VisualHMI - 自定义协议(被动)

1. 概述

VisualHMI 系统原生集集成多种主流工业通信协议的驱动模块，包括 **DCBUS、XGUS、Modbus（RTU/TCP，支持主/从模式）、三菱 FX2N、台达 DVP 以及西门子 PPI** 等。用户在开发过程中，仅需在 **工程配置** → **通讯协议** 中选择对应协议类型并完成参数设置，**无需编写底层通信代码。**

尽管 VisualHMI 内置了对多种主流工业协议的原生支持，但实际工业场景中仍广泛存在**私有通信协议**，，或通过**第二路（或多路）串口**进行额外通信（适用于具备双串口或多串口硬件配置的屏幕型号），则可通过 **LUA 脚本** 对串口进行底层自由控制，实现自定义协议的收发逻辑，从而满足灵活多样的通信需求。

本例程演示了基于 **系统回调机制** 实现私有串口协议解析的开发方式。当屏幕通过指定串口通道（`ch`）接收到数据时，系统会自动触发 `on_uart_recv(ch, packet)` 回调函数，**所传入的** `packet` **并不保证是一帧完整的协议数据**，而可能是：

- 仅包含一帧数据的**前几个字节**（分包）；
- 包含**多帧数据拼接在一起**（粘包）；

所以，需要开发者在该回调函数内部实现自定义协议的解析逻辑（如**帧头识别、长度校验、CRC 验证、数据提取**等），即可完成对私有协议的处理。这种方式属于 无需轮询串口状态，**事件驱动、被动触发** 的通信模型。

> 适用范围：VisualHMI - HMI&M系列&Dx系列
>
> 下载链接：[VisualHMI - 自定义协议-被动接收(点击下载)]

2. API说明

2.1.uart_setup(ch,baudrate,databits,stopbit,parity)

通过 `uart_setup(ch, baudrate, databits, stopbit, parity)` 函数可对指定串口通道进行通信参数初始化。

📊 **参数说明**

参数	类型	说明
`ch`	integer	**串口通道号** • `0`：主串口（通常用于标准协议如 Modbus） • `1, 2, 3...`：副串口（具体可用通道数量及编号依屏幕硬件型号而定）
`baudrate`	integer	**波特率：**单位 bps。 常用值：9600、19200、38400、57600、115200 等

参数	类型	说明
`databits`	integer	**数据位长度：** • `7`：7 位数据位 • `8`：8 位数据位（最常用）
`stopbit`	integer	**停止位** • `0`：1 位停止位 • `1`：1.5 位停止位 （注：部分平台可能不支持 1.5 位，实际以硬件为准）
`parity`	integer	**校验方式** • `0`：无校验（None） • `1`：奇校验（Odd） • `2`：偶校验（Even）

注意事项

- **主串口**：主串口（`ch = 0`）通常仅仅配置串口数据
- **副串口**：未调用 `uart_setup` 的副串口默认处于关闭状态，无法收发数据。

> ✅ 在 `on_init()` 中完成所有副串口的初始化，确保系统启动后即可使用。

2.2.on_uart_recv(ch,packet)

串口数据**被动接收回调函数。仅在以下条件下触发**：

- **主串口（`ch = 0`）**：需在工程配置中将通讯协议设置为 **"自定义"**，或在 Lua 脚本中调用 `set_free_protocol(1)` 启用自由协议模式；
- **副串口（`ch = 1, 2, ...`）**：需在脚本初始化阶段（如 `on_init()`）**调用** `uart_setup()` **完成串口参数配置**，系统才会将该通道交由用户脚本管理，并触发此回调。

> ⚠️ 若主串口已被系统内置协议（如 Modbus、XGUS、FX2N 等deng）占用，则主串口 `on_uart_recv` **不会被调用。**

📊 **参数说明**

参数	类型	说明
`ch`	integer	**串口通道号。** • `0`：主串口 • `1`, `2`, ...：副串口（具体可用通道数量依屏幕硬件型号而定）
`packet`	table	**接收到的原始字节数据数组**，以 Lua 表形式传递，**下标从 1 开始。** 示例：`{0xAA, 0x55, 0x01, 0x02}`

