MagicMatrix 软件结构说明

硬件访问类 MMHardware (Magic Matrix Hardware)

- 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;
```

屏幕使用类 MMScr (Magic Matrix Screen)

- MMScr 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

- MMScr 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 (Magic Matrix Function Pool)

- 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。同时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;
    }
};
```

功能块基础类 MMFunc (Magic Magic)

- MMFunc 说明

实现了功能块的基础功能，使用时需要从基础类派生出自定义功能类

- MMFunc 实现代码

```
class MMFunc {
public:
    uint16_t FID = 0; // 功能块ID
    uint8_t FType = 0; // 类型，共后期扩展使用
    // 所在功能池，使用方法：((MMFuncPool *)(this->FPool))->Exec(0);
    void* FPool = NULL;
    // 构造函数
    MMFunc() {};
    MMFunc(uint16_t fid)
    {
        this->FID = fid;
    }
    MMFunc(uint16_t fid, uint16_t ftype)
    {
        this->FID = fid;
        this->FType = ftype;
    }
    // virtual ~MMFunc(){};
    virtual uint16_t Exec() { return EXECR_ERROR; }; // 虚函数需要子类实现
};
```

- MMFunc 基础类的使用方法

  首先需要从MMFunc派生出自定义功能类

  注意：继承后必须实现虚方法Exec()，此方法是提供给功能池统一调用的接口

```
// 矩阵屏幕测试类，测试RGB显示
class MMF_MatrixTest: public MMFunc {
public:
    // 实现父类的虚方法
    MMF_MatrixTest(uint16_t fid): MMFunc(fid) {}
    virtual uint16_t Exec() {
        mmfill.MatrixTest();
        return EXECR_OK;
    }
};
```

- 定义功能块访问MMF_ID(Magic Matrix Function ID)并实例化功能块

  MMF_ID作为调用功能池中功能的索引不应重复

  如果功能池中出现重复MMF_ID的功能块则只会调用池中第一个功能块

  注：此处的MMF_TYPE(Magic Matrix Function Type)是功能块分类标志，为今后扩展多种功能块调用预留

```
// 定义功能类型,为后期扩展作准备,目前默认只有0
#define MMF_TYPE_0 (0x00)

// 定义矩阵测试功能
#define MMF_ID_MATRIXTEST (0x0001)
MMF_MatrixTest mmf_matrixtest(MMF_ID_MATRIXTEST);
```

- 将自定义功能块实例加入功能池

  通过调用MMFPool的Append方法可将自定义功能块加入功能池

  系统在MMFPSetup函数中将所有自定义功能块统一加入功能池

```
// 将功能模块加入功能池
void MMFPSetup()
{
    FPool.Append(&mmf_matrixtest);
    FPool.Append(&mmf_disptime_1);
}
```

- 调用方法

  使用功能池中的功能需要根据MMF_ID来调用功能池的ExecFunc方法

```
// 执行功能池中的矩阵测试功能
FPool.ExecFunc(MMF_ID_MATRIXTEST);
```

InquireDelay 询问等待类，事件回调机制

- InquireDelay 说明

  由于系统目前采用单线程模式，体统在功能之间切换时需要退出当前功能，功能在执行过程中需要询问当前系统，如果系统切换的功能块，用于在不同的功能中进行切换。

```
// 查询等待类型(接口),在执行过程中询问是否需继续执行当前功能,用于响应退出操作
// 传入的参数是等待的毫秒数,在功能块中应当使用IDelay替代delay()并对返回结果作出响应
// 询问等待的时间当不需要继续执行时可能不传入的时间短
// 返回为false是需要退出
class InquireDelay {
public:
    virtual bool Inquire()
    {
        return true;
    };
    virtual bool IDelay(unsigned long ms)
```

```
    {
        delay(ms);
        return Inquire();
    };
};
```

- main类实现InquireDelay

```
    // 实现InquireDelay方法
  virtual bool Inquire()
  {
      // 读取蓝牙
      while (UART_BLE.available()) {
          // String s = UART_USB1.readString();
          UART_USB.print(char(UART_BLE.read()));
      }
      this->CheckPIRR(); // 由于功能块内部需要调用红外线数据，取消此部分多线程处
理
      this->CheckMenu();
      return (NextMenuItem == CurrMenuItem && NextMenuCate == CurrMenuCate);
  }

  // 实现IDelay方法
  virtual bool IDelay(unsigned long ms)
  {
      bool r = true; // 默认结果为true
      unsigned long startm = millis(); // 记录函数开始时的运行毫秒数
      unsigned long pass = 0; // 逝去的毫秒数
      do {
          delay(10);
          if (!Inquire()) {
              r = false;
              break;
          }
          // 过去的时间
          pass = mmhardware.TickPassed(startm, millis());
      } while (pass < ms);
      return r;
  }

  // 返回初始位置，实现InquireDelay的方法
  // 注意这里只是将下一个菜单项设置为返回，需要功能自行退出
  virtual void GoHome()
  {
      this->NextMenuCate = 0; // 返回0分类的0项
      this->NextMenuItem = 0;
  }
```