VisualHMI - 文件读写

本例程中介绍Lua系统函数中的文件API读写，目前可支持读txt、csv、bin文件等。本文的文件主要介绍以下 2 点：

1. 读文件
2. 写文件
3. 文件遍历
4. 文件拷贝/删除

M系列存储设备标识：

- `'1:'` → **SD 卡**
- `'2:'` → **USB 设备（U盘）**
- `'3:'` → **HMI 内部 Flash 存储**

DH系列存储设备标识：

- `'/sdcard'` → **SD 卡**
- `'/udisk'` → **USB 设备（U盘）**
- `'/data'` → **HMI 内部 Flash 存储**

> 适用范围：VisualHMI - HMI&M系列&Dx系列
> 例程下载链接：《M系列文件读写》(点击跳转)

1.文件系统操作

1.1.dofile(name)

Lua 脚本模块加载函数，`dofile(name)` 是 Lua库提供，用于在运行时**加载并执行指定的** `.lua` **文件**。该功能支持将大型脚本工程**拆分为多个逻辑模块**（如 UI 控制、视频管理、通信协议等），提升代码可读性、可维护性与团队协作效率。

1.2.list_dir(path)

目录遍历函数，`list_dir(path)` 是 HMI 系统提供的**文件系统目录遍历接口**，用于**异步扫描指定存储设备或路径下的所有文件与子目录**。该函数通过回调机制 `on_list_dir(path, filename, type, fsize)` 逐项返回结果。

📊 **参数说明**

参数	类型	说明
`path`	string	目标路径

1.3.on_list_dir(path,filename,type,fsize)

目录遍历结果回调函数,`on_list_dir(path, filename, type, fsize)` 是 HMI 系统在调用 `list_dir(path)` 后**自动触发的异步回调函数**，用于**逐项返回指定目录下的文件与子目录信息**。该函数由系统在扫描过程中**每发现一个条目就调用一次**

📊 **参数说明**

参数	类型	说明
`path`	string	**当前被扫描的完整目录路径**
`filename`	string	**当前条目的名称（不含路径）** • 文件：`"rec_20260204.mp4"`
`type`	number	**条目类型标识** • `0` = 文件夹（Directory） • `1` = 普通文件（File）
`fsize`	number	**文件大小（单位：字节）** • 仅对 `type == 1` 有效 • 文件夹此项通常为 `0`（部分平台未定义）

1.4.file_open(path,mode)

文件打开函数，`file_open(path, mode)` 是 HMI 系统提供的**底层文件操作接口**，用于以指定模式**打开本地存储设备（SD卡、U盘、内部Flash）中的文件**。该函数是后续进行**文件读写**等操作的前提，返回布尔值指示操作是否成功。

📊 **参数说明**

参数	类型	说明
`path`	string	**完整文件路径** • 存储设备前缀：`"1:/xxx"`（SD卡）、`"2:/xxx"`（U盘）、`"3:/xxx"`（内部Flash） • 路径分隔符：必须使用 `/` • 示例：`"1:/config.txt"`、`"2:/logs/data.bin"`
`mode`	number	**打开模式：** 0：FA_READ，只读 1：FA_WRITE，覆盖写 2：FA_WRITE_ADD，追加写

1.5.file_close()

文件关闭函数，`file_close()` 是 HMI 系统提供的**文件句柄释放接口**，用于**关闭当前已打开的文件**，释放系统资源并确保数据完整写入存储设备。该函数是文件操作（`file_open` → 读写 → `file_close`）流程中**不可或缺的收尾步骤**，尤其在写入操作后调用可防止数据丢失或文件损坏。

📊 **参数说明**

项目	说明

项目	说明
参数	无
返回值	boolean
返回值含义	• `true`：文件成功关闭 • `false`：关闭失败（可能因文件未打开、I/O 错误等）
调用前提	必须已通过 `file_open()` 成功打开文件

1.6.file_size()

获取当前打开文件大小函数，`file_size()` 是 HMI 系统提供的**文件信息查询接口**，用于**获取当前已通过 `file_open()` 打开的文件的总字节长度**。该函数返回值为文件的精确字节数（单位：byte）。

