VisualHMI - Lua 脚本API函数接口

概述

在 **VisualHMI 平台（基于 Lua 5.3）** 的嵌入式环境中，**并非完整支持 Lua 5.3 的所有库函数。**平台对 Lua 标准库进行了**裁剪**，以适配资源受限的 MCU 环境，并集成了大量**HMI 专用 API。**

> 适用范围：VisualHMI - HMI&M系列&Dx系列
>
> 例程下载链接：ViusalHMI - Lua脚本.pdf

以下是 **VisualHMI 中 Lua 脚本实际可用的核心库函数分类说明**

✅ 一、基本语言特性（完全支持）

这些是 Lua 语言核心，通常全部可用：

- **控制结构**：`if/then/else`, `while`, `for`, `repeat/until`
- **函数定义**：`function ... end`
- **表（Table）基础操作**

```
t = {a=1, b=2}
t[1] = "hello"
#t      -- 获取长度（仅对数组部分有效）
```

- **算术/逻辑/关系运算符**：`+ - * / % ^`, `and or not`, `== ~= < > <= >=`
- **字符串字面量**：`"..."`, `'...'`, `[[...]]`

✅ 二、部分支持的标准库

1. `string` **库（常用函数支持）**

函数	是否支持	说明
`string.len(s)`	✅	字符串长度
`string.sub(s, i, j)`	✅	截取子串
`string.find(s, pattern, init, plain)`	✅	**不支持完整正则**，`plain=true` 模式安全
`string.match(s, pattern)`	✅	仅支持简单模式（如 `%d+`），复杂模式可能失效
`string.gsub(s, pat, repl, n)`	✅	用于**全局字符串替换**

函数	是否支持	说明
`string.format(fmt, ...)`	✅	**强烈推荐**，用于数值转字符串（如 `"%04X"`）
`string.char(n)` / `string.byte(s, i)`	✅	字符与 ASCII 转换
.....	.....	更多string库函数以实际为准

> ❌ **不支持**：`string.gmatch`, `string.rep`（长重复可能溢出）

---

2. `table` 库（基础操作支持）

函数	是否支持	说明
`table.insert(t, [pos,] val)`	✅	插入元素
`table.remove(t, [pos])`	✅	删除元素
`table.concat(t, sep, i, j)`	✅	数组拼接（高效）
.....	.....	更多table库函数以实际为准

> ❌ **不支持**：`table.unpack`

---

3. `math` 库（有限支持）

函数	是否支持	说明
`math.floor(x)`, `math.ceil(x)`	✅	向下/向上取整
`math.min(a,b,...)`, `math.max(...)`	✅	最值
.....	.....	更多math库函数以实际为准

> ❌ **不支持**：`math.log`, `math.exp`（除非明确需要）

❌ 三、通常被禁用的标准库

库	原因	替代方案
`io`	无文件系统或只读	使用平台专用 `file_open` / `file_read`（如有）
`os`	无操作系统	时间用 `get_date_time()`，延时用 `sys.delay_ms()`

库	原因	替代方案
`debug`	安全与资源限制	不可用于生产环境
`package` / `require`	单文件脚本模型	所有逻辑写入 `main.lua`
`coroutine`	实时性要求	用 `sys.timer_start` 实现异步

📌 四、总结开发原则

原则
✅ **优先使用平台 API**
✅ **用 `string.format` 处理数值转字符串**
⚠️ **避免大表/长字符串操作**
❌ **不依赖 `io` / `os` / `debug`**
🔍 **不确定时先测试**: 用 `print(pcall(function() ... end))` 验证函数是否存在

> **记住**：VisualHMI 的 Lua 是 **“Lua 语法 + 平台 API”** 的混合体，**不是通用 Lua 环境。**聚焦于平台提供的专用接口，才能高效、稳定地完成 HMI 逻辑开发。

💡 五、自动生成Lua

VisualHMI平台提供大量专用API接口，配合工程可以完成大部分的内部逻辑处理， MCU可以只参与数据传输部分，不参与逻辑处理。点击菜单工程栏→脚本编程, 默认在工程目录下生成main.lua，且自动定义变量的数据类型，假设工程当前**Modbus协议**，如下所示：

```lua
ENCRYPT_=0     --LUA脚本加密

--数据类型定义
VT_LW = 1     --变量地址
VT_RW = 2     --FLASH存储
VT_0x = 10    --线圈
VT_1x = 11    --输入点
VT_3x = 12    --输入寄存器
VT_4x = 13    --保持寄存器


function on_init()
end

function on_run(screen)
end

function on_update(slave,vtype,addr)
```

```
end

function on_draw(screen_id,control_id)
end
```

定义的数据类型：VT_0x线圈、VT_1x离散输入、VT_3x输入寄存器，VT_4x保持寄存器。在读(get_uint16/get_int16...)、写(set_uint16/set_int16)、on_update中vtype的变量类型，对应定义的变量，如下所示：

```
--数据类型定义
VT_LW = 1    --变量地址
VT_RW = 2    --FLASH存储
VT_0x = 10   --线圈
VT_1x = 11   --输入点
VT_3x = 12   --输入寄存器
VT_4x = 13   --保持寄存器


function on_init()
    set_bit(VT_0x,0x0000,1) --modbus 协议，设置线圈0x0000地址的值为1
    local val = get_bit(VT_1x,0x0000) --modbus 协议，获取离散输入0x0000地址的值
    set_uint16(VT_4x,0x1000, 1) --modbus 协议，设置保持寄存器0x1000地址的值为1
    local val = get_uint16(VT_3x,0x1000) --modbus 协议，获取输入寄存器0x1000的值
end

function on_update(slave,vtype,addr)
    if slave == 0
    then
        if vtype == VT_4x
        then
            .....
        end
    end
end
```

1.常用回调函数

1.1.on_init()

系统加载LUA脚本文件之后，立即调用此回调函数，通常用于执行初始化操作

```
ENCRYPT_=0    --LUA脚本加密

--数据类型定义
VT_LW = 1    --变量地址
VT_RW = 2    --FLASH存储
VT_0x = 10   --线圈
VT_1x = 11   --输入点
VT_3x = 12   --输入寄存器
VT_4x = 13   --保持寄存器

function on_init()
```

```
    dofile('test.lua')--分多文件编辑,加载其他lua文件
    set_uint16(VT_4x, 0x0000, 1)--初始化寄存
end
```

1.2.on_run(screen)

周期性回调函数，`on_run(screen)` 是 HMI 系统提供的**核心周期性任务调度入口**，由系统在每个主循环周期**自动调用**，用于执行需要定期更新的逻辑，如实时数据显示、状态轮询、后台监控等

参数名	类型	说明
`screen`	number	**当前画面**： 用于区分不同页面的逻辑分支

❗ **问题**:

- 🔴 如HMI作为Modbus RTU Master 时，在`on_run`高频写入从机设备寄存器，造成通信负载过高，可能引发总线堵塞、设备响应延迟
- 🔴 不可在 `on_run` 中使用 `delay_ms()`、长循环、网络同步请求等
- 🔴 **严禁在 `on_run` 内部调用 `set_run_cycle()`**（可能导致调度器死锁）

💡 **示例：** 即在on_ru设置从机寄存器，需要**条件触发式写入**，如下所示:

```
function on_run(screen)
    local start   = get_uint16(VT_4x, 0x1000)  -- 启动标志
    local target  = get_uint16(VT_4x, 0x1001)  -- 目标值
    local current = get_uint16(VT_4x, 0x1002)  -- 当前值

    -- 仅当设备已启动且目标 ≠ 当前值时才操作
    if start == 1 and target ~= current then
        set_uint16(VT_4x, 0x1002, target)
        -- 可选：加一次写入保护，避免重复写
    end
end
```

1.3.on_update(slave, vtype, addr)

HMI 系统提供的**变量变更事件回调函数**，当**用户操作或脚本逻辑**修改了受监控的寄存器/变量值时，系统自动触发该回调。该机制实现了 **"数据驱动"** 的编程模型：**无需轮询，仅在数据变化时响应**，极大提升系统效率与实时性。

- slave：站号索引，0开始
- vtype：变量类型，生成main.lua，自动定义变量的数据类型
- addr：变量地址

参数名	类型	说明
`slave`	number	**站号索引**，0开始
`vtype`	number	**变量类型**，生成main.lua，根据协议自动生成数据类型，如 Modbus RTU 的 `VT_0x`（线圈）、FX3U的 `VT_M`（内部继电器）等

参数名	类型	说明
`addr`	number	**寄存器地址**

✅ **会触发 `on_update` 的场景**

场景	说明
用户界面操作	点击按钮、滑动条、输入框等控件修改绑定寄存器
其他回调on_xx调用 `set_xxx()`	在 `on_init`、`on_run`、`on_draw`、`on_screen_change` 等回调中调用写入函数

❌ **不会触发 `on_update` 的场景**

场景	说明
串口/Modbus 主站写入	外部设备通过通信协议直接写入从机寄存器（如 PLC 修改寄存器值）
`on_update` 内部调用 `set_xxx()`	防止无限递归，系统自动屏蔽嵌套触发

💡 **示例**

如下所示，以Modbus 协议为例，当其他回调函数设置寄存器，均不想触发on_update回调，可以定义全局变量如下处理：

```
ENCRYPT_=0     --LUA脚本加密

EN_ON_UPDATE_API_CB = 1 --全局变量，其他回调函数设置寄存器时，为1执行，为0直接退出

--数据类型定义
VT_LW = 1     --变量地址
VT_RW = 2     --FLASH存储
VT_0x = 10    --线圈
VT_1x = 11    --输入点
VT_3x = 12    --输入寄存器
VT_4x = 13    --保持寄存器

function on_init()
    EN_ON_UPDATE_API_CB = 0

    --user code
    set_uint16(VT_LW, 0x1000, 0x55AA)

    EN_ON_UPDATE_API_CB = 1
end

function on_run(screen)
    EN_ON_UPDATE_API_CB = 0
    --user code
    EN_ON_UPDATE_API_CB = 1
end
```

```
function on_update(slave, vtype, addr)
    if EN_ON_UPDATE_API_CB == 0
    then
        return
    end

    if slave == 0
    then
        if vtype == VT_4x --保持寄存器
        then
            if addr == 0x1000
            then
                local val = get_uint16(VT_4x, addr)
                if val == 1
                then
                    set_uint16(VT_4x, 0x0001, val)
                end
            end
        elseif vtype == VT_0x --线圈
        then
            if addr == 0x0000
            then
                local val = get_uint16(VT_0x, addr)
                if val == 1
                then
                    set_uint16(VT_0x, 0x0001, val)
                end
            end
        end
    end

    if vtype == VT_LW --内部寄存器
    then
        if addr == 0x1000
        then
            local val = get_uint16(VT_LW, addr)
            if 0x55AA == val
            then
                 set_uint16(VT_4x, 0x0000, val)
            end
        end
    end
end

function on_draw(screen_id,control_id)
    EN_ON_UPDATE_API_CB = 0

    --user code

    EN_ON_UPDATE_API_CB = 1
end

function on_screen_change(screen)
    EN_ON_UPDATE_API_CB = 0
```

```
    --user code
    if screen == 1
    then
        set_bit(VT_0x, 0x0000, 1)
    end

    EN_ON_UPDATE_API_CB = 1
end
```

1.4.on_screen_change(screen)

画面切换事件回调函数，在**每次画面切换完成之后**由系统自动调用。该函数用于执行与画面切换相关的**初始化、状态同步、资源加载或业务逻辑处理。**

参数名	类型	描述
screen	number	当前切换到的目标画面ID

1.5.on_press(state, x, y)

系统提供的**全局触摸事件回调函数**，用于捕获用户在屏幕上的**原始触摸操作。**该函数由系统在检测到触摸状态变化时自动调用，**每 100 毫秒最多触发一次**

参数名	类型	说明
`state`	number	**触摸状态标识** • `1`：手指按下 • `2`：长按触发（ • `0`：手指抬起 • 注意：不会重复发送 `1`，仅在按下瞬间触发一次
`x`	number	**触摸点 X 坐标（像素）**
`y`	number	**触摸点 Y 坐标（像素）**

💡**注意**：此函数提供的是**底层坐标事件**，不依赖于具体控件，即使点击空白区域也会触发。

1.6.on_usb_inserted(driver)

系统提供的**U盘插入回调函数**，当用户将 U 盘（或 USB 存储设备）**成功插入 HMI 设备的 USB 接口，并被系统识别后**，系统自动调用此函数。建议用全局变量来记录盘符

参数名	类型	说明
`driver`	string	M系列U盘盘符，为`2:`

💡**示例**

```
g_usb_disk = ''
function on_usb_inserted(driver)
    g_usb_disk= drive..'/'
end

-- 具体文件路径，如U盘目录下的1.txt
>g_usb_disk..'1.txt'

--访问具体文件，建议先判断盘符g_usb_disk是否合法
if g_usb_disk ~= ''
then
    --usercode
end
```

1.7.on_usb_removed()

系统提供的**U盘拔出事件回调函数**，当用户**移除 U 盘**后，系统检测到设备断开时自动调用此函数。

示例

```
function on_usb_removed()
    g_usb_disk= ''
end
```

1.8.on_sd_inserted(dir)

系统提供的**SD卡插入回调函数**，当用户将 SD卡**成功插入 HMI 设备的SD卡接口，并被系统识别后**，系统自动调用此函数。建议用全局变量来记录盘符

参数名	类型	说明
`dir`	string	M系列SD卡盘符，为`1:`

示例

```
g_sd_disk = ''
function on_sd_inserted(dir)
    g_sd_disk= dir..'/'
end

-- 具体文件路径，如SD目录下的1.txt
>g_sd_disk..'1.txt'

--访问具体文件，建议先判断盘符g_sd_disk是否合法
if g_sd_disk ~= ''
then
    --usercode
end
```

1.9.on_sd_removed()

系统提供的**SD拔出事件回调函数**，当用户**移除 SD**后，系统检测到设备断开时自动调用此函数

💡 **示例**

```
function on_sd_removed()
    g_sd_disk= ''
end
```

1.10.on_parse_timestamp(screen,control,timestamp)

系统提供的**时间显示格式化回调函数**，专用于**告警记录、数据记录、操作记录表格等内置时间字段的自定义格式。**

📊 **参数说明**

参数	类型	说明
`screen`	number	目标画面 ID，用于区分不同页面的时间格式需求
`control`	number	控件 ID，通常为**告警显示、数据记录、操作记录**控件ID
`timestamp`	number	32 位 Unix 时间戳（秒级），表示自 1970-01-01 00:00:00 UTC 起的秒数

仅在满足以下条件时才会被系统调用：

> 🔑 `control` **必须 ≠ 0**，控件类型：告警显示、数据记录、操作记录
>
> 即：只有所属的**控件ID具有有效 ID（非 0）** 时，系统才会触发此回调。

💡 **示例**

```
function on_parse_timestamp(screen,control,timestamp)
    local year,mon,day,hour,min,sec = make_datetime(timestamp)
    local new_timestamp = ''
    if screen == 0 and control == 1
    then
        return (string.format('%04d-%02d-%02d %02d:%02d:%02d',
 year,mon,day,hour,min,sec))
    elseif  screen == 2 and control == 1
    then
        return (string.format('%04d-%02d-%02d %02d:%02d:%02d',
 year,mon,day,hour,min,sec))
    end
end
```

2.读写寄存器函数

2.1.set_notify(enable)

