
}

// 暂停运行
void depth_control_app_stop(void)
{
    CPU_SR_ALLOC();
```

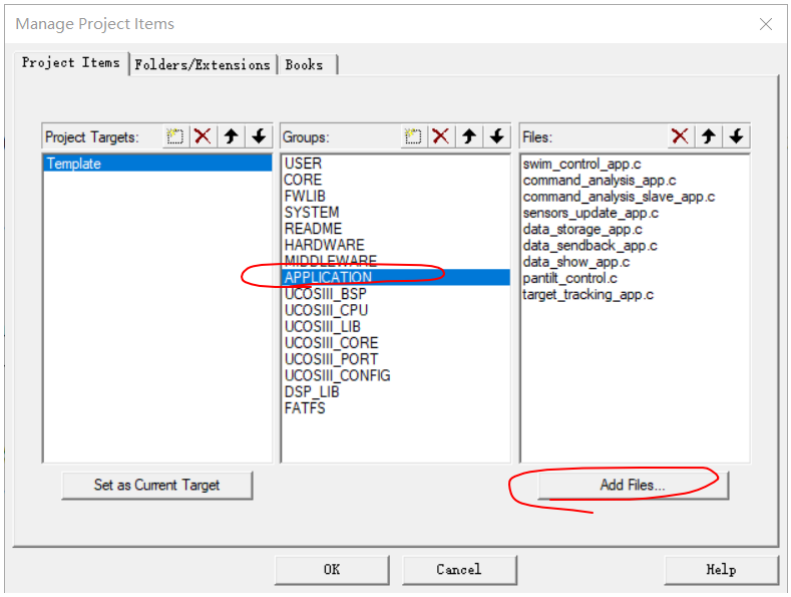
```
    OS_ERR err;

    OS_CRITICAL_ENTER();

    OSTmrStop(&DepthControlTmr, // 暂停定时器
        OS_OPT_TMR_NONE,
        0,
        &err);

    OSTaskSuspend(&DepthControlTCB, &err); // 任务挂起

    OS_CRITICAL_EXIT();
}

// 恢复运行
void depth_control_app_resume(void)
{
    CPU_SR_ALLOC();

    OS_ERR err;

    OS_CRITICAL_ENTER();

    OSTaskResume(&DepthControlTCB, &err); // 重启定时器

    OSTmrStart(&DepthControlTmr, &err); // 启动任务

    OS_CRITICAL_EXIT();
}
```

3.1.4 第四步

在工程中添加源文件，首先点击如下图标：

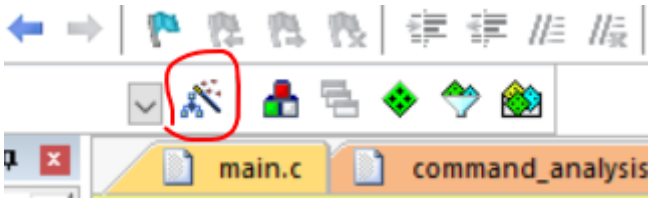


选中APPLICATION，然后点击Add Files选择depth_control_app.c文件

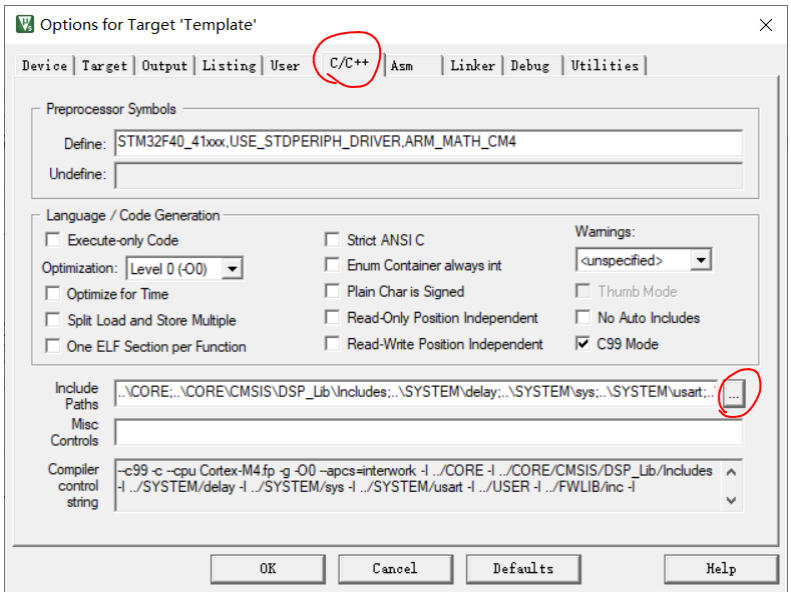


3.1.5 第五步

在工程中添加引用路径，首先点击如下图标：



选择C/C++，将DepthControl文件夹添加到Include Path



3.1.6 第六步

在main.h中添加该头文件

```
#include "depth_control_app.h"
```

在main.c文件中添加任务初始化代码：

```
void start_task(void *p_arg)
{
    ...
    depth_control_app_init();    // 深度控制，添加这一句
    ...
}
```

最后，编译就可以运行代码了。

3.2 获取传感器信息的方法

在你所定义的任务App的源文件中，添加如下语句：