📊 **参数说明**

项目	说明
参数	无
返回值	number
返回值含义	• ≥0：文件大小（字节）
调用时机	在 `file_open()` 成功后、`file_close()` 前调用

1.7.file_seek(offset)

文件读写位置定位函数，`file_seek(offset)` 是 HMI 系统提供的**文件指针重定位接口**，用于将当前已打开文件的**读写位置移动到指定字节偏移处**。

📊 **参数说明**

项目	说明
参数	`offset` (number) —— 目标位置的字节偏移量（≥0）
返回值	boolean
返回值含义	• `true`：成功将文件指针定位到 `offset` • `false`：失败（原因：文件未打开、`offset` 超出文件范围、只写模式限制等）
单位	字节（Byte）
起始位置	文件开头（0 = 第一个字节）

1.8.file_read(count)

文件内容读取函数，`file_read(count)` 是 HMI 系统提供的**文件数据读取接口**，用于从**当前已打开的文件中读取指定字节数的数据**，并以 **Lua 表（table）形式返回字节数组。**

📊 **参数说明**

参数	类型	说明
`count`	number	**要读取的字节数** 取值范围：1 ≤ count ≤ 2048

📊 **返回值说明**

项目	说明
成功返回	Lua 表（table），元素为字节值（0–255），下标从 1 开始
失败返回	`nil`：文件未打开、读取位置越界、I/O 错误、存储设备异常等

🔴 **内存安全警告**：
HMI 内存通常有限，读取 文件 文件会直接导致系统崩溃！

🔴 **重要内存安全规范：**
由于 HMI 设备内存资源极为有限（，**严禁将多次读取的数据累积存储于全局变量或长生命周期的缓冲区中。**典型错误做法包括：首次读取 2KB 数据存入全局 table，后续每次再读取 2KB 并追加至该 table。此类操作会导致内存持续增长，当处理大文件时，极易引发内存溢出，触发系统看门狗复位，造成设备无预警重启。

正确使用原则：
应采用**流式处理（streaming）模式**——每次调用 `file_read(2048)` 后，**立即对当前数据块进行解析、校验、写入或显示等处理，处理完毕即释放该数据块**，不得保留引用。确保任何时刻内存中仅存在一个不超过 2KB 的临时数据缓冲，从而保障系统长期稳定运行。

1.9.file_write(data)

文件写入函数，`file_write(data)` 是 HMI 系统提供的**底层文件写入接口**，用于将指定的字节数据写入当前已通过 `file_open()` 成功打开的文件中。

📊 **参数说明**

参数	类型	说明
`data`	table	**待写入的字节数组** • 元素必须为 **0 ~ 255 的整数（byte 值）** • 索引从 **1 开始连续排列** • 表长度必须满足 **1 ≤ #data ≤ 2048**

📊 **返回值说明**

返回值	类型	说明

返回值	类型	说明
成功	`true`	数据已成功提交至文件系统缓存
失败	`false`	写入失败，可能原因包括： • 文件未打开或已关闭 • 存储设备只读或未就绪 • 存储空间不足

🔴 **重要内存安全规范：**
HMI 设备内存资源极其有限，**严禁在写入前将整个大文件内容（如几 MB 的数据）预先生成并存入全局或局部 table 中。**典型错误做法包括：先构造一个包含数万字节的完整数据缓冲区，再分多次调用 `file_write()` 写入。此类操作在**数据准备阶段**就可能因内存分配过大而触发系统内存溢出，导致设备在实际写入前即发生无预警重启。

🔴 **正确使用原则：**
应采用**流式生成与写入（streaming write）模式**——每次仅准备不超过 2048 字节的待写数据块，立即调用 `file_write()` 写入文件，随后释放该数据块，再生成下一块。确保任意时刻内存中仅存在一个 ≤2KB 的临时写入缓冲，避免累积大尺寸数据结构。

此规范与 `file_read()` 的流式处理原则一致，是保障 HMI 系统在文件写入操作中稳定运行、防止因内存过载而崩溃的核心开发准则。

1.10.file_delete(path)