`set_notify(enable)` 是 HMI 系统提供的**全局串口发送通知开关函数**，用于**临时启用或禁用变量变更时的自动通信行为。**

- 当 `set_notify(0)` 时：后续通过 `set_xxx()` 修改变量**不会触发协议栈向从机（如主板、PLC）发送串口指令**
- 当 `set_notify(1)` 时：恢复默认行为，变量修改会**自动通过串口/总线下发**

📊 **参数说明**

参数	类型	说明
enable	number	0禁止，1使能(默认值)

💡 **示例**

```
--如DCBUS协议下，设置0x1000地址，不通过串口下发给主板：
set_notify(0) -- 禁止通知
set_uint16(VT_LW, 0x1000, 1)--设置参数
set_notify(1)-- 启用通知
```

2.2.select_slave(slave_id)

HMI 在**多从机通信模式**（如 Modbus RTU等总线协议）下，通过`select_slave`，**临时指定从站索引**，使得后续的 `get_xxx()` / `set_xxx()` 操作时，访问指定从机寄存器。

> ✅ **核心作用**：切换指定从站，访问对应从站寄存器。

📊 参数说明

参数	类型	说明
`slave_id`	number	**从机索引** 0 起始，对应工程中配置的从机列表顺序，非 Modbus 站号

术语	含义	示例
从机索引 (slave_id)	HMI 内部从机列表的数组下标（0, 1, 2...）	`0` = 第一个从机
Modbus 站号 (Slave Address)	从机设备的物理通信地址	`10`, `33`

💡 **映射关系由 HMI 工程配置决定**：

- `slave_id = 0` → 站号 = 10
- `slave_id = 1` → 站号 = 33

💡 **示例**

如Modbus RTU协议中，总线有2个从机，第一个从机站号10，第2个从机站号为33，可封装如下函数：

```
-- 全局：从机在线状态掩码地址（由HMI系统定义）
local SLAVE_ONLINE_MASK_ADDR = 0x01A3

-- 安全读取保持寄存器
function mb_read_holding(slave_index, reg_addr)
    -- 检查从机是否在线（bitN = 1 表示在线）
    local online_mask = get_uint16(VT_LW, SLAVE_ONLINE_MASK_ADDR)
    if (online_mask >> slave_index) & 1 == 0 then
        return nil, "offline"  -- 返回错误状态
    end

    select_slave(slave_index)
    return get_uint16(VT_4x, reg_addr), "ok"
end

-- 安全写入保持寄存器
function mb_write_holding(slave_index, reg_addr, value)
    local online_mask = get_uint16(VT_LW, SLAVE_ONLINE_MASK_ADDR)
    if (online_mask >> slave_index) & 1 == 0 then
        return false
    end

    feed_dog()  -- 防止看门狗复位（长操作时必要）
    select_slave(slave_index)
    set_uint16(VT_4x, reg_addr, value)
    return true
end

-- 读取从机0（站号10）的0x1000寄存器
local val, status = mb_read_holding(0, 0x1000)
if status == "ok" then
    print("Value:", val)
else
    print("From slave offline")
end

-- 写入从机1（站号33）的0x2000寄存器
mb_write_holding(1, 0x2000, 12345)
```

2.3.set_endian(en)

变量字节序（大小端）设置函数。 `set_endian(en)` 是 HMI 系统在**主从机通信模式**（如 Modbus、FX3U 等）下提供的**全局字节序控制函数**，用于指定多字节数据（如 `uint16`、`uint32`、`float`）在通过串口/总线与从机交换时的**字节排列顺序。**

📊 **参数说明**

参数	类型	取值	说明
`en`	number	`0`	大端模式（Big-Endian） —— 高位字节在前（默认）
		`1`	小端模式（Little-Endian） —— 低位字节在前

该设置直接影响 `set_xxx()` 和 `get_xxx()` 函数在读写寄存器时的**底层字节打包/解析方式。**

> ✅ **仅在通信协议涉及多字节数据传输时生效**，对纯内部变量（如 `VT_LW`）通常无影响。

2.4.set_bit(vtype, addr, value， count)

多位寄存器批量写入函数，`set_bit(vtype, addr, value, count)` 是 HMI 系统提供的**批量位寄存器写入函数**，用于**一次性设置连续多个位寄存器（bit）或单个位寄存器（bit）的状态**，适用于 Modbus RTU的线圈（Coils）、FX3U 的 X/Y/S/M 等位寄存器。

📊 **参数说明**

参数	类型	说明
`vtype`	number	**变量类型：**如 Modbus RTU 的`VT_0x`（线圈）、FX3U的`VT_M`（内部继电器）等
`addr`	number	寄存器地址
`value`	number	**16 位无符号整数值** 其低 `value` 表示目标位状态（bit0 → 地址 `addr`，bit1 → `addr+1`，...）
`count`	number	**选填，写入位数** 取值范围 1 ~ 16

📣 通信触发机制（关键行为）

> **调用 `set_bit()` 并不总是立即发送串口报文。**其通信行为受以下两个条件共同约束：

✅ **条件 1 :HMI 处于主机模式（Master Mode）**

- 仅当 HMI 配置为 **Modbus 主站 、DCBUS或XGUS**，如配置为**Modbus RTU Master**, 才会**主动发送串口报文**；
- 若为从机模式**Slave**，此API仅更新本地镜像，**不会发送任何报文。**

✅ **条件 2：串口通知未被禁止**`set_notify`

- 系统提供 `set_notify(enable)` 接口用于**临时抑制通信输出**
- **仅当通知使能（`set_notify(1)` 或默认状态）时**，此API才会**主动发送串口报文**；
- 若通知被禁用（`set_notify(0)`），函数仅更新本地缓存

1. 示例：批量下发， Modbus

```
-- 构建16位控制字（仅低13位有效）
local coils = 0
coils = coils | (get_uint16(VT_LW, g_addr['急停关机']) << 0)    -- bit0: 急停
coils = coils | (get_uint16(VT_LW, g_addr['枪锁']) << 2)        -- bit2: 电子锁
coils = coils | ((get_uint16(VT_RW, g_flash['波特率选择']) > 0 and 1 or 0) << 12)
-- bit12: 波特率标志

-- 批量写入13位到线圈区 0x00B6
select_slave(target_slave)
set_bit(VT_0x, 0x00B6, coils, 13)  -- 一次通信完成13个开关控制
```

2. 示例：单个下发， Fx3U

```
set_bit(VT_M, 100, 1)
```

2.5.get_bit (vtype, addr)

位寄存器读取函数。`get_bit(vtype, addr)` 是 HMI 系统提供的**位读取函数**，用于读取单个**位地址**的状态**（布尔量：0 或 1）。该函用于读取** Modbus 线圈/输入。

📊 **参数说明**

参数	类型	说明
`vtype`	number	**变量类型：**如 Modbus RTU 的`VT_0x`（线圈）、FX3U的`VT_M`（内部继电器）等
`addr`	number	**寄存器地址**
返回值	number	0 或 1，表示该位的当前状态

> 🔑 `get_bit(vtype, addr)` **是从 HMI 的本地缓存（内存镜像）中读取数据，而非实时向 PLC/设备发起串口请求。**

🔴 **工作机制：**

1. **HMI 后台任务** 按照工程中配置的**轮询周期**，ModbusRTU为例，自动从 PLC 读取 `VT_0`、`VT_1x` 等通信变量；
2. 读取到的数据被**缓存在 HMI 内存**中，形成“寄存器镜像”；
3. 脚本调用 `get_bit(VT_1x, 0x1000)` 时，**直接返回本地缓存值，不产生新的串口帧**；
4. 因此，**脚本层的** `get_xxx()` **是零通信开销的。**

> ✅ **优势**：避免脚本逻辑导致通信风暴，保证系统实时性与稳定性

🔄 **数据同步机制 -- 主机模式，以Modbus RTU为例**

✅ **工程配置**

- 在 HMI 工程中，当用户为画面控件、告警条件或资料采样项**绑定了通信变量**（如 `VT_0x:100`）时，系统会**自动生成后台轮询任务**；
- 后续调用 `get_bit(VT_0x, 100)` 即可获得最新值；

> 📌 示例：
> 若画面中有一个位状态指示灯绑定到 `VT_0x:100`，则 HMI 会自动将地址 `100` 加入轮询列表，并周期性更新其缓存值。

✅ **脚本主动触发的按需读取（`start_read`）**

- 当脚本逻辑需要访问**未被工程引用的变量**时，可调用 `start_read(vtype, addr, count)` **显式请求数据同步**；
- 调用后，HMI 通信任务会将指定地址范围的数据从设备读入本地缓存；
- 后续调用 `get_bit()` 即可获得最新值；

> 📌 示例：
> start_read(VT_0x, 100, 1)，则 HMI 会自动将地址 `100` 加入轮询列表，并周期性更新其缓存值。

💡 **示例**

```
local state = get_bit(VT_0x, 0x0000) --获取线圈0x0000地址的状态
```

2.6.set_uint16(vtype, addr, value)

16位无符号整型寄存器写入函数，`set_uint16(vtype, addr, value)` 是 HMI 系统提供的**16 位无符号整数写入函数**，用于向指定**地址**写入一个 **0～65535** 范围的数值。

📊 **参数说明**

参数	类型	说明
`vtype`	number	**变量类型**
`addr`	number	**寄存器地址**
`value`	number	**写入值**：取值范围 0 ~ 65535（超出将自动截断为 `value & 0xFFFF`）

✅ **核心用途**：实现对 Modbus 保持寄存器、FX3U D 寄存器、HMI 内部变量等 16 位数据区的写入。

📣 **通信触发机制（关键行为）**

> **调用 `set_uint16()` 并不总是立即发送串口报文。**其通信行为受以下两个条件共同约束：

✅ **条件 1 HMI 处于主机模式（Master Mode）**

- 仅当 HMI 配置为 **Modbus 主站 、DCBUS或XGUS**，如配置为**Modbus RTU Master**, 才会**主动发送串口报文**；
- 若为从机模式**Slave**，此API仅更新本地镜像，**不会发送任何报文。**

✅ **条件 2：串口通知未被禁止** `set_notify`

- 系统提供 `set_notify(enable)` 接口用于**临时抑制通信输出**
- **仅当通知使能（`set_notify(1)` 或默认状态）时**，此API才会**主动发送串口报文**；
- 若通知被禁用（`set_notify(0)`），函数仅更新本地缓存

💡 **示例**

```
set_uint16(VT_LW, 0x1000, 12345) --设置屏幕内部寄存器0x1000地址为12345
```

2.7.get_uint16(vtype, addr)

16位无符号整型寄存器读取函数，`get_uint16(vtype, addr)` 是 HMI 系统提供的**16 位无符号整数读取函数**，用于向指定**地址**获取一个 **0～65535** 范围的数值。

📊 **参数说明**

参数	类型	说明
`vtype`	number	**变量类型：**如 `VT_4x`（Modbus RTU 保持寄存器）、`VT_D`（FX3U D寄存器）、`VT_LW`（内部变量）等
`addr`	number	**寄存器地址**
返回值	number	0 ~ 65535 的整数值（若通信失败，可能返回 0 或旧值）

> 🔑 `get_uint16(vtype, addr)` **是从 HMI 的本地缓存（内存镜像）中读取数据，而非实时向 PLC/设备发起串口请求。**

🔴 **工作机制：**

1. **HMI 后台任务** 按照工程中配置的**轮询周期**，自动从 PLC 读取 `VT_3x`、`VT_4x` 等通信变量；
2. 读取到的数据被**缓存在 HMI 内存**中，形成“寄存器镜像”；
3. 脚本调用 `get_uint16(VT_3x, 0x1000)` 时，**直接返回本地缓存值，不产生新的串口帧**；
4. 因此，**脚本层的** `get_xxx()` **是零通信开销的。**

> ✅ **优势**：避免脚本逻辑导致通信风暴，保证系统实时性与稳定性

🔄 **数据同步机制 -- 主机模式，以Modbus RTU为例**

✅ **工程配置**

- 在 HMI 工程中，当用户为画面控件、告警条件或资料采样项**绑定了通信变量**（如Modbus RTU Master:`VT_4x:100`）时，系统会**自动生成后台轮询任务**；
- 后续调用 `get_uint16(`VT_4x，100)` 即可获得最新值；

> 📌 示例： 若画面中有一个数值控件绑定到 `VT_0x:100`，则 HMI 会自动将地址 `100` 加入轮询列表，并周期性更新其缓存值。

✅ **脚本主动触发的按需读取（`start_read`）**

- 当脚本逻辑需要访问**未被工程引用的变量**时，可调用 `start_read(vtype, addr, count)` **显式请求数据同步**；
- 调用后，HMI 通信任务会将指定地址范围的数据从设备读入本地缓存；
- 后续调用 `get_uint16(VT_4x, 100)` 即可获得最新值；

> 📌 示例：
> start_read(VT_4x, 100, 1)，则 HMI 会自动将地址 `100` 加入轮询列表，并周期性更新其缓存值。

💡 示例

```
local val = get_uint16 (VT_LW, 0x1000) --获取屏幕内部寄存器0x1000地址的
```

2.8.set_int16(vtype, addr, value)

16位有符号整型寄存器写入函数，`set_int16(vtype, addr, value)` 是 HMI 系统提供的**16 位有符号整数写入函数**，用于向指定**地址**写入一个**-32768 ~ +32767** 范围的整数值

📊 参数说明

参数	类型	说明
`vtype`	number	变量类型：**如** `VT_4x` **（Modbus RTU 保持寄存器）、** `VT_D` **（FX3U D寄存器）、** `VT_LW` **（内部变量）等**
`addr`	number	**寄存器地址**
`value`	number	**写入值**：取值范围**-32768 ~ +32767**（超出将自动截断为 `value & 0xFFFF`）

✅ **核心用途**：实现对支持有符号数的 Modbus 寄存器、PLC 数据区或 HMI 内部变量的**带符号整型写入**

📡 通信触发机制（关键行为）

> **调用** `set_int16()` **并不总是立即发送串口报文。**其通信行为受以下两个条件共同约束：

✅ **条件 1 HMI 处于主机模式（Master Mode）**

- 仅当 HMI 配置为 **Modbus 主站 、DCBUS或XGUS**，如配置为**Modbus RTU Master**, 才会**主动发送串口报文**；
- 若为从机模式**Slave**，此API仅更新本地镜像，**不会发送任何报文。**

✅ **条件 2：串口通知未被禁止** `set_notify`

- 系统提供 `set_notify(enable)` 接口用于**临时抑制通信输出**
- **仅当通知使能（** `set_notify(1)` **或默认状态）时**，此API才会**主动发送串口报文**；
- 若通知被禁用（`set_notify(0)`），函数仅更新本地缓存

💡 示例

```
set_int16(VT_LW, 0x1000, 12345) --设置屏幕内部寄存器0x1000地址为-12345
```

2.9.get_int16(vtype, addr)

16位有符号整型寄存器读取函数，`get_int16(vtype, addr)`是 HMI 系统提供的**16 位有符号整数读取函数**，用于从指定**地址**获取一个 **-32768 ~ +32767** 范围的数值。

📊 **参数说明**

参数	类型	说明
`vtype`	number	**变量类型：**如 `VT_4x`（Modbus RTU 保持寄存器）、`VT_D`（FX3U D寄存器）、`VT_LW`（内部变量）等
`addr`	number	**寄存器地址**
返回值	number	**-32768 ~ +32767** 的整数值（若通信失败，可能返回 0 或旧值）