```
extern BOXFISH boxfishstate;
```

然后，在任务主函数中，直接用如下语句就可以获取传感器信息：

```
float pitch = boxfishstate.onboard_imu_data.pitch;
float depth = boxfishstate.depth;
其余数据的方法类似
```

3.3 周期性的储存信息到SD卡的方法

首先，在data_storage_app.c文件的data_storage_app_task函数中，添加需要记录的数据项，例如：

```
// 系统时间
memcpy(databuf, &(boxfishstate.timestamp), 4);
// pitch
memcpy(databuf+4, &(boxfishstate.gimbal_imu_data.roll), 4);
// roll
memcpy(databuf+8, &(boxfishstate.gimbal_imu_data.pitch), 4);
// yaw
memcpy(databuf+12, &(boxfishstate.gimbal_imu_data.yaw), 4);
// 其他你所需要记录的数据
memcpy(databuf+16, &(你所需要记录的数据), (你所需要记录的数据的大小));
```

注意，要把数据复制到databuf中。

然后，要在datacenter.h中，修改DATA_SAVE_LENGTH的大小，例如这里总共16个字节的数据，所以DATA_SAVE_LENGTH就是16，即：

```
#define DATA_SAVE_LENGTH 16 // 储存数据的长度
```

此外，储存数据的频率，可以通过设置data_storage_app.h中的DATA_STORAGE_APP_TIMER_PERIOD_MS变量来设置。默认设置为每20ms记录一次。

3.4 采集新的传感器数据的方法

首先，如果有新的传感器接入系统的话，需要先编写该传感器的驱动，放在HARDWARE文件夹下。

其次，需要在robotstate.h中修改BOXFISH结构体的组成，添加新的传感器数据变量。

再次，需要修改main.c中的BOXFISH boxfishstate初始化的部分。

最后，需要在sensors_update_app.c中，把传感器的数据读取出来，放入boxfishstate结构体中。

注意，传感器数据的更新频率可以在sensors_update_app.h文件下的SENSORS_UPDATE_APP_TIMER_PERIOD_MS变量设置，默认是每10ms更新一次。如果有些传感器更新不用那么快，可以在任务主函数中，增加一层循环。

3.5 接收和扩展RFLink命令的方法

首先，需要在rflink.h文件下的Command的枚举类型中添加新的命令。例如，我要添加新命令为：SET_SWIM_RUN，那么，应该按如下设置：

```
typedef enum Command
{
    SHAKING_HANDS=1,
    ...
    SET_SWIM_RUN,
    ...
    LAST_COMMAND_FLAG
}Command;
```

然后，在command_analysis_app.c文件下，添加关于这一条指令的处理过程：

```
switch(command)
{
    ...

    case SET_SWIM_RUN:
        swim_control_start();
        #ifdef PRINT_COMMAND_EN
        BuffPrintf("SET_SWIM_RUN\n");
        #endif
        break;

    ...
}
```

最后，需要特别注意的是，当修改了单片机代码中的Command枚举类型以后，一定也要同步修改上位机中的Command类型，否则就会出现指令不匹配。

3.6 开启或关停某个任务App的方法

方法一：直接在main.c中注释掉这个任务App的初始化代码

方法二：调用*****_app_stop()方法

3.7 串口打印数据的方法

方法一：调用**MyPrintf()函数**发送数据。不过这个方式，数据量大时不可用，并且如果多个任务App都在发送数据时，也不可用。并且这种方法只可以发送回串口助手，上位机无法识别。 使用示例：(%d 用来替代整型数据，%s用来替代字符串，%f用来替代浮点型数据)

```
char name[] = "ZPF";
int id = 1;
float t = 0.01;
MyPrintf(" No %d is %s. It is ready at time %f.", id, name, t);
```

方法二：调用**BuffPrintf()函数**发送数据。这个方式带有一个发送缓冲区，可以发送数量较多的数据，并且多个任务App都在发送数据时，也互相不冲突。不过，这个方式同样只可以发送给串口助手，上位机同样无法识别。 使用示例：(%d 用来替代整型数据，%s用来替代字符串，%f用来替代浮点型数据)

```
char name[] = "ZPF";
int id = 1;
float t = 0.01;
BuffPrintf(" No %d is %s. It is ready at time %f.", id, name, t);
```

方法三：调用**RFLink_message()函数**发送数据。这个方式带有一个发送缓冲区，可以发送数量较多的数据，并且多个任务App都在发送数据时，也互相不冲突。此外，这个方式可以发送给串口助手，上位机也可以识别。 使用示例：(%d用来替代整型数据，%s用来替代字符串，%f用来替代浮点型数据)

```
char name[] = "ZPF";
int id = 1;
float t = 0.01;
RFLink_message(" No %d is %s. It is ready at time %f.", id, name,
t);
```

3.8 控制机器鱼游动的方法

首先，在你的任务App的源文件中添加对swim_control_app.h的应用，接下来就可以开始控制了

```
#include "swim_control_app.h"
```

控制尾巴摆动幅度，调用函数：

```
void set_tail_amp(float amp); // 这个函数输入参数单位是弧度
```

控制尾巴摆动频率，调用函数：

```
void set_tail_freq(float freq); // 这个函数输入参数单位是Hz
```

控制尾巴摆动偏置，调用函数：

```
void set_tail_offset(float offset); // 这个函数输入参数单位是弧度
```

控制胸鳍旋转角度，调用函数：

```
void set_pect_fin_angle(float angle, uint8_t channel); // 参数1单
位是弧度，参数2代表通道数，0代表左胸鳍，1代表右胸鳍
```

4 其他说明