文件删除函数，`file_delete(path)` 是 HMI 系统提供的**文件系统管理接口**，用于**删除指定路径下的文件。**操作成功后，文件将从存储设备中永久移除，且**不可恢复。**

📊 **参数说明**

参数	类型	说明
`path`	string	**待删除的完整文件路径** • 必须包含设备标识：`"1:/"`（SD卡）、`"2:/"`（U盘）、`"3:/"`（内部Flash）

1.11.file_copy(src_path, dst_path)

文件复制函数，`file_copy(src_path, dst_path)` 是 HMI 系统提供的**文件系统操作接口**，用于将指定源路径的文件完整复制到目标路径，系统会自动触发**文件拷贝进度回调函数** `on_copy_file_process(status, filesize, transfersize)`

📊 **参数说明**

参数	类型	说明
`src_path`	string	**源文件的完整路径** • 必须包含设备前缀（如 `"1:/data.log"`） • 文件必须存在且未被占用 • 示例：`"1:/config.txt"`、`"3:/default.set"`

参数	类型	说明
`dst_path`	string	**目标文件的完整路径** • 必须包含设备前缀（可与源不同，如从 SD 卡复制到 U 盘） • 目标文件的父目录必须已存在（如 `"2:/backup/"` 需预先创建） • 若目标文件存在，通常会被覆盖

1.12.on_copy_file_process(status,filesize,transfersize)

文件拷贝进度回调函数，`on_copy_file_process(status, filesize, transfersize)` 是 HMI 系统在调用 `file_copy(src_path, dst_path)` 后**自动触发的异步进度回调函数**，用于实时反馈文件复制的状态与传输进度

📊 **参数说明**

参数	类型	说明
`status`	number	**拷贝状态码** • `0`：拷贝失败（如源文件不存在、目标空间不足等） • `1`：拷贝进行中 • `2`：拷贝成功完成
`filesize`	number	**源文件的总大小（字节）** • 在整个回调过程中保持不变 • 可用于计算进度百分比
`transfersize`	number	截至当前回调，已成功**复制的字节数** • 初始为 0，最终应等于 `filesize`（若成功） • 仅在 `status == 1` 时有效

1.13.mkdir(dir)

在 VisualHMI 平台中，`mkdir(dir)` 是系统提供的**同步文件夹创建接口**，用于在指定存储设备路径下创建一个新目录（文件夹）

📊 **参数说明**

参数	类型	说明
`dir`	string	完整目录路径，必须包含设备前缀和 `/` 分隔符

2.读文件

读取SD卡根目录的test1.txt、test2.txt文件，以字符串方式显示在数据记录控件上；

读取test1.bin、test2 .bin文件，以字节方式显示在数据记录控件上。

2.1.画面配置

在画面中添加1个位状态按钮，关联地址LW1050，用于选择读txt或bin文件；添加两个字状态按钮，均关联LW1501，其中设置1，读“test1.txt”文件，设置2，读“test2.txt”文件；添加一个数据记录控件，关联资料采集“文件读 字符串显示”。

同理，读bin文件添加两个位状态按钮和1个数据记录控件关联“文件读 数值显示”。

****PS:本章节不对资料采集、数据记录控件进行相关说明，详细可以参考相关文档*****

2.2.Lua 脚本

2.2.1.读取实现原理：流式分块 + 回调处理

💡 核心思想

> **不将整个文件加载到内存，而是每次只读取 ≤2048 字节的小块，立即交给用户回调函数处理，用完即弃。**

2.2.2.主要函数

组件	作用
`file.read_stream(filepath, on_chunk)`	入口函数：负责打开文件、分块读取、调用回调
`on_chunk(chunk, offset, is_last)`	用户定义的处理函数：对每一块数据做解析/显示/转发等
`file_read(count)`	HMI 底层 API：每次最多读 2048 字节，返回字节数组（table）

2.2.2.执行流程

1. **校验文件存在性** → 调用 `file.exists()`
2. **打开文件** → `file_open(path, 0)`
3. **获取总大小** → `file_size()`
4. **循环分块读取：**
   - 计算偏移：`offset = (i-1) * CHUNK_SIZE`
   - 定位指针：`file_seek(offset)`
   - 读取数据：`file_read(count)`（最后一块可能 <2048）
5. **调用用户回调**：`on_chunk(chunk, offset, is_last)`
6. **关闭文件** → `file_close()`