> 🔑 `get_int16(vtype, addr)` **是从 HMI 的本地缓存（内存镜像）中读取数据，而非实时向 PLC/设备发起串口请求。**

🔴 **工作机制：**

1. **HMI 后台任务** 按照工程中配置的**轮询周期**，自动从 PLC 读取 `VT_3x`、`VT_4x` 等通信变量；
2. 读取到的数据被**缓存在 HMI 内存**中，形成“寄存器镜像”；
3. 脚本调用 `get_int16(VT_3x, 0x1000)` 时，**直接返回本地缓存值**，**不产生新的串口帧**；
4. 因此，**脚本层的** `get_xxx()` **是零通信开销的。**

> ✅ **优势**：避免脚本逻辑导致通信风暴，保证系统实时性与稳定

🔄 **数据同步机制 -- 主机模式，以Modbus RTU为例**

✅ **工程配置**

- 在 HMI 工程中，当用户为画面控件、告警条件或资料采样项**绑定了通信变量**（如Modbus RTU Master:`VT_4x:100`）时，系统会**自动生成后台轮询任务**；
- 后续调用 `get_int16(`VT_4x，100)` 即可获得最新值；

> 📌 示例： 若画面中有一个数值控件绑定到 `VT_4x:100`，则 HMI 会自动将地址 `100` 加入轮询列表，并周期性更新其缓存值。

✅**脚本主动触发的按需读取（`start_read`）**

- 当脚本逻辑需要访问**未被工程引用的变量**时，可调用 `start_read(vtype, addr, count)` **显式请求数据同步**；
- 调用后，HMI 通信任务会将指定地址范围的数据从设备读入本地缓存；
- 后续调用 `get_int16(VT_4x, 100)` 即可获得最新值；

> 📌 示例：
> start_read(VT_4x, 100，1)，则 HMI 会自动将地址 `100` 加入轮询列表，并周期性更新其缓存值。

💡 **示例**

```
local val = get_int16 (VT_LW, 0x1000)--获取屏幕内部寄存器0x1000地址的值
```

2.10.set_uint32(vtype, addr, value)

32位无符号整型寄存器写入函数，`set_uint32(vtype, addr, value)` 是 HMI 系统提供的**32 位无符号整数写入函数**，用于向指定**地址**写入一个 **0 ~ 4,294,967,295** 范围内的整数值。

📊 **参数说明**

参数	类型	说明
`vtype`	number	**变量类型：**如 `VT_4x`（Modbus RTU 保持寄存器）、`VT_D`（FX3U D寄存器）、`VT_LW`（内部变量）等
`addr`	number	**寄存器地址**
`value`	number	**写入值:**取值范围 0 ~ 4294967295（超出将自动截断为 `value & 0xFFFFFFFF`）

✅ **核心用途**：实现对 Modbus 保持寄存器、FX3U D 寄存器等区域的**32 位无符号整型数据写入。**

📡 通信触发机制（关键行为）

> **调用 `set_uint32()` 并不总是立即发送串口报文。**其通信行为受以下两个条件共同约束：

✅ **条件 1 HMI 处于主机模式（Master Mode）**

- 仅当 HMI 配置为 **Modbus 主站 、DCBUS或XGUS**，如配置为**Modbus RTU Master**, 才会**主动发送串口报文**；
- 若为从机模式**Slave**，此API仅更新本地镜像，**不会发送任何报文。**

✅ **条件 2：串口通知未被禁止 `set_notify`**

- 系统提供 `set_notify(enable)` 接口用于**临时抑制通信输出**
- **仅当通知使能（`set_notify(1)` 或默认状态）时**，此API才会**主动发送串口报文**；
- 若通知被禁用（`set_notify(0)`），函数仅更新本地缓存

💡 示例

```
set_uint32(VT_LW, 0x1000, 123456) --设置屏幕内部寄存器0x1000地址为123456
```

2.11.get_uint32(vtype, addr)

32位无符号整型寄存器读取函数，`get_uint32(vtype, addr)` 是HMI 系统提供的**32 位无符号整数读取函数**，用于从指定地址读取一个 **0 ~ 4,294,967,295** 范围的整数值

📊 **参数说明**

参数	类型	说明
`vtype`	number	**变量类型:**如 `VT_3x`（输入寄存器）、`VT_4x`（保持寄存器）、`VT_LW`（内部变量）等
`addr`	number	**寄存器地址**
返回值	number	0 ~ 4294967295 的 32 位无符号整数（若通信失败，可能返回 0 或旧值）

> 🔑 `get_uint32()` **仅从 HMI 内存缓存中读取数据，不会向 PLC/设备发送任何串口指令。**

🔴 **工作机制：**

1. **HMI 后台任务** 按照工程中配置的**轮询周期**，自动从 PLC 读取 `VT_3x`、`VT_4x` 等通信变量；
2. 读取到的数据被**缓存在 HMI 内存**中，形成"寄存器镜像"；
3. 脚本调用 `get_uint32(VT_3x, 0x1000)` 时，**直接返回本地缓存值，不产生新的串口帧**；
4. 因此，**脚本层的** `get_xxx()` **是零通信开销的。**

> ✅ **优势**：避免脚本逻辑导致通信风暴，保证系统实时性与稳定性

🔄 **数据同步机制 -- 主机模式，以Modbus RTU为例**

✅ **工程配置**

- 在 HMI 工程中，当用户为画面控件、告警条件或资料采样项**绑定了通信变量**（如Modbus RTU Master:`VT_4x:100`）时，系统会**自动生成后台轮询任务**；
- 后续调用 `get_uint32(`VT_4x，100)` 即可获得最新值；

> 📌 示例： 若画面中有一个数值控件绑定到 `VT_4x:100`，则 HMI 会自动将地址 `100` 加入轮询列表，并周期性更新其缓存值。

✅ **脚本主动触发的按需读取（`start_read`）**

- 当脚本逻辑需要访问**未被工程引用的变量**时，可调用 `start_read(vtype, addr, count)` **显式请求数据同步**；
- 调用后，HMI 通信任务会将指定地址范围的数据从设备读入本地缓存；
- 后续调用 `get_uint32(VT_4x, 100)` 即可获得最新值；

> 📌 示例：
> start_read(VT_4x, 100，2)，则 HMI 会自动将地址 `100` 加入轮询列表，并周期性更新其缓存值。

💡 **示例**

```
local val = get_uint32 (VT_LW, 0x1000)--获取屏幕内部寄存器0x1000地址的值
```

2.12.set_int32(vtype, addr, value)

32位有符号整型寄存器写入函数，`set_int32(vtype, addr, value)` 是 HMI 系统提供的**32 位有符号整数写入函数**，用于向指定地址的**双字地址**写入一个 ** **-2,147,483,648 ~ +2,147,483,647** ** 范围内的整数值。

📊 参数说明

参数	类型	说明
`vtype`	number	**变量类型:**如 `VT_4x`（保持寄存器）、`VT_D`（PLC 数据寄存器）、`VT_LW`（内部变量）等
`addr`	number	**寄存器地址**
`value`	number	写入值，取值范围 **-2,147,483,648 ~ +2,147,483,647**（超出将自动截断为 `value & 0xFFFFFFFF`）

✅ **核心用途**：实现对 Modbus 保持寄存器、FX3U D 寄存器等区域的**32 位有符号整型数据写入。**

📡 通信触发机制（关键行为）

> **调用 `set_int32()` 并不总是立即发送串口报文。**其通信行为受以下两个条件共同约束：

✅ **条件 1 HMI 处于主机模式（Master Mode）**

- 仅当 HMI 配置为 **Modbus 主站 、DCBUS或XGUS**，如配置为**Modbus RTU Master**, 才会**主动发送串口报文**；
- 若为从机模式**Slave**，此API仅更新本地镜像，**不会发送任何报文。**

✅ **条件 2：串口通知未被禁止 `set_notify`**

- 系统提供 `set_notify(enable)` 接口用于**临时抑制通信输出**
- **仅当通知使能（`set_notify(1)` 或默认状态）时**，此API才会**主动发送串口报文**；
- 若通知被禁用（`set_notify(0)`），函数仅更新本地缓存

💡 示例

```
set_int32(VT_LW, 0x1000, -123456) --设置屏幕内部寄存器0x1000地址为-123456
```

2.13.get_int32(vtype, addr)

32位有符号整型寄存器读取函数，`get_int32(vtype, addr)` 是HMI 系统提供的**32 位有符号整数读取函数**，用于从指定地址读取一个 **-2,147,483,648 ~ +2,147,483,647** 范围内的整数值

📊 参数说明

参数	类型	说明
`vtype`	number	变量类型，如 `VT_3x`（输入寄存器）、`VT_4x`（保持寄存器）、`VT_LW`（内部变量）等

参数	类型	说明
`addr`	number	寄存器地址
返回值	number	**-2,147,483,648 ~ +2,147,483,647** 的 32 位有符号整数（若通信失败，可能返回 0 或旧值）

> 🔑 `get_int32()` **仅从 HMI 内存缓存中读取数据，不会向 PLC/设备发送任何串口指令。**

🔴 **工作机制：**

1. **HMI 后台任务** 按照工程中配置的**轮询周期**，自动从 PLC 读取 `VT_3x`、`VT_4x` 等通信变量；
2. 读取到的数据被**缓存在 HMI 内存**中，形成“寄存器镜像”；
3. 脚本调用 `get_int32(VT_3x, 0x1000)` 时，**直接返回本地缓存值，不产生新的串口帧**；
4. 因此，**脚本层的 `get_xxx()` 是零通信开销的。**

> ✅ **优势**：避免脚本逻辑导致通信风暴，保证系统实时性与稳定性

🔄 **数据同步机制 -- 主机模式，以Modbus RTU为例**

✅ **工程配置**

- 在 HMI 工程中，当用户为画面控件、告警条件或资料采样项**绑定了通信变量**（如Modbus RTU Master:`VT_4x:100`）时，系统会**自动生成后台轮询任务**；
- 后续调用 `get_`
- `int32(VT_4x，100)` 即可获得最新值；

> 📌 示例： 若画面中有一个数值控件绑定到 `VT_4x:100`，则 HMI 会自动将地址 `100` 加入轮询列表，并周期性更新其缓存值。

✅ **脚本主动触发的按需读取（`start_read`）**

- 当脚本逻辑需要访问**未被工程引用的变量**时，可调用 `start_read(vtype, addr, count)` **显式请求数据同步**；
- 调用后，HMI 通信任务会将指定地址范围的数据从设备读入本地缓存；
- 后续调用 `get_int32(VT_4x，100)` 即可获得最新值；

> 📌 示例：
> start_read(VT_4x, 100，2)，则 HMI 会自动将地址 `100` 加入轮询列表，并周期性更新其缓存值。

💡 **示例**

```
local val = get_int32 (VT_LW, 0x1000)--获取屏幕内部寄存器0x1000地址的值
```

2.14.set_uint64(vtype, addr, value)

64位无符号整型寄存器写入函数，`set_uint64(vtype, addr, value)` 是 HMI 系统提供的**64 位无符号整数写入函数**，用于**地址**写入一个 **0 ~ $2^{64}-1$** 范围内的整数值。

📊 **参数说明**

参数	类型	说明
`vtype`	number	**变量类型*:**如 `VT_4x`（保持寄存器）、`VT_LW`（内部变量）等
`addr`	number	**寄存器地址**
`value`	number	写入值，取值范围**0 ~ $2^{64}-1$**

✅ **核心用途**：实现对支持 64 位数据的设备或 HMI 内部变量的**超大无符号整型写入。**

🔔 通信触发机制（关键行为）

> **调用** `set_uint64()` **并不总是立即发送串口报文。**其通信行为受以下两个条件共同约束：

✅ **条件 1 HMI 处于主机模式（Master Mode）**

- 仅当 HMI 配置为 **Modbus 主站 、DCBUS或XGUS**，如配置为**Modbus RTU Master**, 才会**主动发送串口报文**；
- 若为从机模式**Slave**，此API仅更新本地镜像，**不会发送任何报文。**

✅ **条件 2：串口通知未被禁止** `set_notify`

- 系统提供 `set_notify(enable)` 接口用于**临时抑制通信输出**
- **仅当通知使能（** `set_notify(1)` **或默认状态）时**，此API才会**主动发送串口报文**；
- 若通知被禁用（`set_notify(0)`），函数仅更新本地缓存

💡 示例

```
set_uint64(VT_LW, 0x1000, 1234567890) --设置屏幕内部寄存器0x1000地址为1234567890
```

2.15.get_uint64 (vtype, addr)

64位有符号整型寄存器读取函数，`get_uint64(vtype, addr)`是HMI 系统提供的**64 位有符号整数读取函数**，用于从指定地址读取一个**0 ~ $2^{64}-1$** 范围内的整数值。

📊 参数说明

参数	类型	说明
`vtype`	number	**变量类型:**如 `VT_3x`（输入寄存器）、`VT_4x`（保持寄存器）、`VT_LW`（内部变量）等
`addr`	number	**寄存器地址**
返回值	number	**0 ~ $2^{64}-1$** 的 64 位有符号整数，（若通信失败，可能返回 0 或旧值）

> 🔑 `get_uint64()` **仅从 HMI 内存缓存中读取数据，不会向 PLC/设备发送任何串口指令。**

🔴 **工作机制：**

1. **HMI 后台任务** 按照工程中配置的**轮询周期**，自动从 PLC 读取 `VT_3x`、`VT_4x` 等通信变量；

2. 读取到的数据被**缓存在 HMI 内存**中，形成“寄存器镜像”；
3. 脚本调用 `get_uint64(VT_3x, 0x1000)` 时，**直接返回本地缓存值，不产生新的串口帧**；
4. 因此，**脚本层的 `get_xxx()` 是零通信开销的。**

> ✅ **优势**：避免脚本逻辑导致通信风暴，保证系统实时性与稳定性

🔁 **数据同步机制 -- 主机模式，以Modbus RTU为例**

✅ **工程配置**

- 在 HMI 工程中，当用户为画面控件、告警条件或资料采样项**绑定了通信变量**（如Modbus RTU Master:`VT_4x:100`）时，系统会**自动生成后台轮询任务**；
- 后续调用 `get_uint64(`VT_4x，100)` 即可获得最新值；

> 📌 示例： 若画面中有一个数值控件绑定到 `VT_4x:100`，则 HMI 会自动将地址 `100` 加入轮询列表，并周期性更新其缓存值。

✅ **脚本主动触发的按需读取（`start_read`）**

- 当脚本逻辑需要访问**未被工程引用的变量**时，可调用 `start_read(vtype, addr, count)` **显式请求数据同步**；
- 调用后，HMI 通信任务会将指定地址范围的数据从设备读入本地缓存；
- 后续调用 `get_uint64(VT_4x，100)` 即可获得最新值；

> 📌 示例：
> start_read(VT_4x, 100，4)，则 HMI 会自动将地址 `100` 加入轮询列表，并周期性更新其缓存值。

💡 **示例**

```
local val = get_uint64 (VT_LW, 0x1000)--获取屏幕内部寄存器0x1000地址的值
```

2.16.set_int64(vtype, addr, value)

64位有符号整型寄存器写入函数，`set_int64(vtype, addr, value)` 是 HMI 系统提供的**64 位有符号整数写入函数**，用于**地址**写入一个 **-2^{63} ~ $2^{63}-1$**范围的整数值。

📊 **参数说明**

参数	类型	说明
`vtype`	number	**变量类型：**如 `VT_4x`（保持寄存器）、`VT_LW`（内部变量）等
`addr`	number	**寄存器地址**
`value`	number	写入值，取值范围 **-2^{63} ~ $2^{63}-1$**