💡 **关键注意事项**

- **数据非帧对齐**：`packet` 可能包含**不完整帧（分包）**、**多帧拼接（粘包）** 或任意长度的数据片段，**用户必须自行实现缓冲与帧解析逻辑。**
- **非中断驱动**：该回调由底层的系统任务轮询触发，非硬件中断，应避免在回调中执行耗时操作。
- **副串口必须初始化**：未调用 `uart_setup(ch, ...)` 的副串口不会自动启用，也不会触发回调。

- **主串口（`ch = 0`）**：需在工程配置中将通讯协议设置为 **“自定义”**，或在 Lua 脚本中调用 `set_free_protocol(1)` 启用自由协议模式；
- **DCBUS 与 XGUS 协议对串口参数有固定要求**：
  - 停止位：**固定为 1 位**
  - 校验位：**固定为无校验（None）**

2.3.uart_send(ch,packet)

向指定串口通道发送字节数组

参数	类型	说明
`ch`	integer	**串口通道号。** • `0`：主串口（默认） • `1`, `2`, ...：副串口（具体可用通道依硬件型号而定） ⚠️ **该通道必须处于自由协议模式**（即未被 Modbus/XGUS 等系统协议占用）。
`packet`	table	**待发送的字节数组** 以 Lua 表形式传入，**下标从 1 开始。** 每个元素应为 `0~255` 的整数。 示例：`{0xAA, 0x55, 0x01, 0x02}`

2.4.set_free_protocol(en)

动态切换**主串口（通道 0）** 的协议模式，用于在系统内置协议与用户自定义协议之间灵活切换。

📊 **参数说明**

参数	类型	说明
`en`	integer	**协议模式使能标志**： • `1`：启用**自由协议模式**，主串口交由 Lua 脚本控制，数据收发通过 `on_uart_recv(ch, packet)` 和 `uart_send(ch, packet)` 处理； • `0`：恢复为**工程配置的原始协议**（如 Modbus、XGUS、FX2N 等），系统重新接管主串口通信。

3. 协议举例

本例程通过控制 LED 灯的**开关状态**、**亮度调节**和**设备名称修改**，演示如何在 VisualHMI 中使用 **LUA 脚本实现自定义串口协议通信**。系统与外设（如单片机控制的智能灯模块）通过主串口或副串口进行双向交互。

3.1 协议帧格式

通信采用固定帧结构，如 **下表** 所示：

方向	帧头 (1B)	长度 (1B)	功能码 (2B)	数据内容 (nB)	帧尾 (2B)
设备 → 屏幕	`0x5A`	从功能码到帧尾的字节数	`0x0001` ~ `0xFFFF`	可变长度	`0xFC` `0xFF`
屏幕 → 设备	`0xA9`	从功能码到帧尾的字节数	`0x0001` ~ `0xFFFF`	可变长度	`0xFC` `0xFF`

> 📌 **说明**：
>
> - **长度字段** = 功能码(2) + 数据内容(n) + 帧尾(2) = `n + 4`
> - 所有多字节字段（如功能码）均采用 **大端（Big-Endian）** 格式传输。

3.2 功能码定义

功能码（Hex）	功能描述	数据内容格式（示例）
`0x0001`	设置灯的开关	`0x00`（关），`0x01`（开）
`0x0002`	设置灯的亮度	`0x00` ~ `0xFF`（0=最暗，255=最亮）
`0x0003`	修改设备名称	ASCII 字符串（如 `{0x4C, 0x69, 0x67, 0x68, 0x74}` → "Light"）