> 📌 整个过程中，**最大内存占用 ≈ 2048 字节（当前块）+ 少量局部变量**

💡2.2.3.读取 txt

- **目标文件**：`1:/test1.txt` 与 `1:/test2.txt`（位于 SD 卡根目录，设备前缀 `1:/` 表示 SD 存储）。
- **读取方式**：通过 **流式分块读取接口**（如 `file.read_stream()`），以 ≤2048 字节的块粒度逐段读取文件内容，避免一次性加载全文件至内存。
- 数据解析：
  - 将每一块字节数组（`table`）转换为字符串；
  - 维护**跨块行缓冲区**，确保换行符 `\n` 被正确识别，即使其跨越两个读取块；
  - 按完整行（line-oriented）进行分割与提交。
- **显示目标**：将解析出的每一行文本字符串，依次写入 **数据记录控件索引 0**（`record_write_string(0, line)`），实现逐行滚动显示。

```lua
--main.lua
    if addr == 0x1501
    then
        local idx = get_uint16(vtype, addr)
        if idx == 0 then return end

        local path = (idx == 1) and "1:/test1.txt" or "1:/test2.txt"
        record_clear(0)

        g_line_buffer = ""  -- 重置行缓冲

        file.read_stream(path, function(chunk, offset, is_last)
            -- 将字节块转为字符串
            local str = ""
            for _, b in ipairs(chunk) do str = str .. string.char(b) end

            -- 拼接到行缓冲
            g_line_buffer = g_line_buffer .. str

            -- 按行分割
            local lines = my_split(g_line_buffer, "\n")
            g_line_buffer = lines[#lines]  -- 保留不完整行

            -- 显示完整行
```

```
            for i = 1, #lines - 1
            do
                if #lines[i] > 0 then record_write_string(0, lines[i]) end
            end

            return true
        end)

        -- 显示最后一行（如果非空）
        if string.len(g_line_buffer) > 0 then record_write_string(0,
g_line_buffer) end

    elseif addr == ......
    end


--file.lua

-- 【安全】流式读取文件（核心接口）
-- @filepath: 文件路径
-- @on_chunk: 回调函数(chunk_table, offset, is_last)
--   - chunk_table: 当前块字节数组 (1~2048 字节)
--   - offset: 当前块起始偏移
--   - is_last: 是否最后一块
-- @return: true 成功, false 失败
function file.read_stream(filepath, on_chunk)
    if not file.exists(filepath) then return false end

    local ok = file_open(filepath, file.mode.read)
    if not ok then return false end

    local size = file_size()
    if size == 0 then
        file_close()
        return true
    end

    local blocks = math.ceil(size / CHUNK_SIZE)
    for i = 1, blocks do
        local offset = (i - 1) * CHUNK_SIZE
        if not file_seek(offset) then break end

        local count = (i == blocks) and (size % CHUNK_SIZE ~= 0 and size %
CHUNK_SIZE or CHUNK_SIZE) or CHUNK_SIZE
        local chunk = file_read(count)

        if not chunk or #chunk == 0 then break end

        local is_last = (i == blocks)
        if not on_chunk(chunk, offset, is_last) then
            -- 回调返回 false 可提前终止
            break
        end
        feed_dog()
    end
```

```
    file_close()
    return true
end
```