✅ **核心用途**：实现对支持 64 位数据的设备或 HMI 内部变量的**超大无符号整型写入。**

🔔 **通信触发机制（关键行为）**

> **调用 `set_int64()` 并不总是立即发送串口报文。**其通信行为受以下两个条件共同约束：

✅ **条件 1 HMI 处于主机模式（Master Mode）**

- 仅当 HMI 配置为 **Modbus 主站 、DCBUS或XGUS**，如配置为**Modbus RTU Master**, 才会**主动发送串口报文**；
- 若为从机模式**Slave**，此API仅更新本地镜像，**不会发送任何报文。**

✅ **条件 2：串口通知未被禁止 `set_notify`**

- 系统提供 `set_notify(enable)` 接口用于**临时抑制通信输出**
- **仅当通知使能（`set_notify(1)` 或默认状态）时**，此API才会**主动发送串口报文**；
- 若通知被禁用（`set_notify(0)`），函数仅更新本地缓存

💡 **示例**

```
set_int64(VT_LW, 0x1000, -1234567890) --设置屏幕内部寄存器0x1000地址为-1234567890
```

2.17.get_int64 (vtype, addr)

64位有符号整型寄存器读取函数，`get_int64(vtype, addr)`是HMI 系统提供的**64 位有符号整数读取函数**，用于从指定地址读取一个 **-2^{63} ~ $2^{63}-1$** 范围内的整数值。

📊 **参数说明**

参数	类型	说明
`vtype`	number	**变量类型:**如 `VT_3x`（输入寄存器）、`VT_4x`（保持寄存器）、`VT_LW`（内部变量）等
`addr`	number	**寄存器地址**
返回值	number	**-2^{63} ~ $2^{63}-1$**的 64 位无符号整数，（若通信失败，可能返回 0 或旧值）

> 🔑 `get_int64()` **仅从 HMI 内存缓存中读取数据，不会向 PLC/设备发送任何串口指令。**

🔴 **工作机制：**

1. **HMI 后台任务** 按照工程中配置的**轮询周期**，自动从 PLC 读取 `VT_3x`、`VT_4x` 等通信变量；
2. 读取到的数据被**缓存在 HMI 内存**中，形成“寄存器镜像”；
3. 脚本调用 `get_int64(VT_3x, 0x1000)` 时，**直接返回本地缓存值，不产生新的串口帧**；
4. 因此，**脚本层的 `get_xxx()` 是零通信开销的。**

> ✅ **优势**：避免脚本逻辑导致通信风暴，保证系统实时性与稳定性

🔁 **数据同步机制 -- 主机模式，以Modbus RTU为例**

✅ **工程配置**

- 在 HMI 工程中，当用户为画面控件、告警条件或资料采样项**绑定了通信变量**（如Modbus RTU Master:`VT_4x:100`）时，系统会**自动生成后台轮询任务**；
- 后续调用 `get_int64(VT_4x，100)` 即可获得最新值；

> 📌 示例： 若画面中有一个数值控件绑定到 `VT_4x:100`，则 HMI 会自动将地址 `100` 加入轮询列表，并周期性更新其缓存值。

✅ **脚本主动触发的按需读取（`start_read`）**

- 当脚本逻辑需要访问**未被工程引用的变量**时，可调用 `start_read(vtype, addr, count)` **显式请求数据同步**；
- 调用后，HMI 通信任务会将指定地址范围的数据从设备读入本地缓存；
- 后续调用 `get_int64(VT_4x, 100)` 即可获得最新值；

> 📌 示例：
> start_read(VT_4x, 100, 4)，则 HMI 会自动将地址 `100` 加入轮询列表，并周期性更新其缓存值。

💡 **示例**

```
local val = get_int64 (VT_LW, 0x1000)--获取屏幕内部寄存器0x1000地址的值
```

2.18.set_float(vtype, addr, value)

单精度浮点数寄存器写入函数，`set_float(vtype, addr, value)` 是 HMI 系统提供的**单精度浮点数（IEEE 754 32 位）写入函数**，用于向指定**地址**写入实数值。

📊 **参数说明**

参数	类型	说明
`vtype`	number	**变量类型：**如 `VT_4x`（保持寄存器）、`VT_D`（PLC 数据寄存器）、`VT_LW`（内部变量）等
`addr`	number	**寄存器地址**
`value`	number	浮点写入值，如 `1.234`、`-45.6`、`3.14159e5`

✅ **核心用途**：实现对 Modbus 保持寄存器、PLC 数据区或 HMI 内部变量的**浮点型参数设置与指令下发。**

📡 **通信触发机制（关键行为）**

> **调用 `set_float()` 并不总是立即发送串口报文。**其通信行为受以下两个条件共同约束：

✅ **条件 1 HMI 处于主机模式（Master Mode）**

- 仅当 HMI 配置为 **Modbus 主站 、DCBUS或XGUS**，如配置为**Modbus RTU Master**, 才会**主动发送串口报文**；
- 若为从机模式**Slave**，此API仅更新本地镜像，**不会发送任何报文。**

✅ **条件 2：串口通知未被禁止 `set_notify`**

- 系统提供 `set_notify(enable)` 接口用于**临时抑制通信输出**
- **仅当通知使能（`set_notify(1)` 或默认状态）时**，此API才会**主动发送串口报文**；
- 若通知被禁用（`set_notify(0)`），函数仅更新本地缓存

💡 **示例**

```
set_float(VT_LW, 0x1000, 1.234)--设置屏幕内部寄存器0x1000地址为1.234
```

2.19.get_float (vtype, addr)

单精度浮点数寄存器读取函数，`get_float(vtype, addr)` 是 HMI 系统提供的**单精度浮点数（IEEE 754 32 位）读取函数**，用于从指定**地址**读取一个带小数的实数值。

📊 **参数说明**

参数	类型	说明
`vtype`	number	**变量类型:**如 `VT_3x`（输入寄存器）、`VT_4x`（保持寄存器）、`VT_LW`（内部变量）等
`addr`	number	**寄存器地址**
返回值	number	IEEE 754 单精度浮点数

> 🔑 `get_float` **仅从 HMI 内存缓存中读取数据，不会向 PLC/设备发送任何串口指令。**

🔴 **工作机制：**

1. **HMI 后台任务** 按照工程中配置的**轮询周期**，自动从 PLC 读取 `VT_3x`、`VT_4x` 等通信变量；
2. 读取到的数据被**缓存在 HMI 内存**中，形成“寄存器镜像”；
3. 脚本调用 `get_float(VT_3x, 0x1000)` 时，**直接返回本地缓存值，不产生新的串口帧**；
4. 因此，**脚本层的** `get_xxx()` **是零通信开销的。**

> ✅ **优势**：避免脚本逻辑导致通信风暴，保证系统实时性与稳定

🔄 **数据同步机制 -- 主机模式，以Modbus RTU为例**

✅ **工程配置**

- 在 HMI 工程中，当用户为画面控件、告警条件或资料采样项**绑定了通信变量**（如Modbus RTU Master:`VT_4x:100`）时，系统会**自动生成后台轮询任务**；
- 后续调用 `get_float(`VT_4x，100)` 即可获得最新值；

> 📌 示例： 若画面中有一个数值控件绑定到 `VT_4x:100`，则 HMI 会自动将地址 `100` 加入轮询列表，并周期性更新其缓存值。

✅ **脚本主动触发的按需读取（`start_read`）**

- 当脚本逻辑需要访问**未被工程引用的变量**时，可调用 `start_read(vtype, addr, count)` **显式请求数据同步**；
- 调用后，HMI 通信任务会将指定地址范围的数据从设备读入本地缓存；
- 后续调用 `get_float(VT_4x, 100)` 即可获得最新值；

> 📌 示例：
> start_read(VT_4x, 100，4)，则 HMI 会自动将地址 `100` 加入轮询列表，并周期性更新其缓存值。

💡 **示例**

```
local val = get_float(VT_LW, 0x1000)--获取屏幕内部寄存器0x1000地址的值
```

2.20.set_double(vtype, addr, value)

双精度浮点数寄存器写入函数，`set_double(vtype, addr, value)` 是 HMI 系统提供的**双精度浮点数（IEEE 754 64 位）写入函数**，用于向指定**地址**写入一个高精度实数值。

📊 **参数说明**

参数	类型	说明
`vtype`	number	**变量类型:**如 `VT_LW`（内部变量）、`VT_RW`（掉电保存）等
`addr`	number	**寄存器地址**
`value`	number	双精度浮点值，如 `3.141592653589793`、`-1.23e-10`

✅ **核心用途**：实现对 Modbus 保持寄存器、PLC 数据区或 HMI 内部变量的**双精度型参数设置与指令下发。**

📣 **通信触发机制（关键行为）**

> **调用 `set_double()` 并不总是立即发送串口报文。**其通信行为受以下两个条件共同约束：

✅ **条件 1 HMI 处于主机模式（Master Mode）**

- 仅当 HMI 配置为 **Modbus 主站 、DCBUS或XGUS**，如配置为**Modbus RTU Master**, 才会**主动发送串口报文**；
- 若为从机模式**Slave**，此API仅更新本地镜像，**不会发送任何报文。**

✅ **条件 2：串口通知未被禁止 `set_notify`**

- 系统提供 `set_notify(enable)` 接口用于**临时抑制通信输出**
- **仅当通知使能（`set_notify(1)` 或默认状态）时**，此API才会**主动发送串口报文**；
- 若通知被禁用（`set_notify(0)`），函数仅更新本地缓存

💡 **示例**

```
set_double(VT_LW, 0x1000,1.23456789)--设置屏幕内部寄存器0x1000地址为1.23456789
```

2.21.get_double (vtype, addr)

双精度浮点数寄存器读取函数，`get_double(vtype, addr)` 是 HMI 系统提供的**双精度浮点数（IEEE 754 64 位）读取函数**，用于从指定**地址**读取一个高精度实数值。

📊 **参数说明**

参数	类型	说明
`vtype`	number	**变量类型:**如 `VT_LW`（内部变量）、`VT_RW`（掉电保存）等
`addr`	number	**寄存器地址**
返回值	number	IEEE 754 双精度浮点数（Lua 原生 number 类型）

> 🔑 `get_double` **仅从 HMI 内存缓存中读取数据，不会向 PLC/设备发送任何串口指令。**

🔴 **工作机制：**

1. **HMI 后台任务** 按照工程中配置的**轮询周期**，自动从 PLC 读取 `VT_3x`、`VT_4x` 等通信变量；
2. 读取到的数据被**缓存在 HMI 内存**中，形成“寄存器镜像”；
3. 脚本调用 `get_double (VT_3x, 0x1000)` 时，**直接返回本地缓存值，不产生新的串口帧**；
4. 因此，**脚本层的** `get_xxx()` **是零通信开销的。**

> ✅ **优势**：避免脚本逻辑导致通信风暴，保证系统实时性与稳定性

🔄 **数据同步机制 -- 主机模式，以Modbus RTU为例**

✅ **工程配置**

- 在 HMI 工程中，当用户为画面控件、告警条件或资料采样项**绑定了通信变量**（如Modbus RTU Master:`VT_4x:100`）时，系统会**自动生成后台轮询任务**；
- 后续调用 `get_double(`VT_4x，100)` 即可获得最新值；

> 📌 示例： 若画面中有一个数值控件绑定到 `VT_4x:100`，则 HMI 会自动将地址 `100` 加入轮询列表，并周期性更新其缓存值。

✅ **脚本主动触发的按需读取（`start_read`）**

- 当脚本逻辑需要访问**未被工程引用的变量**时，可调用 `start_read(vtype, addr, count)` **显式请求数据同步**；
- 调用后，HMI 通信任务会将指定地址范围的数据从设备读入本地缓存；
- 后续调用 `get_double(VT_4x, 100)` 即可获得最新值；

> 📌 示例：
> start_read(VT_4x, 100，8)，则 HMI 会自动将地址 `100` 加入轮询列表，并周期性更新其缓存值。

💡 **示例**

```
local val = get_double(VT_LW, 0x1000)--获取屏幕内部寄存器0x1000地址的值
```

2.22.set_string(vtype, addr, strings)

字符串寄存器写入函数，`set_string(vtype, addr, str)` 是 HMI 系统提供的**字符串写入函数**，用于向指定**地址序列**写入一个**'UTF-8' 或'GBK' 编码**的字符串。

📊 **参数说明**

参数	类型	说明
`vtype`	number	**变量类型：**如 `VT_LW`（内部变量）、`VT_4x`（保持寄存器）等
`addr`	number	**寄存器地址**
`str`	string	要写入的字符串，如 `"https://www.gz-dc.com/"`，最大2K字节

✅ **核心用途**：实现对 HMI 内部变量或支持字符串的设备寄存器的**文本数据写入。**

🔔 通信触发机制（关键行为）

> **调用 `set_string()` 并不总是立即发送串口报文。**其通信行为受以下两个条件共同约束：

✅ 条件 1 HMI 处于主机模式（Master Mode）

- 仅当 HMI 配置为 **Modbus 主站 、DCBUS或XGUS**，如配置为**Modbus RTU Master**，才会**主动发送串口报文**；
- 若为从机模式**Slave**，此API仅更新本地镜像，**不会发送任何报文。**

✅ 条件 2：串口通知未被禁止`set_notify`

- 系统提供 `set_notify(enable)` 接口用于**临时抑制通信输出**
- **仅当通知使能（`set_notify(1)` 或默认状态）时**，此API才会**主动发送串口报文**；
- 若通知被禁用（`set_notify(0)`），函数仅更新本地缓存

💡 示例

```
set_string(VT_LW, 0x1000, 'https://www.gz-dc.com/')--设置屏幕内部寄存器0x1000地址为https://www.gz-dc.com/
```

2.23.get_string(vtype, addr，len)

字符串变量读取函数，`get_string(vtype, addr, len)` 是 HMI 系统提供的**字符串读取函数**，用于从指定存储区的**地址**中读取一段以空字符 `\0` 结尾的文本数据

📊 参数说明

参数	类型	说明
`vtype`	number	**变量类型：**如 `VT_LW`（内部变量）、`VT_RW`（掉电保存）等
`addr`	number	**寄存器地址**
`len`	number (可选)	**最大读取字节数:**默认 128 字节，最大支持 2048 字节（2KB）
返回值	string	得到的字符串（自动在 `\0` 处截断）

> 🔑 `get_string` **仅从 HMI 内存缓存中读取数据，不会向 PLC/设备发送任何串口指令。**

🔴 工作机制：

1. **HMI 后台任务** 按照工程中配置的**轮询周期**，自动从 PLC 读取 `VT_3x`、`VT_4x` 等通信变量；
2. 读取到的数据被**缓存在 HMI 内存**中，形成"寄存器镜像"；
3. 脚本调用 `get_string(VT_3x, 0x1000)` 时，**直接返回本地缓存值**，**不产生新的串口帧**；
4. 因此，**脚本层的 `get_xxx()` 是零通信开销的。**

> ✅ **优势**：避免脚本逻辑导致通信风暴，保证系统实时性与稳定性

🔄 数据同步机制 -- 主机模式，以Modbus RTU为例

✅ 工程配置

- 在 HMI 工程中，当用户为画面控件、告警条件或资料采样项**绑定了通信变量**（如Modbus RTU Master:`VT_4x:100`）时，系统会**自动生成后台轮询任务**；
- 后续调用 `get_string(`VT_4x，100)` 即可获得最新值；

> 📌 示例： 若画面中有一个文本控件绑定到 `VT_4x:100`，则 HMI 会自动将地址 `100` 加入轮询列表，并周期性更新其缓存值。

✅ 脚本主动触发的按需读取（`start_read`）

- 当脚本逻辑需要访问**未被工程引用的变量**时，可调用 `start_read(vtype, addr, count)` **显式请求数据同步**；
- 调用后，HMI 通信任务会将指定地址范围的数据从设备读入本地缓存；
- 后续调用 `get_string(VT_4x，100)` 即可获得最新值；