4. 应用

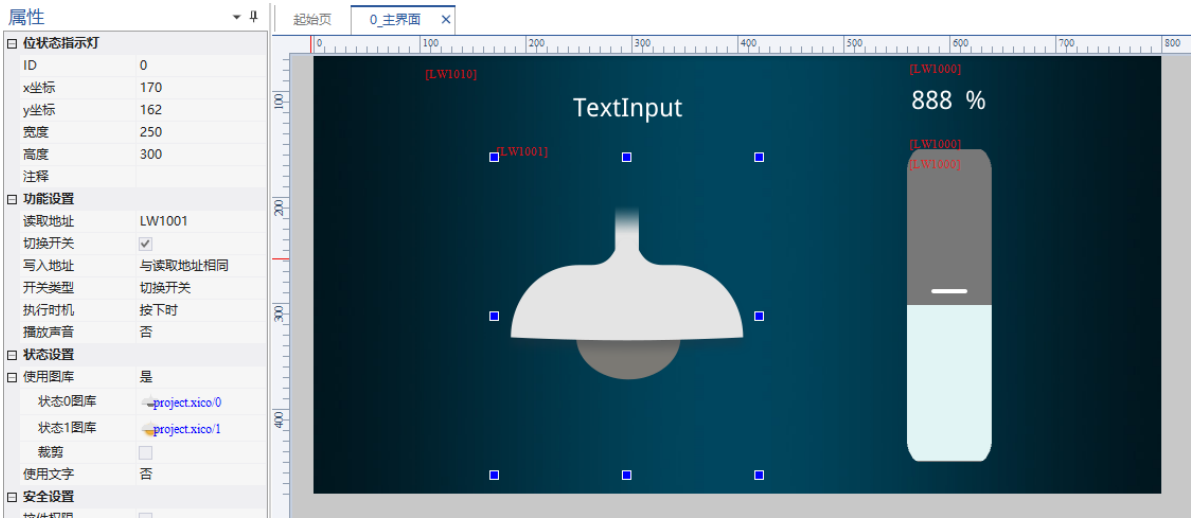
4.1.启用自定义协议

在 VisualHMI 工程开发环境中，，在工程配置中**启用自由协议模式。**具体操作路径如下：

> **工程配置 → 通讯协议 → 协议类型 → 选择“自定义”**

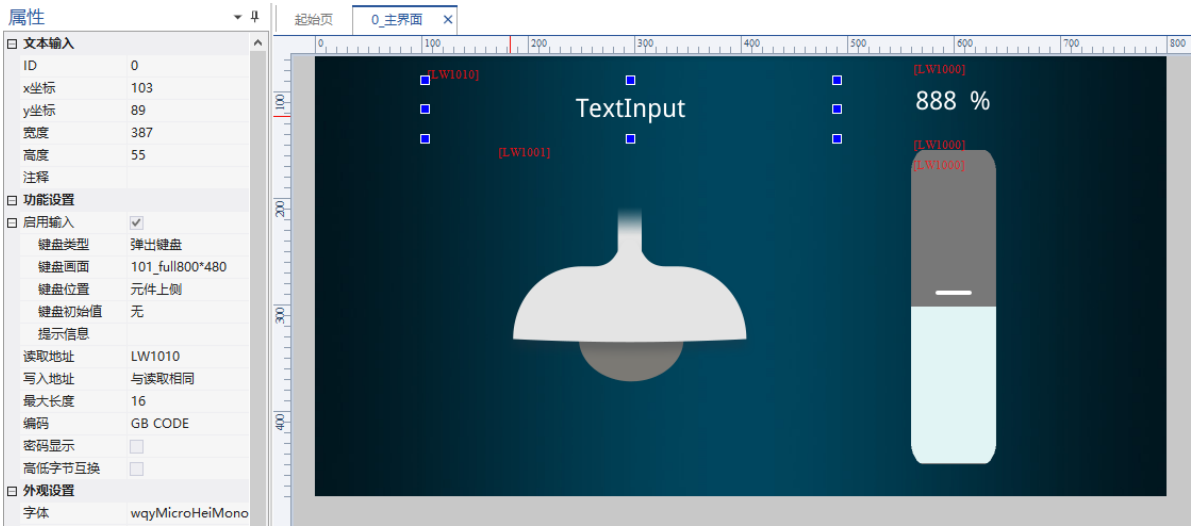
4.2.添加位状态指示灯控件

添加位状态指示灯控件：将该控件关联到 `LW1001` 寄存器。此寄存器用于控制灯的开关状态。即当寄存器值为 `0x00` 时表示灯关闭，值为 `0x01` 时表示灯开启