💡 2.2.4.读取bin

- **目标文件**：`1:/test1.bin` 与 `1:/test2.bin`（位于 SD 卡根目录）。
- **读取方式**：同样采用 **流式分块读取**，每次获取原始字节块（不进行字符集转换）。
- 数据解析：
  - 直接遍历字节数组中的每个元素（值域 0–255）；
  - 可按固定宽度（如每 16 字节）组织为一行，用于十六进制或数值化展示；
  - 不进行语义解释，保留原始二进制语义。
- **显示目标**：将字节序列（或格式化后的字节行）写入 **数据记录控件索引 1**（`record_write_data(1, byte_array)`）

```
--main.lua
    elseif addr == 0x1502 -- 读取二进制文件（流式 16 字节/行）
    then
        local idx = get_uint16(vtype, addr)
        if idx == 0 then return end

        local path = (idx == 1) and "1:/test1.bin" or "1:/test2.bin"
        record_clear(1)

        -- 重置全局缓冲区
        g_byte_buffer = {}

        file.read_stream(path, function(chunk, offset, is_last)
            -- 将当前 chunk 的所有字节追加到缓冲区
            for _, b in ipairs(chunk) do
                table.insert(g_byte_buffer, b)

                -- 每满 16 字节，提交一行
                if #g_byte_buffer >= 16
                then
                    local line = {}
                    for i = 1, 16 do line[i] = g_byte_buffer[i] end
                    record_write_data(1, line)

                    -- 移除已提交的 16 字节
                    for i = 1, 16 do table.remove(g_byte_buffer, 1) end
                end
            end

            return true
        end)

        -- 【可选】处理文件末尾不足 16 字节的数据 例如：用 0 填充或单独显示
        if #g_byte_buffer > 0
        then
             while #g_byte_buffer < 16 do table.insert(g_byte_buffer, 0) end
             record_write_data(1, g_byte_buffer)
```

```lua
        end


--file.lua
-- 【安全】流式读取文件（核心接口）
-- @filepath: 文件路径
-- @on_chunk: 回调函数(chunk_table, offset, is_last)
--   - chunk_table: 当前块字节数组 (1~2048 字节)
--   - offset: 当前块起始偏移
--   - is_last: 是否最后一块
-- @return: true 成功, false 失败
function file.read_stream(filepath, on_chunk)
    if not file.exists(filepath) then return false end

    local ok = file_open(filepath, file.mode.read)
    if not ok then return false end

    local size = file_size()
    if size == 0 then
        file_close()
        return true
    end

    local blocks = math.ceil(size / CHUNK_SIZE)
    for i = 1, blocks do
        local offset = (i - 1) * CHUNK_SIZE
        if not file_seek(offset) then break end

        local count = (i == blocks) and (size % CHUNK_SIZE ~= 0 and size %
CHUNK_SIZE or CHUNK_SIZE) or CHUNK_SIZE
        local chunk = file_read(count)

        if not chunk or #chunk == 0 then break end

        local is_last = (i == blocks)
        if not on_chunk(chunk, offset, is_last) then
            -- 回调返回 false 可提前终止
            break
        end
        feed_dog()
    end

    file_close()
    return true
end
```

❗性能与看门狗

- 大文件读取需调用 `feed_dog()`（当前已在循环中调用 ✅）。
- 避免在回调中执行耗时操作（如复杂计算、多次文件写入）。

3.写文件

3.1.画面配置

在画面中添加2个文本控件，关联LW1510、LW1520寄存器，用于键盘输入文件名称和内容；添加两个字状态按钮，均关联LW1503，设置1，创建新的文件写入；设置2，追加文件末尾写入，画面如下所示

3.2.Lua脚本

点击LW1053寄存器绑定的按钮，触发on_updata（...）函数，将获取文件名、文件内容，并写入SD卡，具体代码如下所示：

3.2.1.覆盖写（`mode = file.mode.create 0x01`）

- **语义**：若文件存在，则**清空原内容**；若不存在，则**新建文件**。
- **效果**：文件最终内容 = 本次写入的 `data`。

3.2.2.追加写（`mode = file.mode.append 0x02`）

- **语义**：若文件存在，在**末尾追加**新数据；若不存在，**新建并写入**。
- 底层流程：
  1. 调用 `file_open(path, 2)`；
  2. 调用 `file_size()` 获取当前长度 `size`；
  3. 调用 `file_seek(size)` 定位到文件末尾；
  4. 写入新数据；
  5. 关闭文件。
- **效果**：文件最终内容 = 原内容 + 本次写入的 `data`。

> 📌 注意：HMI 系统的 `file_open(mode=2)` **不会自动定位到末尾**！必须显式调用 `file_seek(file_size())`，否则会从开头覆盖写！