> 📌 示例：
> start_read(VT_4x, 100，128)，则 HMI 会自动将地址 `100` 加入轮询列表，并周期性更新其缓存值。

💡 示例

```
local string = get_string (VT_LW, 0x1000)--获取屏幕内部寄存器0x1000地址的值
```

2.24.set_uint16_ex(vtype, addr, value1,value2, ..., value120)

批量无符号16位寄存器写入函数,`set_uint16_ex(vtype, addr, value1, value2, ..., valueN)` 是 HMI 系统提供的**批量 16 位无符号整数写入函数**，用于**一次性向连续多个寄存器**（最多 120 个）。

📊 参数说明

参数	类型	说明
`vtype`	number	**变量类型:**如 `VT_4x`（保持寄存器）、`VT_D`（PLC 数据寄存器）、`VT_LW`（内部变量）等
`addr`	number	**寄存器地址**
`value1 ~ valueN`	number	1 到 120 个无符号 16 位整数值，每个取值范围 0 ~ 65535

参数	类型	说明
最大数量	—	120 个寄存器

✅ **核心用途**：高效批量设置设备参数。

📣 通信触发机制（关键行为）

> **调用 `set_uint16_ex()` 并不总是立即发送串口报文。**其通信行为受以下两个条件共同约束：

✅ **条件 1 HMI 处于主机模式（Master Mode）**

- 仅当 HMI 配置为 **Modbus 主站 、DCBUS或XGUS**，如配置为**Modbus RTU Master**, 才会**主动发送串口报文**；
- 若为从机模式**Slave**，此API仅更新本地镜像，**不会发送任何报文。**

✅ **条件 2：串口通知未被禁止 `set_notify`**

- 系统提供 `set_notify(enable)` 接口用于**临时抑制通信输出**
- **仅当通知使能（`set_notify(1)` 或默认状态）时**，此API才会**主动发送串口报文**；
- 若通知被禁用（`set_notify(0)`），函数仅更新本地缓存

💡 示例

```
local r_addr = 0x6060
set_uint16_ex(vtype, r_addr,
        1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,
        21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,
        41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,
        61,62,63,64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80,
        81,82,83,84,85,86,87,88,89,90,91,92,93,94,95,96,97,98,99,100,
 101,102,103,104,105,106,107,108,109,110,111,112,113,114,115,116,117,118,119,120
)
```

2.25.set_array(vtype, addr, buff)

批量寄存器写入函数，`set_array(vtype, addr, buff)` 是 HMI 系统提供的**高效批量寄存器写入函数**，用于将一个 **16 位字（word）数组**一次性写入指定起始地址的连续寄存器区域。

与 `set_uint16_ex` 不同，`set_array` 接收一个 **Lua 表（table）作为缓冲区**，而非可变参数列表，更适合程序化生成和动态数据处理。

📊 参数说明

参数	类型	说明
`vtype`	number	**变量类型:**如 `VT_4x`（保持寄存器）、`VT_D`、`VT_LW` 等
`addr`	number	**寄存器地址**
`buff`	table	**字（word）数组:**每个元素为 0 ~ 65535 的整数

参数	类型	说明
最大长度	—	120 个字（即最多写入 120 个连续 16 位寄存器）

✅ **核心用途**：高效写入结构化数据块、通信协议帧、设备配置数组、HMI 内部状态缓存等。

📣 通信触发机制（关键行为）

> **调用 `set_array()` 并不总是立即发送串口报文。**其通信行为受以下两个条件共同约束：

✅ **条件 1 HMI 处于主机模式（Master Mode）**

- 仅当 HMI 配置为 **Modbus 主站 、DCBUS或XGUS**，如配置为**Modbus RTU Master**, 才会**主动发送串口报文**；
- 若为从机模式**Slave**，此API仅更新本地镜像，**不会发送任何报文。**

✅ **条件 2：串口通知未被禁止 `set_notify`**

- 系统提供 `set_notify(enable)` 接口用于**临时抑制通信输出**
- **仅当通知使能（`set_notify(1)` 或默认状态）时**，此API才会**主动发送串口报文**；
- 若通知被禁用（`set_notify(0)`），函数仅更新本地缓存

💡 示例

```
local buff = {}
local r_addr = 0x6060
local index = 120
local data  = 0
for i = 1, 120
do
    buff[i] = i
end
set_array(vtype, r_addr, buff)
```

2.26.start_read(index,vtype, addr,quantity，cycle，clcye_run，mode)

主机模式下**周期性自动读取寄存器**的通信控制函数，`start_read` 是 HMI 在 **主机模式**（如 Modbus Master、FX3U 主站）下的**后台自动轮询机制**，用于**周期性向从机设备发起读取请求。**

📊 参数说明

参数	类型	必填	说明
`index`	number	✅	**任务索引号:**（**0～127**），用于后续 `stop_read(index)` 停止该任务
`vtype`	number	✅	**数据类型:**如 `VT_3x`，`VT_4x`
`addr`	number	✅	**寄存器地址**

参数	类型	必填	说明
`quantity`	number	✅	**读取数量:**1 ~ 120 个寄存器
`cycle`	number	❌	**选填**，轮询周期倍数（默认 `0` → 每周期都读）
`cycle_run`	number	❌	**选填**，在周期内的第几次执行（0-based，需 `< cycle`）
`mode`	number	❌	**选填**，读取模式： `0` = 持续周期读取（默认） `1` = 仅读一次（需先调用 `create_resp_que()`）

⚠️ **注意**:

- 该函数**无返回值**;
- 实际数据需通过 `get_uint16`、`get_float` 等函数从**本地缓存**读取;
- **仅在主机模式下有效**（如 Modbus RTU/FX3U）。

💡 **示例**

以Modbus RTU Master模式为例

2.26.1自动后台轮询

- 调用 `start_read` 后，HMI 系统将该读取任务加入**通信调度队列**;
- 每个通信周期，系统自动发送 Modbus 请求;
- 返回数据自动写入 HMI 内存镜像区;
- 脚本通过 `get_xx(VT_4x, addr)` **直接读缓存**，**无通信延迟。**

2.26.2. `cycle` 与 `cycle_run` 的调度逻辑

HMI 将通信周期划分为多个“子周期”，用于**区分高/低频任务**:

调用示例	行为说明
`start_read(0, VT_4x, 0x1000, 10)`	每个通信周期都读 `4x1000～1009`
`start_read(1, VT_4x, 0x2000, 10, 3, 2)`	**每 3 个周期读一次**，且在第 **3 个周期**（即 `cycle_run=2`，0-based）执行

🔄 **调度时序示例:**

```
1周期 #0 (0ms)   → 读 0x1000（高频）
2周期 #1 (50ms)  → 读 0x1000
3周期 #2 (100ms) → 读 0x1000 + 读 0x2000（低频任务触发）
4周期 #3 (150ms) → 读 0x1000
5周期 #4 (200ms) → 读 0x1000
6周期 #5 (250ms) → 读 0x1000 + 读 0x2000
7...
```

> 💡 **设计目的**:

> - 高频变量（如速度、温度）→ `cycle=0`（每周期读）
> - 低频变量（如累计量、状态字）→ `cycle=5, cycle_run=4`（每 5 周期读一次）→ **降低总线负载，提升系统实时性**

2.27.stop_read(index)

停止后台周期性读取任务，`stop_read(index)` 是 HMI 系统在**主机模式**（如 Modbus Master、FX3U 主站）下提供的**通信任务管理函数**，用于**停止由** `start_read` **启动的后台周期性读取任务。**调用后，HMI 将**立即从通信调度队列中移除该任务**，不再向从机设备发送对应的读取指令，但**本地缓存中的数据保持不变**（保留最后一次成功读取的值）。

📊 **参数说明**

参数	类型	说明
`index`	number	**任务索引号（**0 ~ 127**）:**必须与 `start_read` 中注册的 `index` 一致

✅ **核心用途**：动态控制通信负载、释放无效轮询、实现按需读取策略。

2.28.stop_all_read()

停止所有脚本启动的后台读取任务，`stop_all_read()` 是 HMI 系统在**主机模式**（如 Modbus RTU/TCP Master、FX3U 主站等）下提供的**全局通信任务停止函数**，用于**一次性停止所有由** `start_read` **启动的后台周期性读取任务**

✅ **核心用途**：

- 快速释放所有脚本注册的通信资源；
- 适用于系统复位、模式切换、紧急停机等场景；
- 避免逐个调用 `stop_read(index)` 的繁琐操作。

2.29.set_auto_read(en)

控制"画面绑定变量"自动读取的全局开关，`set_auto_read(en)` 是 HMI 系统在**主机模式**（如 Modbus Master、FX3U 主站）下提供的**全局通信策略控制函数**，用于**启用或禁用画面控件所绑定寄存器的自动轮询机制。**

📊 **参数说明**

参数	类型	说明
`en`	number	**使能标志：** `1` = 启用画面绑定变量的自动读取（默认） `0` = 禁止自动读取，仅依赖 `start_read` 等脚本控制

✅ **核心用途**：

- **精细控制通信行为**，将"画面自动读取"与"脚本主动读取"解耦；
- 在高性能或低带宽场景下，**完全由脚本通过** `start_read` **接管数据更新**；
- 避免系统自动生成冗余或冲突的读取指令。

2.30.create_resp_que()

建响应队列以支持单次读取模式，`create_resp_que()` 是 HMI 系统在**主机模式**（如 Modbus Master、FX3U 主站）下提供的**通信响应管理函数**，用于**创建一个临时响应队列**，使得后续调用 `start_read(..., mode=1)` 能够实现**仅发送一次读取指令**（而非周期性轮询）

核心用途：

- 实现**按需单次读取**（如设备信息查询、参数加载、诊断请求）；
- 避免为一次性操作启动长期后台任务；
- 与 `on_cmd_resp()` 配合，可检测是否收到从机响应。

💡 示例

```
ENCRYPT_=0     --LUA脚本加密

--数据类型定义
VT_LW = 1     --变量地址
VT_RW = 2     --FLASH存储
VT_0x = 10    --线圈
VT_1x = 11    --输入点
VT_3x = 12    --输入寄存器
VT_4x = 13    --保持寄存器

function on_init()
    set_auto_read(0) --脚本控制读
    create_resp_que() --创建队列,可以只读1次存器指令
    start_read(1, VT_4x, 0xA009, 30, 0, 0, 1) --只发一次读A009 ~ A01D 范围的数据
end
```

2.31.on_cmd_resp(slave,vtype,addr,count,ret,wr)

主机模式通信指令回调函数，`on_cmd_resp` 是 HMI 系统在**主机模式**（如 Modbus Master、FX3U 主站）下提供的**通信结果异步回调函数**，用于**捕获每一次读/写指令的执行结果。**该函数在 HMI **收到从机响应或发生通信超时后自动触发**，是实现**通信状态监控、错误处理、流程控制**的核心机制。

📊 参数说明

参数	类型	说明
`slave`	number	**从机设备索引:**即工程中配置的“从机号”
`vtype`	number	**寄存器类型:**如 `VT_3x=12`，`VT_4x=13`
`addr`	number	**寄存器地址**
`count`	number	**寄存器数量**
`ret`	number	**执行结果:** `0` = 成功 `1` = 异常（超时、CRC 错、从机异常码等）
`wr`	number	**操作类型:** `0` = 读操作 `1` = 写操作

✅ **核心用途**：

- 判断单次读写是否成功；
- 实现请求-响应式业务逻辑（如参数下发确认）；
- 配合 `create_resp_que()` 构建可靠的按需通信流程。

💡 **示例**

```
ENCRYPT_=0     --LUA脚本加密

--数据类型定义
VT_LW = 1     --变量地址
VT_RW = 2     --FLASH存储
VT_0x = 10    --线圈
VT_1x = 11    --输入点
VT_3x = 12    --输入寄存器
VT_4x = 13    --保持寄存器

function on_init()
    set_auto_read(0) --脚本控制读
    create_resp_que() --创建队列,可以只读1次存器指令
    start_read(1, VT_4x, 0xA009, 30, 0, 0, 1) --只发一次读A009 ~ A01D 范围的数据
end

function on_cmd_resp(slave,vtype,addr,count,ret,wr)
    print('on_cmd_resp(vtype = '..vtype..', addr = '..(string.format('%04X',
addr))..', count = '..count..', ret = '..ret..', wr ='..wr..')')
end

> 通讯成功 ：  on_cmd_resp(vtype = 13, addr = A009, count = 30, ret = 0, wr = 0)
>
```

2.32.set_slave_site(idx, slave_id)

动态修改 从机站号，`set_slave_site(idx, slave_id)` 是 HMI 系统在**主机模式**（如 Modbus RTU/TCP Master、FX3U 主站）下提供的**从机设备地址动态配置函数**，用于**在运行时修改已配置从机的站号**（Slave ID）。

✅ **核心用途**：

- 适应**现场设备站号可变**的灵活组网需求。

📊 **参数说明**

参数	类型	说明
`idx`	number	**从机索引（0-based）**:对应工程中"通信设置"里从机设备的顺序编号
`slave_id`	number	**新的从机站号：** 如 Modbus RTU，1 ~ 247

3.绘图函数

3.1.on_draw(screen,control)

`on_draw` 是 HMI 系统提供的**控件级自绘回调接口**，用于在指定画面的特定控件区域上执行**用户自定义的图形绘制。**开发者可`on_draw`函数里调用各类 `draw_xxx` 图形 API，实现**图片、文字、图形(点、线、几何图形)的绘制**

📊 **参数说明**

参数	类型	说明
`screen_id`	number	**当前画面 ID** • 表示触发重绘的界面编号 • 可用于区分不同画面的绘制逻辑
`control_id`	number	**触发重绘的控件 ID** • 表示需要自绘的控件唯一标识 • 必须 ≠ 0

⚠️ **注意**:

1. 该函数为**系统回调函数**，用户**不得直接调用**;
2. 所有 `draw_xxx` 图形绘制指令（如 `draw_line`、`draw_surface` 等）**必须在此函数内调用**才能生效;
3. 控件**ID ≠ 0** ，否则自绘功能无效。

3.1.1.触发条件 on_draw

- 界面包含动画、视频、RTC 时间等动态元素刷新;
- 用户触摸或操作屏幕控件;
- 脚本通过 `set_xxx` 更新控件属性;
- 串口/LUA 指令修改寄存器，从而修改控件状态;
- 主动调用 `redraw()`

3.1.2. 多图层绘制机制

核心原理

HMI 系统采用**控件 Z 轴顺序**（即界面编辑器中的控件叠放顺序）作为天然的图层管理机制。`on_draw(screen_id, control_id)` 回调函数在**每个可重绘控件独立触发**，开发者可通过判断 `control_id`，将不同的图形内容绘制到**对应控件图层（控件叠放上下关系）**，从而实现逻辑上的**“分层叠加”。**