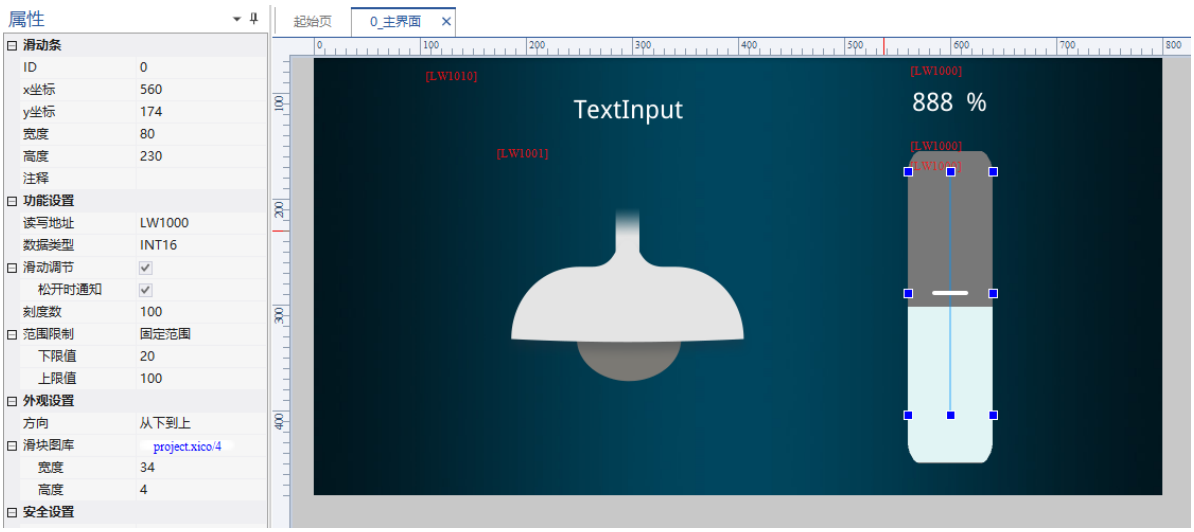
4.3.添加文本控件

添加文本控件：将文本控件关联到 `LW1010` 寄存器。这个寄存器用于修改灯的名称。输入与显示**：允许用户输入新的名称，并将其存储在 `LW1010` 寄存器中。同时，从该寄存器读取当前名称并在文本控件中显示。



4.4.添加数值、滑动、进度条控件

1. **添加数值控件**：添加一个数值输入控件，并关联到 `LW1000` 寄存器。这个寄存器用于控制灯的亮度。
2. **添加滑动条控件**：添加一个滑动条控件，并同样关联到 `LW1000` 寄存器。确保滑动条能够精确地控制亮度级别（例如，范围可以是从 `0x00` 到 `0xFF`）。
3. **添加进度条控件**：添加一个进度条控件，也关联到 `LW1000` 寄存器，以直观展示当前亮度水平。



5. 编辑Lua

5.1 串口接收

本实现采用**有限状态机（FSM）** 模型，按协议帧结构逐阶段解析数据，确保在粘包、分包或噪声干扰下仍能可靠识别完整帧。

5.1.1. 初始化状态：`IDLE`

目标：寻找有效帧头 `0x5A`。

1. 逐字节扫描输入数据；
2. 若遇到 `0x5A`，则：

- 将其存入缓冲区 `buff`；
- 切换状态为 `LENGTH`；
- 重置索引计数器。

3. 其他字节直接丢弃（自动跳过乱码或残留数据）。

> 📌 此设计确保即使通信中途断开或收到异常数据，系统也能**自动重新同步**到下一帧。

5.1.2. 解析长度字段：`LENGTH`

目标：读取第2字节（即“长度字段 L”），确定整帧预期长度。

- 第1字节：帧头 `0x5A`
- 第2字节：**L = 功能码(2B) + 数据(nB) + 帧尾(2B) = n + 4**
- 整帧总长度 = `1（帧头） + 1（L） + L = L + 2`
- 读取第2字节 → 得到 `L`；
- 校验 `L`,范围在`4 ≤ L ≤ 250`，防止非法值）；
  - 若非法 → 回退到 `IDLE`，丢弃当前帧；
  - 若合法 → 计算 `total_len = L + 2`，进入 `DATA` 状态。

> ⚠️ 注意：此处**不再拼接多字节长度**，因为协议定义长度字段为**单字节**。

5.1.3. 接收数据体：`DATA`

目标：持续接收后续字节，直到达到 `total_len`。

- 每收到一个字节，追加到 `buff`；
- 当 `buff`长度 == total_len时：
  - 校验帧尾：检查最后两个字节是否为 `{0xFC, 0xFF}`；
    - 若匹配 → 调用 `__ProtocolParse(buff)` 处理完整帧；
  - 若不匹配 → 视为无效帧，丢弃；
- **无论成功与否**，立即重置状态为 `IDLE`，准备接收下一帧。

