MagicMatrix 软件结构说明

硬件 MMHardware

MMHardware 说明

> 将所有与硬件相关的操作封装，便于集中管理与使用

MMHardware 实现代码

```
class MMHardware {
private:
    /* data */
public:
    IRrecv irrecv; // 定义红外接收对象
    ThreeWire myWire; // 定义i2c通信模块
    RtcDS1302<ThreeWire> Rtc; // 定义rtc时钟模块
    DHT_Unified dht; // 定义dht温湿度模块
    // 构造函数
    MMHardware()
        : irrecv(PIN_IRR) // 构造irrecv
        , myWire(PIN_I2C_DAT, PIN_I2C_CLK, PIN_DS1302_CS) // 构造i2c通信
        , Rtc(myWire) // 构造rtc时钟模块
        , dht(PIN_DHT, DHT11) // 构造dht温湿度模块
    {
        irrecv.enableIRIn(); // 打开红外接收
        dht.begin(); // 打开dht温湿度传感器
    };
    // ~MMHardware();

    // IRRCode红外线读取到的结果代码
    uint16_t IRRCode()
    {
        uint16_t r = IRK_NONE;
        if (this->irrecv.decode()) { // 如果红外线读取到数据
            r = this->irrecv.decodedIRData.command;
            this->irrecv.resume();
        }
        return r;
    }

} mmhardware;
```

屏幕 FFScr

FFScr 说明

> 在矩阵屏幕的基础上将相关的操作进行封装，如：绘制像素颜色，刷新矩阵显示等

FFScr 实现代码

```
class MMScr {
public:
    // 清空但是不刷新
    void SetEmpty()
    {
        matrix.clear();
    }

    // 更新矩阵
    void Update()
    {
        matrix.show();
    }

    // 清空矩阵，并立刻显示
    void Clear()
    {
        SetEmpty();
        Update();
    }

    // 设置一个像素的颜色
    void SetPixel(uint16_t x, uint16_t y, uint8_t R, uint8_t G, uint8_t B)
    {
        matrix.drawPixel(x, y, matrix.Color(R, G, B));
    }

    // 填充整个屏幕
    void Fill(uint8_t R, uint8_t G, uint8_t B)
    {
        for (uint8_t r = 0; r < M_HEIGHT; ++r) { // 将所有颜色清空
            for (uint8_t c = 0; c < M_WIDTH; ++c) {
                matrix.drawPixel(c, r, matrix.Color(R, G, B));
            }
        }
    }

    // 将RamBmp绘制到屏幕
    void DrawRamBmp(MMRamBmp& rb)
    {
        for (uint8_t r = 0; r < M_HEIGHT; ++r) { // 将所有颜色清空
            for (uint8_t c = 0; c < M_WIDTH; ++c) {
                RGB t;
                rb.GetPixcel(c, r, t); // 定义临时像素并从RamBmp读取
                matrix.drawPixel(c, r, t.Color()); // 将像素绘制到矩阵屏幕
            }
        }
    }

} mmscr;
```

功能池 MMFuncPool

MMFuncPool 说明

> 将系统的功能以对象的方式存贮在一个功能池中、 使用功能ID进行调用 目的是为了统一调用接口方便扩展 在系统运行过程中可以对功能池进行动态维护

MMFuncPool 实现代码

```
class MMFuncPool {
private:
    // 功能块列表
    std::vector<MMFunc*> items;
public:
    // MMFuncPool();
    // 析构,释放功能列表
    ~MMFuncPool() {
        this->items.clear();
    };
    // 将功能块加入到池中
    void Append(MMFunc* mmf)
    {
        mmf->FPool = this;
        this->items.push_back(mmf);
    }

    // 根据功能块FID执行对应的功能
    uint16_t Exec(uint16_t fid)
    {
        uint16_t r = EXECR_ERROR; // 定义返回结果,默认为错误
        for (auto i : items) {
            if (i->FID == fid) {
                r = i->Exec();
                break;
            }
        }
        return r;
    }
};
```