> ✅ **关键设计**：控件在界面中的**上下排列顺序**直接决定了最终视觉的**图层叠加顺序。**

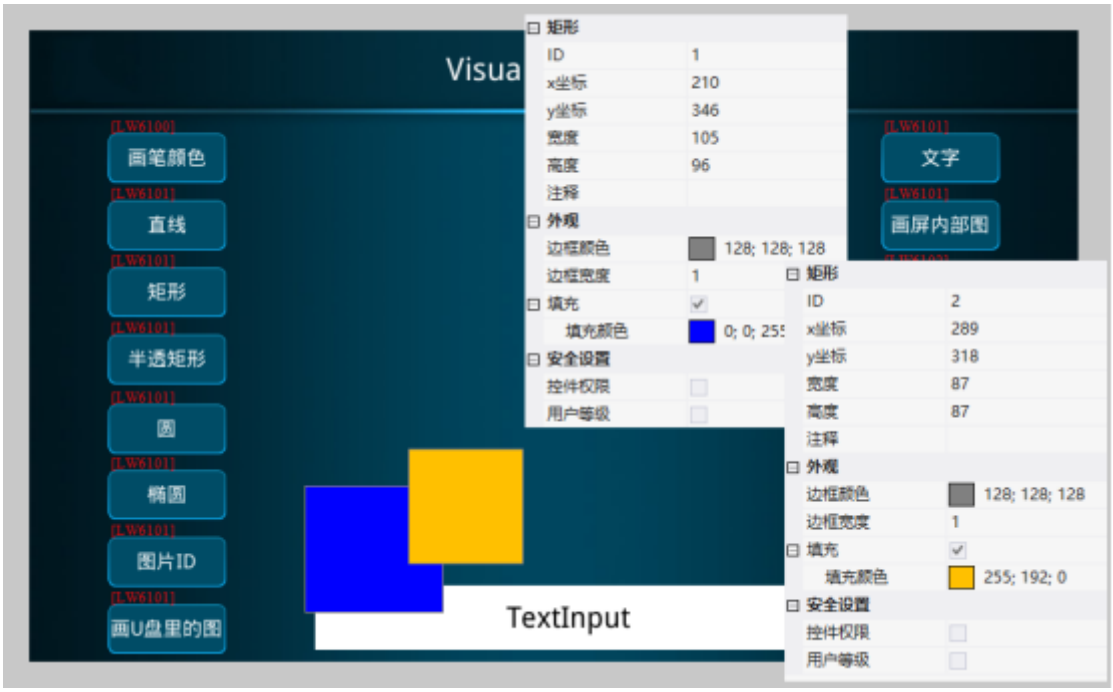
典型图层结构示例

假设界面从底到顶包含以下元素（Z 轴顺序由低到高）：

图层层级	元素类型	实现方式	作用
下层	蓝色背景区域	ID = 10 的矩形控件（蓝色填充）	作为“水果图层”的容器

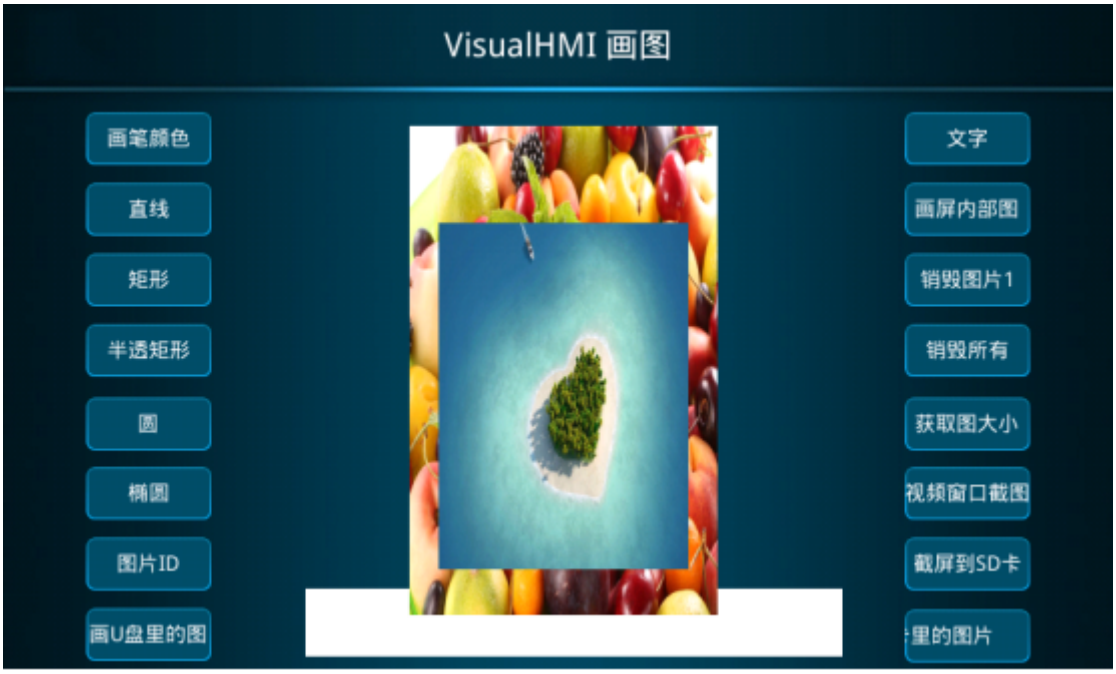
图层层级	元素类型	实现方式	作用
上层	黄色背景区域	ID = 11 的矩形控件（黄色填充）	作为“小岛图层”的容器

> 📌 **注意**：
>
> - 蓝色矩形（ID=10）在编辑器中位于黄色矩形（ID=11）**下方**，因此视觉上被后者部分覆盖；
> - 两个矩形均启用自绘功能（ID ≠ 0），系统将在各自区域触发 `on_draw`。



当系统刷新界面时：

1. **遍历所有需重绘的控件**（按 Z 轴从底到顶）；
2. 对每个支持自绘的控件，**调用一次** `on_draw(screen_id, control_id)`；
3. 开发者通过 `control_id` 判断当前绘制目标：
   - 若 `control_id == 10` → 在**蓝色矩形区域**内绘制“水果图片”；
   - 若 `control_id == 11` → 在**黄色矩形区域**内绘制“小岛图片”；
4. 最终合成效果：
   原本画面（底）← 蓝色背景 + 水果（中）← 黄色背景 + 小岛（顶）




```
function on_draw(screen_id, control_id)
    if screen_id == 0 and control_id == 1
    then
        draw_surface(surface[1], 293, 88, 222, 353, 0, 0)  --裁剪显示

    elseif screen_id == 0 and control_id == 2
    then
        draw_surface(surface[2], 314, 158, 180, 250, 0, 0) --裁剪显示
    end
end
```

3.2.redraw()

全局界面重绘触发函数，`redraw()` 是 HMI 系统提供的**全局重绘请求接口**，用于**主动触发当前画面所有支持自绘的控件执行** `on_draw` **回调函数**，强制刷新界面显示内容。

3.3.set_pen_color(color)

设置绘图画笔颜色，`set_pen_color(color)` 是 HMI 系统提供的**设置画笔颜色函数**，用于**设置后续所有绘图操作的前景色（画笔颜色）**，适用于线条、矩形边框、圆形轮廓、文本描边等**非填充型或边框型图形的颜色。**

📊 **参数说明**

参数	类型	说明

参数	类型	说明
`color`	number	**画笔颜色值**（RGB565 格式） • 高 5 位为 Red（0～31） • 中 6 位为 Green（0～63） • 低 5 位为 Blue（0～31） • 取值范围：`0x0000`（黑）至 `0xFFFF`（白） • 用于后续 `draw_line`、`draw_rect`、`draw_circle` 等矢量图形的边框或线条颜色

RGB565 编码示例：

颜色	RGB (888)	RGB565 (Hex)	Lua 常量建议
红色	(255, 0, 0)	`0xF800`	`COLOR_RED = 0xF800`
绿色	(0, 255, 0)	`0x07E0`	`COLOR_GREEN = 0x07E0`
蓝色	(0, 0, 255)	`0x001F`	`COLOR_BLUE = 0x001F`
白色	(255,255,255)	`0xFFFF`	`COLOR_WHITE = 0xFFFF`
黑色	(0, 0, 0)	`0x0000`	`COLOR_BLACK = 0x0000`

3.4.draw_line(x0,y0,x1,y1,width)

直线绘制函数，`draw_line` 是 HMI **绘制一条指定起点、终点及线宽的直线段。**该线段**颜色**依赖**当前画笔颜**（由 `set_pen_color` 设定）

📊 **参数说明**

参数	类型	说明
`x0`	number	**起始点 X 轴坐标**： 直线起点的水平位置，取值范围依据实际屏幕分辨率而定
`y0`	number	**起始点 Y 轴坐标**： 直线起点的垂直位置，取值范围依据实际屏幕分辨率而定
`x1`	number	**结束点 X 轴坐标**： 直线终点的水平位置，取值范围依据实际屏幕分辨率而定
`y1`	number	**结束点 Y 轴坐标**： 直线终点的垂直位置，取值范围依据实际屏幕分辨率而定
`width`	number	**线条的厚度**： 表示绘制直线时线条的宽度，有效取值范围为 1 到 10

💡 **示例**

```
draw_type = 0

mode = {   --绘制类型表
```

```
    line = 1,
}

function on_update(slave,vtype,addr)

    if addr == 0x1000
    then
        draw_type = get_uint16(VT_LW, addr) --获取字设置按钮键值
        redraw()--绘图
     end
end

function on_draw(screen_id,control_id)

   local switch = {
        [mode.line] = function(control)
            if screen_id == 0 and control == 1
            then
                --设置画笔黄色划线
                set_pen_color(0xFFE0)
                draw_line(225, 253, 405, 253)

                --设置画笔红色，划线
                set_pen_color(0xF800)
                draw_line(508, 128, 508, 378, 5)
            end
        end,
    end

    if switch[draw_type]
    then
        switch[draw_type](control)
    end
end
```

3.5.draw_rect(x0,y0,x1,y1,fill)

矩形绘制函数，`draw_rect` 函数是HMI（人机交互界面）开发中常用的绘图接口之一，通过定义左上角和右下角的坐标以及是否填充该矩形，在**指定位置绘制一个矩形。**

📊 **参数说明**

参数	类型	说明
`x0`	number	**左上角 X 坐标** • 相对于绘图上下文原点（通常是控件或画布的左上角）的水平偏移量
`y0`	number	**左上角 Y 坐标** • 相对于绘图上下文原点的垂直偏移量
`x1`	number	**右下角 X 坐标** • 定义矩形宽度的结束位置

参数	类型	说明
`y1`	number	**右下角 Y 坐标** • 定义矩形高度的结束位置
`fill`	integer	**填充标志** • 若值为 `1`，则使用当前填充色填充整个矩形 • 若值为 `0`，仅绘制矩形边框

💡 示例

```
draw_type = 0

mode = {  --绘制类型表
    rect = 2,
}

function on_update(slave,vtype,addr)

    if addr == 0x1000
    then
        draw_type = get_uint16(VT_LW, addr)--获取字设置按钮键值
        redraw()--绘图
     end
end

function on_draw(screen_id,control_id)

   local switch = {
        [mode.rect] = function(control)
            if screen_id == 0 and control == 1
            then
                --绘制矩形
                draw_rect(225, 128, (225+180), (128+250), 0)
                draw_rect(418, 163, (418+180), (128+180), 1)
            end
        end,
    end

    if switch[draw_type]
    then
        switch[draw_type](control)
    end
end
```

3.6.draw_rect_alpha(x0,y0,x1,y1,alpha)

半透明实心矩形绘制函数，`draw_rect_alpha` 是 HMI 图形系统提供的**半透矩形填充绘图接口**，用于指定位内绘制一个**带有指定透明度的实心矩形。**该函数常用于实现遮罩层

📊 参数说明

参数	类型	说明
`x0`	number	**矩形左上角 X 坐标** • 相对于当前控件客户区左上角的像素偏移 • 通常 ≥ 0
`y0`	number	**矩形左上角 Y 坐标** •相对于当前控件客户区左上角的像素偏移 • 通常 ≥ 0
`x1`	number	**矩形右下角 X 坐标** • 必须 > `x0`，否则矩形无效或不可见
`y1`	number	**矩形右下角 Y 坐标** • 必须 > `y0`，否则矩形无效或不可见
`alpha`	number	**透明度系数** • 有效范围：0 ~ 255 • 0 = 完全透明（不可见） • 255= 完全不透明 • 中间值实现与背景的 Alpha 混合

示例

```
draw_type = 0

mode = {  --绘制类型表
    rect_alpha = 3,
}

function on_update(slave,vtype,addr)

    if addr == 0x1000
    then
        draw_type = get_uint16(VT_LW, addr)--获取字设置按钮键值
        redraw()--绘图
     end
end

function on_draw(screen_id,control_id)

   local switch = {
        [mode.rect_alpha] = function(control)
            if screen_id == 0 and control == 1
            then
                --绘制半透矩形
                draw_rect_alpha(225, 128, (225+180), (128+250), 30)
                draw_rect_alpha(418, 163, (418+180), (128+180), 60)
            end
        end,
    end

    if switch[draw_type]
    then
        switch[draw_type](control)
    end
```

```
end
```

3.7.draw_circle(x,y,r,fill)

圆形绘制函数，`draw_circle` 是用于绘制圆形的函数。此函数可以基于提供的中心坐标 `(x, y)`、半径 `r` 以及填充标志 `fill` 来绘制一个指定大小和样式的圆形。根据 `fill` 参数的不同，该函数可以绘制出轮廓圆（非填充）或实心圆（填充）

📊 参数说明

参数	类型	默认值	说明
`x`	数字	-	圆的中心点 X 坐标
`y`	数字	-	圆的中心点 Y 坐标
`r`	数字	-	圆的半径，决定了圆的大小
`fill`	数字	0	**填充标志:** • `0` 表示填充圆（实心圆） • 非 `0` 表示绘制圆周线（空心圆），此时值可作为线条厚度

💡 示例

```
draw_type = 0

mode = {  --绘制类型表
    circle = 4,
}

function on_update(slave,vtype,addr)

    if addr == 0x1000
    then
        draw_type = get_uint16(VT_LW, addr)--获取字设置按钮键值
        redraw()--绘图
     end
end

function on_draw(screen_id,control_id)

   local switch = {
        [mode.circle] = function(control)
            if screen_id == 0 and control == 1
            then
                --绘制圆形
                draw_circle(300, 253, 100, 0)
                draw_circle(450, 253, 150, 1)
            end
        end,
    end

    if switch[draw_type]
```

```
    then
        switch[draw_type](control)
    end
end
```

3.8.draw_ellipse(x0,y0,x1,y1,fill)

椭圆绘制函数，`draw_ellipse` 是 HMI 图形系统提供**椭圆绘图接口**，用于绘制一个内切于指定矩形区域的椭圆。通过控制填充模式与边框表现，可灵活实现从实心椭圆到带厚度轮廓的空心椭圆

📊 **参数说明**

参数	类型	说明
`x0`	number	**外接矩形左上角 X 坐标**
`y0`	number	**外接矩形左上角 Y 坐标**
`x1`	number	**外接矩形右下角 X 坐标** • 必须 > `x0`
`y1`	number	**外接矩形右下角 Y 坐标** • 必须 > `y0`
`fill`	number	**填充控制标志** • `0`：绘制实心填充椭圆 • 非 `0`：绘制空心椭圆轮廓，其值通常表示线条厚度（如 `fill=2` 表示 2 像素粗的边框）

💡 **示例**

```
draw_type = 0

mode = {  --绘制类型表
    ellipse = 5,
}

function on_update(slave,vtype,addr)

    if addr == 0x1000
    then
        draw_type = get_uint16(VT_LW, addr)--获取字设置按钮键值
        redraw()--绘图
     end
end

function on_draw(screen_id,control_id)

   local switch = {
        [mode.ellipse] = function(control)
            if screen_id == 0 and control == 1
            then
                --绘制椭圆
```

```
                draw_circle(300, 253, 100, 0)
                draw_circle(450, 253, 150, 1)
            end
        end,
    end

    if switch[draw_type]
    then
        switch[draw_type](control)
    end
end
```