> ✅ 优势：避免因一帧错误导致后续所有数据错位。

5.1.4. 缓冲区与状态重置

- 每次完成一帧解析（或判定失败）后，**清空缓冲区并回到** `IDLE`；
- 不依赖全局标记位（如原版的 `head_tag`），而是通过**明确状态转移**控制流程；
- 无内存泄漏风险（每次新帧都新建或重置 `buff` 表）。

接收函数代码，如下所示：

```
-- 协议常量
_UART1_HEAD_ = 0x5A
_UART1_TAIL_HIGH_ = 0xFC
_UART1_TAIL_LOW_  = 0xFF

-- 功能码定义
g_func = {
    onoff = 0x0001,
    light = 0x0002,
    name  = 0x0003
```

```lua
}

-- 串口通信对象
uart1 = {
    cmd = {
        buff = {},
        index = 0,
        state = "IDLE",
        total_len = 0
    }
}

-- 协议解析
function __ProtocolParse(buff)

    local fcode = (buff[3] << 8) | buff[4]

    local switch = {
        [g_func.onoff] = function()
            local val = buff[5]
            set_uint16(VT_LW, 0x1001, val)
        end,

        [g_func.light] = function()
            local val = buff[5]
            set_uint16(VT_LW, 0x1000, val)
        end,

        [g_func.name] = function()
            local name = ''
            for i = 5, #buff - 2
            do
                name = name .. string.char(buff[i])
            end
            set_string(VT_LW, 0x1010, name)
        end
    }

    local handler = switch[fcode]
    if handler
    then
        handler()
    end
end

-- 串口接收处理
function uart1.ComRecv(ch, packet)
    for i = 1, #packet
    do
        local byte = packet[i]
        local cmd = uart1.cmd

        if cmd.state == "IDLE"
        then
            if byte == _UART1_HEAD_
```

```
            then
                cmd.buff = { byte }
                cmd.index = 1
                cmd.state = "LENGTH"
            end

        elseif cmd.state == "LENGTH"
        then
            table.insert(cmd.buff, byte)
            cmd.index = cmd.index + 1

            local L = byte
            if L < 4 or L > 250
            then
                cmd.state = "IDLE"
            else
                cmd.total_len = 1 + 1 + L
                if cmd.total_len <= 2
                then
                    cmd.state = "IDLE"
                else
                    cmd.state = "DATA"
                end
            end

        elseif cmd.state == "DATA"
        then
            table.insert(cmd.buff, byte)
            cmd.index = cmd.index + 1

            if cmd.index == cmd.total_len
            then

                local len = #cmd.buff

                if len >= 3
                   and cmd.buff[len - 1] == _UART1_TAIL_HIGH_
                   and cmd.buff[len]     == _UART1_TAIL_LOW_
                then
                    __ProtocolParse(cmd.buff)
                end
                cmd.state = "IDLE"
            end
        end
    end
end
```

5.2 串口发送

为提升代码复用性与可维护性，**屏幕向设备发送指令时，应将协议帧构造逻辑封装为独立函数。**本例中，通过 `uart1.send(ch, fcode, data)` 实现统一的报文打包与发送。

当用户在 HMI 界面上操作控件（如切换开关、拖动亮度滑块、修改名称），系统会触发 `on_update(addr, vt, value)` 回调。开发者可在该回调中**判断寄存器地址**，并调用封装好的串口发送函数，将控制命令下发至外设。