3.2.3.`file.write_chunk` 封装逻辑（关键代码解析）

```
function file.write_chunk(filepath, data, mode)
    -- 1. 模式校验
    if mode ~= file.mode.create and mode ~= file.mode.append then return false
end

    -- 2. 数据类型与长度校验（≤2048）
```

```lua
    local t = type(data)
    local len = (t == "string" or t == "table") and #data or -1
    if len <= 0 then return true end
    if len > CHUNK_SIZE then return false end  -- 防止内存溢出

    -- 3. 打开文件
    local ok = file_open(filepath, mode)
    if not ok then return false end

    -- 4. 【追加模式】必须手动定位到末尾!
    if mode == file.mode.append then
        local size = file_size()
        if not file_seek(size) then
            file_close()
            return false
        end
    end

    -- 5. 构建字节数组（string → byte table）
    local buf = {}
    if t == "string" then
        for i = 1, len do buf[i] = string.byte(data, i) end
    else
        for i = 1, len do buf[i] = data[i] end
    end

    -- 6. 调用系统 API 写入
    local ret = file_write(buf)
    file_close()
    return ret
end
```

⚠️ 3.2.4.关键注意事项（开发必读）

1. **仅支持小块写入（≤2048 字节）**
   - **严禁**传入大字符串（如日志拼接后 >2KB）；
   - 若需写入大内容，必须使用 **多次调用**
2. **追加写必须显式 seek 到末尾**
   - HMI 底层 `file_open(append)` **不等于 POSIX 的 O_APPEND**；
   - 若省略 `file_seek(file_size())`，数据将从偏移 0 开始写入，**覆盖原文件开头**！
3. **文件路径必须带设备前缀**
   - 正确：`"1:/config.txt"`（SD 卡）
   - 错误：`"config.txt"` 或 `"1:config.txt"`（缺少 `/`）
4. **写入内容为“全量替换”或“单次追加”**
   `write_chunk` 不是流式累积接口
   - 覆盖写：文件 = 本次 data；
   - 追加写：文件 = 原内容 + 本次 data；
   - 若需多次追加，需**多次调用** `write_chunk(..., append)`。
5. **字符串自动转字节数组**
   - 支持直接传入 Lua string（如 `"Hello"`）；
   - 内部自动转换为 `{72, 101, 108, 108, 111}`；

6. **断电异常**
    - 写入过程中断电 → 文件可能损坏或部分写入；

✅ 3.2.5.典型使用示例

场景 1：新建写

```
local name = get_string(VT_LW, 0x1510)
local msg  = get_string(VT_LW, 0x1520)
if not name or #name == 0 or not msg then return end

local path = "1:/" .. name .. ".txt"-- 直接写入（msg 应为小字符串）
local ret = file.write_chunk(path, msg..'\n',  1)
```

场景 2：追加写

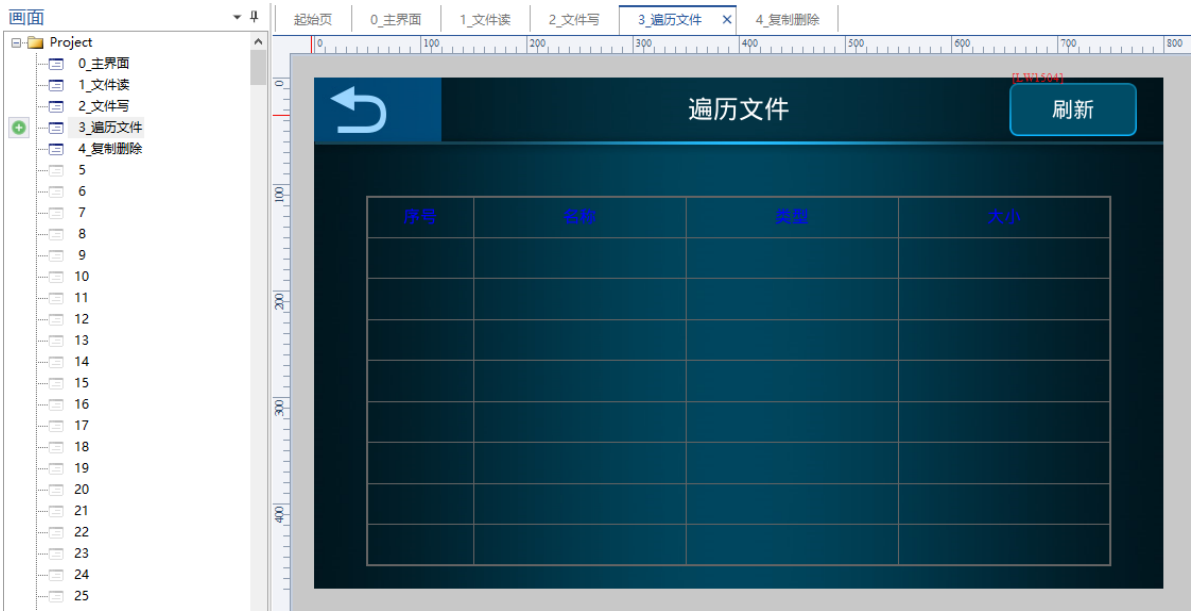
```
local name = get_string(VT_LW, 0x1510)
local msg  = get_string(VT_LW, 0x1520)
if not name or #name == 0 or not msg then return end

local path = "1:/" .. name .. ".txt"-- 直接写入（msg 应为小字符串）
local ret = file.write_chunk(path, msg..'\n',  2)
```