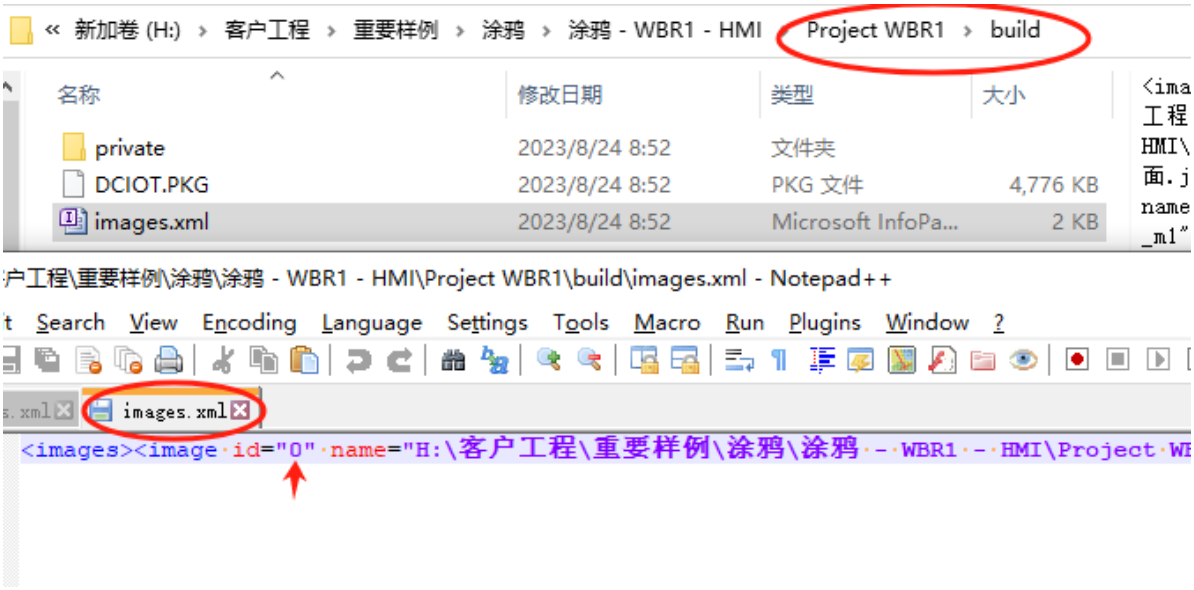
3.9.draw_image(image_id,frame_id,dstx,dsty,width,height,srcx,srcy)

图片绘制函数，`draw_image` 是 HMI系统提供**绘制图片资源**的接口。此函数允许开发者通过指定图片资源ID、显示位置与大小，以及可选的裁剪区域来灵活地控制图像显示。

📊 **参数说明**

参数	类型	说明
`image_id`	number	**图片资源的唯一标识符** • 标识将要绘制的图片资源；
`frame_id`	number	**帧ID** • 对于支持帧动画的图片类型，指明所使用的具体帧； • 对于非动画图片类型，该值通常固定为0；
`dstx`	number	**图片显示的X坐标** • 目标图片左上角相对于客户区左上角的水平偏移量；
`dsty`	number	**图片显示的Y坐标** • 目标图片左上角相对于客户区左上角的垂直偏移量；
`width`	number	**图片显示的宽度** • 指定图片在屏幕上显示时的宽度，实现缩放效果；
`height`	number	**图片显示的高度** • 指定图片在屏幕上显示时的高度，实现缩放效果；
`srcx`	number	**图片裁剪的起始X坐标** • 若不希望从图片的原点开始绘制，可以指定从原图的哪个点开始裁剪；
`srcy`	number	**图片裁剪的起始Y坐**标 • 配合 `srcx` ，定义裁剪矩形的起点；

> [!note|tip:如何确定image_id]
> 1.在工程目录下的../build/文件夹，打开image.xml文件，image id = "xx"表示第一个参数



示例

```
draw_type = 0

mode = {  --绘制类型表
    imageId = 6,
}

function on_update(slave,vtype,addr)

    if addr == 0x1000
    then
        draw_type = get_uint16(VT_LW, addr)--获取字设置按钮键值
        redraw()--绘图
     end
end

function on_draw(screen_id,control_id)

   local switch = {
        [mode.imageId] = function(control)
            if screen_id == 0 and control == 1
            then
                --绘制图片
                draw_image(1, 0, 225, 128, (225+180), (128+250), 0, 0)
                draw_image(2, 0, 418, 163, (418+180), (128+180), 0, 0)
            end
        end,
    end

    if switch[draw_type]
    then
        switch[draw_type](control)
    end
end
```

3.10.draw_text(text,x,y,w,h,font_id,size,color,align,charcode)

文本绘制函数，`draw_text` 是 HMI 图形系统提供的**文字绘制**接口，用于在指定区域内绘制字符串。支持自定义**字体、字号、颜色、对齐方式及字符编码。**

📊 **参数说明**

参数	类型	说明
`text`	string	**待显示的文本内容** • 支持 ASCII 及多字节字符（如 UTF-8、GB2312），具体由 `charcode` 决定
`x`	number	文本区域左上角 X 坐标
`y`	number	文本区域左上角 Y 坐标
`w`	number	文本区域宽度（像素）
`h`	number	文本区域高度（像素）
`font_id`	number	**字体资源 ID** • 对应工程中预加载的字体编号（如 0=默认字体，1=自定义矢量字体）
`size`	number	**字体大小（像素）** • 表示字体高度
`color`	number	**文字颜色（RGB565 格式）** • 高 5 位 Red，中 6 位 Green，低 5 位 Blue • 例如：`0xFFFF` = 白色，`0x0000` = 黑色
`align`	number	**文本对齐方式** • 常见值： `0` = 左对齐 `1` = 居中对齐 `2` = 右对齐 • 部分平台支持垂直对齐组合（需查手册）
`charcode`	number	**字符编码类型** • 指定 `text` 字符串的编码格式，选填。0 或不填，默认 lua 文件 UTF-8 编。1 位 GBK

💡 **示例**

```
draw_type = 0

mode = {  --绘制类型表
    text           = 7,
}

function on_update(slave,vtype,addr)

    if addr == 0x1000
    then
        draw_type = get_uint16(VT_LW, addr)--获取字设置按钮键值
        redraw()--绘图
     end
```

```
end

function on_draw(screen_id,control_id)

   local switch = {
        [mode.text] = function(control)
            if screen_id == 0 and control == 1
            then
                --绘制文字
                draw_text('广州大彩光电科技有限公司\nCopyright? gz-dc.com ', 195, 96,
408 ,318, 0, 32, 0xF800, 5, 1)
            end
        end,
    end

    if switch[draw_type]
    then
        switch[draw_type](control)
    end
end
```

3.11.load_surface (filename)

图像加载函数,`load_surface` 用于从指定路径加载 JPEG 或 PNG 格式的图片文件(**外部图形,非工程打包的image.bin里的图片资源**)，并将其作为句柄（surface）返回。`surface`后续绘图操作中被引用。

📊 参数说明

参数	类型	说明
`filename`	string	**待加载的图片文件路径：** 支持的格式，**JPEG, PNG**

⚠️ 关键注意事项

- **路径准确性**：确保提供的 `filename` 路径准确无误，指向有效的 JPEG 或 PNG 文件。
- **内存占用**：大尺寸图像或大量图像的同时加载会占用较多内存，应注意资源管理。
- **透明度处理**：对于带有透明度信息的 PNG 图像，占用内存比较大

💡 示例

```
if surface > 0
then
    destroy_surface(surface)
end
surface = load_surface('3:/test.jpg')
```

3.12.destroy_surface (surface)

图层资源销毁函数，`destroy_surface(surface)` 用于**释放"由** `load_surface` **"申请的图形资源。**调用后，该图层指针失效，不可再用于任何绘图操作

📊 **参数说明**

参数	类型	说明
`surface`	number	**图层资源句柄** • 由 `load_surface()`、函数返回的有效指针或对象引用

✅ **核心用途**:

- 防止内存泄漏，尤其在动态加载/卸载图片的场景中；
- 释放 GPU 或帧缓冲区中缓存的图像数据；
- 管理有限的嵌入式系统图形资源。

💡 **示例**

```
destroy_surface(surface)
```

3.13.destroy_all_surface()

批量销毁所有图层资源函数,`destroy_all_surface()` 是 HMI 图形系统提供的**全局图层资源清理接口**，用于**一次性释放当前脚本上下文中所有已加载的图层（surface）所占用的内存资源。**调用后，所有通过 `load_surface` 创建的图层句柄均失效

✅ **核心用途**:

- 快速回收全部图像资源，避免逐个管理；
- 简化资源管理逻辑，防止因遗漏导致的内存泄漏。

3.14.draw_surface (surface,dstx,dsty,width,height,srcx,srcy)

`draw_surface` 是绘制**由** `load_surface` **申请的图片资源** 的接口，将已加载的图层资源（如 JPEG/PNG 图片）**绘制到指定位置。**支持**区域裁剪**与**尺寸缩放。**该函数**必须在** `on_draw` **回调中调用**才能生效。

📊 **参数说明**

参数	类型	说明
`surface`	number	**图层资源句柄** • 由 `load_surface()` 返回的有效图像对象
`dstx`	number	目标显示区域左上角 X 坐标
`dsty`	number	目标显示区域左上角 Y 坐标
`width`	number	**目标显示宽度（可选）** • 原图将水平缩放至该宽度 • 若省略或设为 `0`，使用原图宽度

参数	类型	说明
`height`	number	**目标显示高度（可选）** • 原图将垂直缩放至该高度 • 若省略或设为 `0`，使用原图高度
`srcx`	number	**源图裁剪起始 X 坐标（可选）** • 从原图 `(srcx, srcy)` 开始截取内容 • 若省略或设为 `0`，从原图左上角开始
`srcy`	number	**源图裁剪起始 Y 坐标（可选）** • 配合 `srcx` 定义裁剪起点 • 若省略或设为 `0`，从原图左上角开始

💡 示例

```
draw_type = 0
surface = 0
mode = {  --绘制类型表
    surface_path3 = 8,
}

function on_update(slave,vtype,addr)

    if addr == 0x1000
    then
        if surface > 0
        then
            destroy_surface(surface)
        end
        surface = load_surface('3:/test.jpg')
        --surface = load_surface('1:/test.jpg') --SD卡图片
        --surface = load_surface('2:/test.jpg') --U盘图片
        redraw()--绘图
     end
end

function on_draw(screen_id,control_id)

   local switch = {
        [mode.surface_path3] = function(control)
            if screen_id == 0 and control == 1
            then
                --绘制图片
                draw_surface(surface, 175, 60, 222, 353, 0, 0)
            end
        end,
    end

    if switch[draw_type]
    then
        switch[draw_type](control)
    end
end
```

3.15.get_surface_size (surface)

获取图层尺寸函数，`get_surface_size(surface)` 是 HMI 图形系统提供的**图层属性查询接口**，用于**获取已加载图层（surface）的原始像素尺寸**。该函数返回图像的宽度和高度

📊 **参数说明**

参数	类型	说明
`surface`	number	**图层资源句柄** • 由 `load_surface()` 返回的有效图像对象

💡 **示例**

```
local cur_w,cur_h = get_surface_size (surface)
```

3.16.clear_image_buffer()

清除内部图片资源缓存函数，`clear_image_buffer()` 是 HMI 系统提供的**全局图像缓存管理接口**，用于**强制释放图形的图片数据**，以降低内存占用

3.17.screen_shoot(filepath)

屏幕截图函数，`screen_shoot(filepath)` 是 HMI 系统提供的**屏幕内容捕获接口**，用于将当前显示画面（或指定区域）保存为图像文件（JPEG 格式）

📊 **参数说明**

参数	类型	说明
`filepath`	string	**截图保存的完整文件路径** • 必须包含文件名及扩展名（如 `1.jpg`）

4.数据记录

4.1.record_get_count(sn)

获取资料采样记录总条数函数，`record_get_count(sn)` 是 HMI 系统提供的**数据记录查询接口**，用于获取指定资料采样通道（索引为 `sn`）中**已存储的历史记录总条数**

📊 **参数说明**

参数	类型	说明
`sn`	number	**资料采样通道索引** • 取值范围：0 ~ 19（共 20 个通道）

💡 **示例**

```
local cnt = record_get_count(idx)
```

4.2.record_modify_data(sn, index, data)

修改资料记录内容函数，`record_modify_data(sn, index, data)` 是 HMI 系统提供的**数据记录修改接口**，用于**修改指定资料采样通道中某一条历史记录的数据内容**

📊 **参数说明**

参数	类型	说明
`sn`	number	**资料记录通道索引** • 取值范围：0 ~ 19 • 对应工程中配置的第 `sn` 个采样通道
`index`	number	**记录行号（从 0 开始）** • 表示要修改的第 `index` 条记录 • 必须小于 `record_get_count(sn)` 返回值
`data`	table (array)	**待写入的数据数组** • 字一维数组，Lua 表索引从 1 开始

⚠️ **重要限制**：若该资料采样已启用**块地址存储模式**，数据不可篡改性。

💡 **示例1**

1. 假设修改第0个资料采样，**为uint16，共100个数据**，则修改第0行记录为1~100实例如下：

```
for i = 1, 100
do
    dataTb[i] = i
end
record_modify_data(0, 0, data)
```

💡 **示例2**

2. 假设修改第0个资料采样，**为uint32，共100个数据**，则修改第0行记录为1~100实例如下：

```
for i = 1, 200
do
    if i % 2 == 0
    then
        dataTb[i] = i
    else
        dataTb[i] = 0
    end
end
record_modify_data(0, 0, data)
```

uint32共100个参数，共占200个字，所有，data的大小为200。依次类推，64位寄存器为400

4.3.record_write_data(sn, data)

添加资料记录函数，`record_write_data(sn, data)` 是 HMI 系统提供的**数据记录写入接口**，用于**向指定资料采样通道追加一条新的记录。**

📊 **参数说明**

参数	类型	说明
`sn`	number	**资料记录通道索引** • 取值范围：0 ~ 19 • 对应工程中已配置的第 `sn` 个采样通道
`data`	table (array)	**待写入的数据数组** • 字一维数组，Lua 表索引从 1 开始

💡 **示例1**

1. 假设第0个资料采样，**为uint16，共100个数据**，添加一条记录，则添加内容为1~100实例如下：

```
for i = 1, 100
do
    dataTb[i] = i
end
record_write_data(0, data)
```

💡 **示例2**

2. 假设第0个资料采样，**为uint32，共100个数据**，添加一条记录，则添加内容为1~100实例如下：

```
for i = 1, 200
do
    if i % 2 == 0
    then
        dataTb[i] = i
    else
        dataTb[i] = 0
    end
end
record_write_data(0, data)
```

4.4.record_read_data(sn, index)

读取资料记录内容函数，`record_read_data(sn, index)` 是 HMI 系统提供的**历史数据查询接口**，用于从指定资料采样通道中**读取某一条记录的原始数据及其时间戳。**

📊 **参数说明**

参数	类型	说明

参数	类型	说明
`sn`	number	**资料记录通道索引** • 取值范围：0 ~ 19 • 对应工程中配置的第 `sn` 个采样通道
`index`	number	**记录行号（从 0 开始）** • 表示要读取的第 `index` 条记录 • 必须满足 `0 ≤ index < record_get_count(sn)`