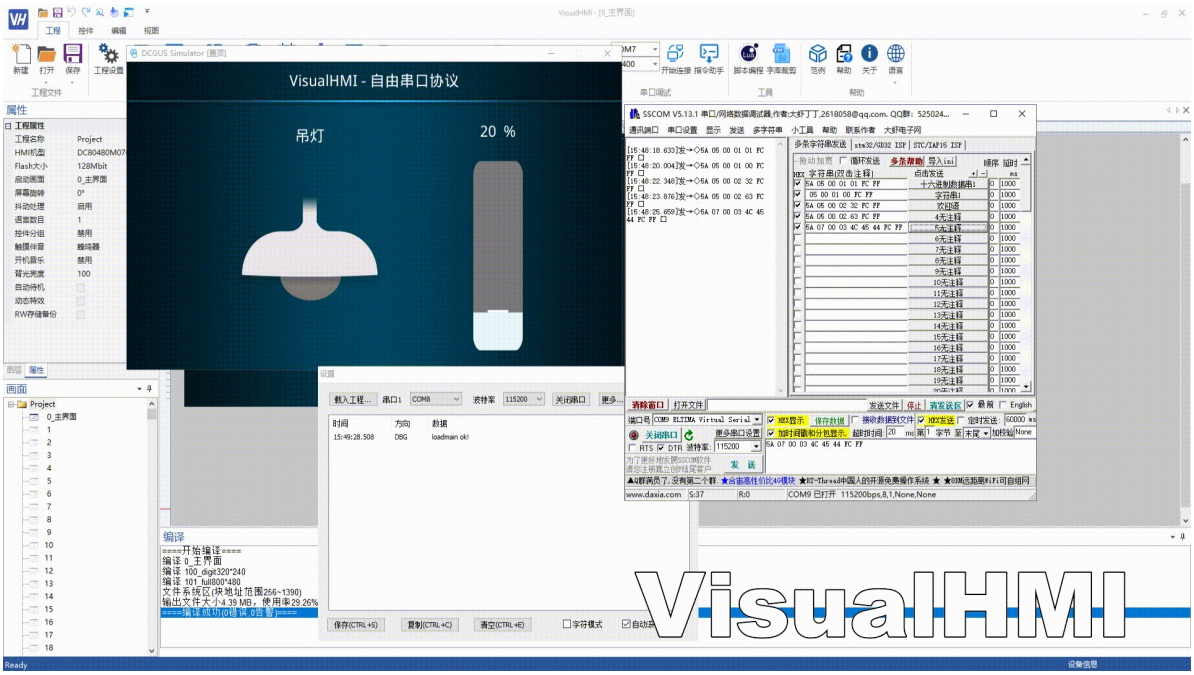
发送函数代码，如下所示：

```lua
--comm.lua
-- 发送协议帧（屏幕 → 设备，帧头 0xA9）
function uart1.dataSend(ch, fcode, data)
    local send_buff = {}

    send_buff[1] = 0xA9                     -- 帧头

    -- 长度字段暂留位置（1字节，在最后计算后填入）
    send_buff[2] = 0x00

    send_buff[3] = (fcode >> 8) & 0xFF      -- 功能码高字节
    send_buff[4] = fcode & 0xFF             -- 功能码低字节

    if fcode == 0x0001 or fcode == 0x0002
    then
        -- 开关/亮度：data 为数值，仅取低8位（1字节）
        send_buff[5] = data & 0xFF

    elseif fcode == 0x0003
    then
        -- 名称：data 为字符串，逐字节追加
        for i = 1, string.len(data)
        do
            send_buff[#send_buff + 1] = string.byte(data, i)
        end
    end

    -- 添加帧尾
    send_buff[#send_buff + 1] = 0xFC
    send_buff[#send_buff + 1] = 0xFF

    -- 计算长度：从功能码到帧尾的字节数 = 总长 - 2（帧头+长度）
    local datalen = #send_buff - 2
    send_buff[2] = datalen & 0xFF           -- 长度字段（1字节）

    uart_send(ch, send_buff)
end

--main.lua

function on_update(slave,vtype,addr)

    if _UART_UPDATA_ == 0 then return end

    if VT_LW == vtype
    then
        if addr == 0x1000
        then
            uart1.dataSend(0, g_func.light, get_uint16(vtype, addr))
        elseif addr == 0x1001
        then
            uart1.dataSend(0, g_func.onoff, get_uint16(vtype, addr))
        elseif addr == 0x1010
```

```
        then
            uart1.dataSend(0, g_func.name, get_string(vtype, addr))
        end
    end
end
```

> 📌 **协议说明**：
>
> - **长度字段为 1字节**，表示从功能码（第4字节）到帧尾（最后2字节）的总字节数

6. 运行预览

运行虚拟屏，模拟PLC/MCU发送指令控制屏幕，效果如下所示：

- **开关状态上报**：
  `5A 05 00 01 01 FC FF` （开）
  `5A 05 00 01 00 FC FF` （关）
- **亮度值上报**：
  `5A 05 00 02 32 FC FF` （亮度 = 50）
  `5A 05 00 02 63 FC FF` （亮度 = 99）
- **设备名称上报**：
  `5A 07 00 03 4C 45 44 FC FF` （名称 = "LED"）