4.遍历文件

本章节将遍历SD卡根目录的文件，并把文件名称、文件名、文件大小显示在数据记录控件。

4.1.画面配置

在画面中添加1个字状态按钮，关联LW1504寄存器，用于刷新遍历SD卡根目录的内容到数据记录控件（关联“遍历文件”资料采集），画面如下所示



4.2.Lua脚本

用户触控触发 `LW1054` 寄存器写事件后，系统调用 `list_dir("1:")` 对 SD 卡根目录发起**异步根目录枚举**。HMI 运行时通过**回调驱动模型**，将每个目录项（文件或子目录）依次传递至全局回调函数 `on_list_dir(...)`。应用层在回调中完成类型识别、元数据格式化，并将结构化字符串输出至数据记录控件，实现文件系统的可视化呈现。

```
--main.lua

function get_extension(filename)
    local dot = string.find(filename, ".", 1, true)
    if dot then
        return string.sub(filename, 1, dot-1), string.sub(filename,
dot+1):lower()
    end
    return filename, "未知"
end

function on_list_dir(path, filename, type, fsize)
    local msg
    if type == 0 then
        msg = filename .. ";文件夹;;"
    else
        local sz
        if fsize >= 1024^3 then
            sz = string.format("%.2fG", fsize / 1024^3)
        elseif fsize >= 1024^2 then
            sz = string.format("%.2fM", fsize / 1024^2)
        elseif fsize >= 1024 then
            sz = string.format("%.2fKB", fsize / 1024)
        else
            sz = fsize .. "byte"
        end
        local name, ext = get_extension(filename)
        msg = name .. ";" .. ext .. ";" .. sz .. ";"
    end
    record_write_string(2, msg)
end


function on_update(slave,vtype,addr)

    if vtype == VT_LW
    then
        ......

        elseif addr == 0x1504 and get_uint16(vtype, addr) == 0x01
        then
            record_clear(2) --清除记录
            list_dir('1:')  --遍历SD卡文件

       ......
    end
end
```

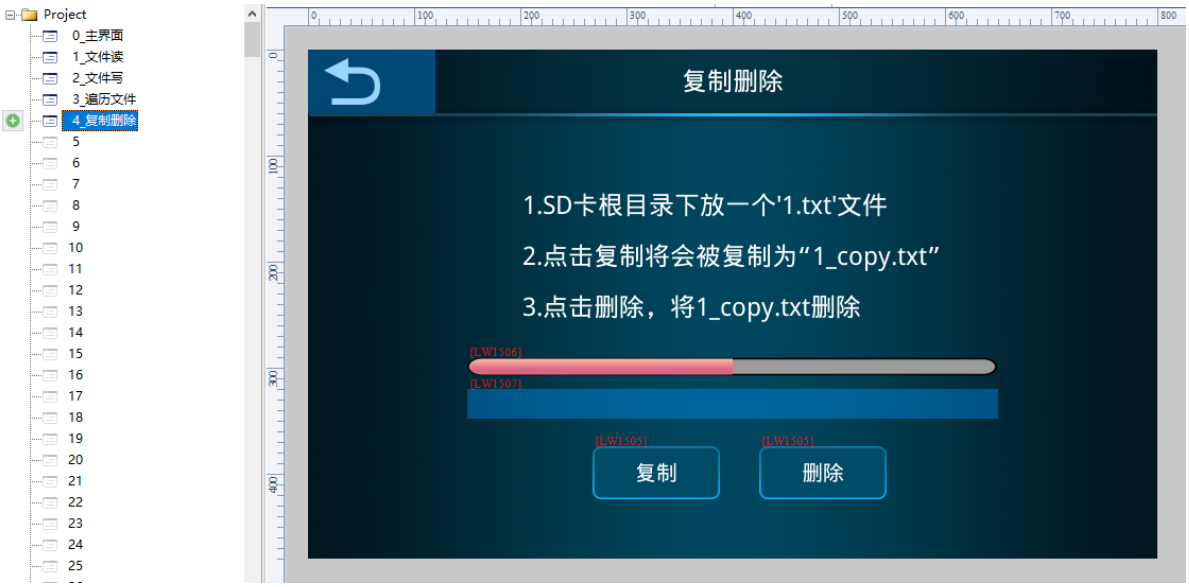
```
function on_update(slave, vtype, addr)
......
    elseif addr == 0x1504
    then
        if get_uint16(vtype, addr) == 1
        then
            record_clear(2)
            list_dir("1:")
        end
......
end
```

5.复制拷贝

本章节演示SD根目录的文件的复制、删除，将“1.txt”放在SD根目录，将会被复制为“1_copy.txt”，并且创建一个空的test文件夹；删除操作会把“1_copy.txt”删除，并且删除空文件夹test。

5.1.画面配置

在画面中添加2个字状态按钮，均关联LW1505寄存器。其中设置1，用于复制文件。设置2，用于删除；添加一个进度条控件，关联LW1506寄存器，用于显示拷贝进度；添加一个多状态指示灯，关联LW1507寄存器，用于显示拷贝、删除的结果，画面如下所示



5.2.Lua脚本

户点击“复制”按钮（LW1505=1）时，系统将 SD 卡根目录下的 `1.txt` 复制为 `1_copy.txt`，并创建空文件夹 `test`；点击“删除”按钮（LW1505=2）时，系统删除 `1_copy.txt` 和空文件夹 `test`。复制过程通过 `on_copy_file_process` 回调实时更新进度条（LW1506），操作结果通过多状态指示灯（LW1507）显示。

```
function on_copy_file_process(status, filesize, transfersize)
    if status == 0 then
        set_uint16(VT_LW, 0x1507, 3)
    elseif status == 1 then
```

```lua
        local p = math.floor(transfersize * 100 / filesize)
        set_uint16(VT_LW, 0x1506, p)
        refresh_screen()
    elseif status == 2 then
        set_uint16(VT_LW, 0x1506, 100)
        set_uint16(VT_LW, 0x1507, 2)
        refresh_screen()
    end
end


--main.lua
function on_update(slave,vtype,addr)

    if vtype == VT_LW
    then
        ......
            -- 文件操作
    elseif addr == 0x1505
    then
        local mode = get_uint16(vtype, addr)
        if mode == 1 then
            if file.exists("1:/1.txt")
            then
                file_copy("1:/1.txt", "1:/1_copy.txt")
            else
                set_uint16(VT_LW, 0x1507, 1)
            end
            mkdir('1:/test')

        elseif mode == 2
        then
            file_delete("1:/test")

            if file.exists("1:/1_copy.txt")
            then
                file_delete("1:/1_copy.txt")
                if not file.exists("1:/1_copy.txt")
                then
                    set_uint16(VT_LW, 0x1507, 4)
                else
                    set_uint16(VT_LW, 0x1507, 5)
                end
            else
                set_uint16(VT_LW, 0x1507, 1)
            end
        end

        refresh_screen()

    end
end
```