返回值

返回值	类型	说明
`data`	table (array)	**记录数据数组** • 元素为 **16 位无符号整数（uint16）** • **Lua 表索引从 1 开始**，即 `data[1]` 为记录的第一个 word 字段
`timestamp`	number	**32 位 Unix 时间戳（秒）**

💡 **示例1**

1. 假设第0个资料采样，**为uint16，共100个数据**，读取第0个资料采样，第0条数据

```
local data,timestamp =  record_read_data(0, 0)
print('size'..(#data))

>size = 100
```

💡 **示例2**

2. 假设第0个资料采样，**为uint32，共100个数据**，读取第0个资料采样，第0条数据

```
local data,timestamp =  record_read_data(0, 0)
print('size'..(#data))

>size = 200
```

4.5.record_modify_string(sn, index, strings)

修改字符串类型资料记录函数，`record_modify_string(sn, index, strings)` 是 HMI 系统为**字符串型资料记录通道**提供的专用接口，用于**修改指定记录行中的内容。每一列数据**以分号 `;` 分隔的字符串形式传入

📊 **参数说明**

参数	类型	说明

参数	类型	说明
`sn`	number	**资料记录通道索引** • 取值范围：0 ~ 19 • 必须对应一个配置为字符串类型的采样通道
`index`	number	**记录行号（从 0 开始）** • 表示要修改的第 `index` 条记录 • 必须满足 `0 ≤ index < record_get_count(sn)`
`strings`	string	**分号分隔的字符串字段序列** • 格式：`"字段1;字段2;字段3;..."`

⚠️ **重要限制**：若该资料采样已启用**块地址存储模式**，数据不可篡改性。

💡 **示例**

```
--修改第0笔资料，第1行数据的数据为‘item1;item2;item3;item4;item5;’
record_modify_string(0, 0,‘item1;item2;item3;item4;item5;’)
```

4.6.record_read_string(sn, index)

读取字符串类型资料记录函数，`record_read_string(sn, index)` 是 HMI 系统为**字符串型资料记录通道**提供的专用查询接口，用于**读取指定记录行的内容及其时间戳。**返回的数据以分号 `;` 分隔的字符串

📊 **参数说明**

参数	类型	说明
`sn`	number	**资料记录通道索引** • 取值范围：0 ~ 19
`index`	number	**记录行号（从 0 开始）** • 表示要读取的第 `index` 条记录 • 必须满足 `0 ≤ index < record_get_count(sn)`

返回值

返回值	类型	说明
`strings`	string	**分号分隔的字符串字段序列** • 格式：`"字段1;字段2;字段3;..."`
`timestamp`	number	**32 位时间戳（秒）**

💡 **示例**

假设第0个资料采，**为string**，读取第0个资料采样，第0条数据

```
local strings, timestamp = record_read_string(0, 0)
print('strings = '..strings)

> strings = item1;item2;item3;item4;item5;
```

4.7.record_write_string(sn, strings)

添加字符串类型资料记录函数，`record_write_string(sn, strings)` 是 HMI 系统为**字符串型资料记录通道**提供的专用写入接口，用于**向指定通道追加一条新的文本记录。**数据以分号 `;` 分隔的字符串形式传入。

📊 **参数说明**

参数	类型	说明
`sn`	number	**资料记录通道索引** • 取值范围：0 ~ 19
`strings`	string	**分号分隔的字符串字段序列** • 格式：`"字段1;字段2;字段3;..."`

💡 **示例**

假设第0个资料采，**为string**，添加一条记录**"item1;item2;item3;item4;item5;"**

```
--第0笔资料，添加一条记录'item1;item2;item3;item4;item5;'
 record_write_string( 0,'item1;item2;item3;item4;item5;')
```

4.8.record_clear(sn)

清除资料记录函数，`record_clear(sn)` 是 HMI 系统提供的**资料记录重置接口**，用于**清空指定通道（`sn`）中的所有历史记录数据。**

📊 **参数说明**

参数	类型	说明
`sn`	number	**资料记录通道索引** • 取值范围：0 ~ 19 • 对应工程中配置的第 `sn` 个采样通道（数值型或字符串型均适用）

4.9.on_parse_record(sn,export,datatb)

资料采样记录解析回调函数，`on_parse_record(sn, export, datatb)` 是 HMI 系统提供的**资料记录自定义解析回调接口**，属于**定制功能。**当系统需要**显示**某条资料采样记录时，会自动调用此函数，允许开发者将原始 `word` 数组（`datatb`）转换为**自定义显示格式**

📊 **参数说明**

参数	类型	说明
`sn`	number	**资料采样通道索引** • 取值范围：0 ~ 19
`export`	number	**操作类型标志** • `1`：数据导出中 • `0`：数据未导出
`datatb`	table (array)	**原始记录数据数组** • 元素为 16 位无符号整数（uint16） • Lua 表索引从 1 开始，即 `datatb[1]` 为第一个 word 字段 • 长度由通道配置决定

适用于**超长(很多列)**、**每列数据格式不一样**的应用，搭配`on_get_record_channel(index,channel)`使用

4.10.on_get_record_channel(index,channel)

表格单元格数据格式化解析回调函数，`on_get_record_channel`是 HMI 系统为**资料采样表格控件**提供的**单元格级自定义格式化回调接口（定制功能）**。当表格控件需要渲染**当前视口（可见区域）内某一单元格**时，系统会调用此函数，允许开发者将原始记录中的某字段（word 数据）转换为带单位、小数位数等

📊 **参数说明**

参数	类型	说明
`index`	number	**资料采样通道索引** • 取值范围：0 ~ 19
`channel`	number	**通道索引（从 0 开始）** • 表示当前要解析的是第 `channel` 列（即第几个字段）

适用于**超长(很多列)**、**每列数据格式不一样**的应用，搭配`on_parse_record(index,export,datatb)`使用

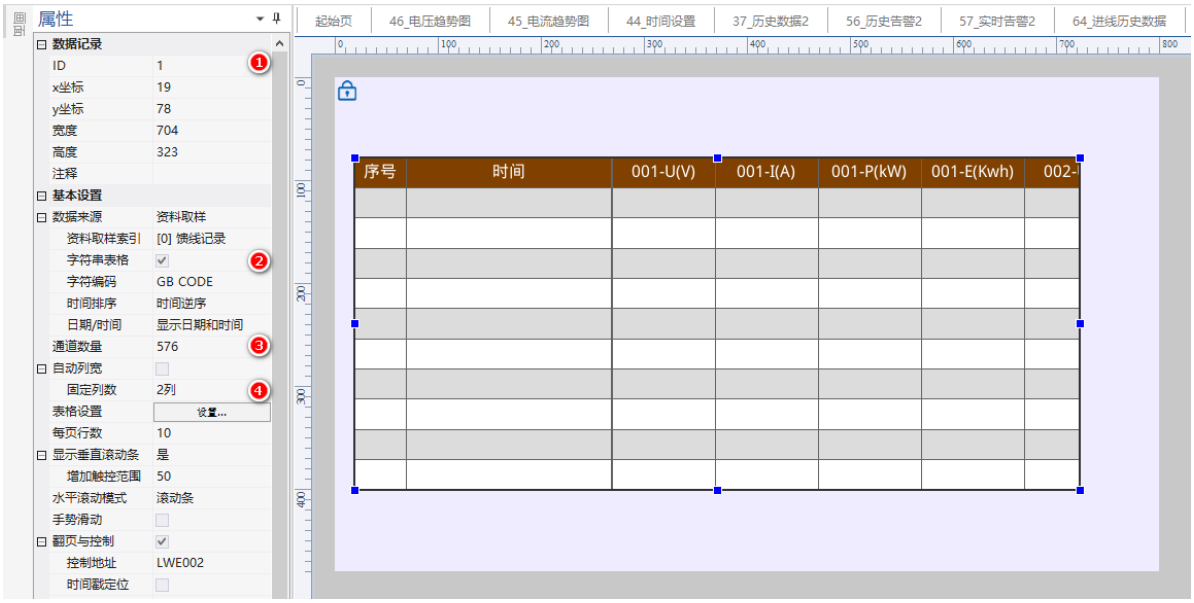
💡 **应用场景： on_parse_record + on_get_record_channel**

在列头柜智能监控系统中，需要记录**144路支路×4参数=576个动态监测点**（电压Int16/1位小数、电流Uint16/2位小数、功率Uint32/3位小数、电能Uint32/2位小数），形成工业现场典型的**超宽数据矩阵。**VisualHMI 通过**动态列按需解析**的架构，在资源受限的嵌入式HMI上实现**毫秒级响应、零卡顿滚动**的工业级数据可视化体验。

资料采样配置：

采样设置	
描述信息	馈线记录
采样模式	触发采样
记录条数	2000
数据地址	LWE100
数据类型	STRING
数据个数	864
高低互换	☐
触发地址	LWE000
模式	OFF->ON
复位	☑
控制地址	LWE001
导出标题	$馈线记录
采样控制地址	☐
掉电存储	是
存储块地址	2600
占用块数	1002

数据记录表格配置：



双回调协同机制：数据流与计算流的精密解耦

📦 数据注入层（`on_parse_record`）

在Lua脚本`on_parse_record`回调中，datatb大小为864字

- 若HMI当前没有导出记录`export = 0`，则赋值给全局变量`g_switch_datatb = datatb，return nil`
  - `g_switch_datatb[1..864]` （支路1: [1]=电压, [2]=电流, [3-4]=功率, [5-6]=电能 → 支路144: [859..864]）
- 若HMI当前需要导出记录`export = 1`，该回调函数会触发N次（当前记录N条记录）。每次回调，拼接144个支路参数形成一条记录，并返回`return record_msg`

```
function on_parse_record(index,export,datatb)

    local record_msg = ''

    if index == 0
    then
        if export == 0 --非导出
        then
             g_switch_datatb = datatb
             return nil

        else --导出,拼接每行数据导出

            for i = 1, #datatb,  6
            do
                local u = datatb[i] --INT16
                u = ((u  &  0x8000) == 0x8000) and ((0xFFFF - u - 1)*(-1)) or u

                local a = datatb[i + 1] --UINT16
                local p = ((datatb[i + 2] << 16) | datatb[i + 3]) & 0xFFFFFFFF -
-UINT32
                local e = ((datatb[i + 4] << 16) | datatb[i + 5]) & 0xFFFFFFFF -
-UINT32

                u = string.format('%0.1f', (u /   10))..';'
                a = string.format('%0.2f', (a /  100))..';'
                p = string.format('%0.3f', (p / 1000))..';'
                e = string.format('%0.2f', (e /  100))..';'
                record_msg = record_msg..u..a..p..e

           end
        end
    end

    return record_msg
end
```

单元格解析层（`on_get_record_channel`）

解析当前视口（可见区域）内的单元格数据格式，并返回。假设：显示第10～20行中的第4～8列动态列时，系统自动忽略不可见列，精准调用55次，函数依据`channel`参数计算，如下所示：
1️⃣ **数学映射定位** → 根据`channel`计算支路参数在`g_switch_datatb`中的偏移

```
local byPass = (channel // 4) --1~144
local idx = (channel) % 4 --第几个参数/第几列
```

2️⃣ **格式化数据** → 执行符号处理/高低字拼接/缩放系数转换

```
if idx == 0 --电压 INT16
then
    val = g_switch_datatb[(byPass)*6 + 1]
    val = ((val  &  0x8000) == 0x8000) and ((0xFFFF - val - 1)*(-1)) or val
    val = string.format('%0.1f', (val /   10))
```

```
elseif idx == 1 --电流 UINT16
then
    val = g_switch_datatb[(byPass)*6 + 2]
    val = string.format('%0.2f', (val /  100)

elseif idx == 2 --功率 UINT32
then
    val = ((g_switch_datatb[(byPass)*6 + 3] << 16) | g_switch_datatb[(byPass)*6
+ 4]) & 0xFFFFFFFF
    val = string.format('%0.3f', (val / 1000))

elseif idx == 3 --电能 UINT32
then
    val = ((g_switch_datatb[(byPass)*6 + 5] << 16) | g_switch_datatb[(byPass)*6
+ 6]) & 0xFFFFFFFF --UINT32
    val = string.format('%0.2f', (val /  100))

end
```

处理**解析当前视口（可见区域）内的单元格数据格式**完整代码如下所示：

```
--表格每通道回调
--index：资料采集索引
--channel:通道,显示当前第几通道
function on_get_record_channel(index,channel) --当下发滚动条拖动时候，channel实时变化

    if index == 0
    then

        local val = 0
        local byPass = (channel // 4) --1~144
        local idx = (channel) % 4 --第几个参数

        if idx == 0 --电压 INT16
        then
            val = g_switch_datatb[(byPass)*6 + 1]
            val = ((val  &  0x8000) == 0x8000) and ((0xFFFF - val - 1)*(-1)) or
val
            val = string.format('%0.1f', (val /   10))

        elseif idx == 1 --电流 UINT16
        then
            val = g_switch_datatb[(byPass)*6 + 2]
            val = string.format('%0.2f', (val /  100)

        elseif idx == 2 --功率 UINT32
        then
            val = ((g_switch_datatb[(byPass)*6 + 3] << 16) |
g_switch_datatb[(byPass)*6 + 4]) & 0xFFFFFFFF
            val = string.format('%0.3f', (val / 1000))

        elseif idx == 3 --电能 UINT32
        then
            val = ((g_switch_datatb[(byPass)*6 + 5] << 16) |
g_switch_datatb[(byPass)*6 + 6]) & 0xFFFFFFFF --UINT32
```

```
            val = string.format('%0.2f', (val /  100))

        end

        return val
    end

end
```

5.告警

在 HMI 系统中，常规告警需求应优先使用**内置的“告警设置”进行配置**（图形化、可靠、支持历史记录与确认机制）。本文所述的**脚本告警 API 仅适用于特殊场景**：当告**警条目数量庞大、内容高度结构化、触发条件具有逻辑规律时**，通过脚本动态生成和管理告警可显著提升开发效率与系统灵活性。

5.1.warning_set_mode(en)

告警触发模式控制函数，`warning_set_mode(en)` 是 HMI 脚本告警系统的**全局使能开关**，用于**启用后，需要** `warning_set` **函数的告警触发。**

📊 参数说明

参数	类型	说明
`en`	number	告警模式使能标志 • `1`：开启Lua解析告警模式 • `0`：关闭Lua解析告警模式

💡 示例

```
ENCRYPT_=0     --LUA脚本加密

--数据类型定义
VT_LW = 1     --变量地址
VT_RW = 2     --FLASH存储

function on_init()
    warning_set_mode(1) --LUA解析告警
end
```

5.2.warning_set(warning_id,value,count)

批量告警状态设置函数，`warning_set()` 是 HMI 脚本告警系统的核心操作接口，用于**按位批量触发或清除连续的告警条目。**该函数将 `value` 视为一个**位掩码（bitmap）**，其低 `count` 位分别对应从 `warning_id` 开始的 `count` 个告警的状态（1=触发，0=清除）。

🔑 **前提条件**：必须先调用 `warning_set_mode(1)` 启用脚本告警模式，否则本函数无效。

📊 **参数说明**

参数	类型	说明
`warning_id`	number	**起始告警 ID（从 0 开始）**
`value`	number	**告警状态位图（16 位）** • 最低位（bit 0）对应 `warning_id` • bit 1 对应 `warning_id + 1`，依此类推 • 每BIT：`1` = 触发告警，`0` = 清除告警
`count`	number	**连续操作的告警数量** • 取值范围：1 ~ 16

💡 示例

```
--读取寄存器值
local err_val = get_uint16(VT_V, 0x1D00)

--将寄存器值的每一个位，对应到告警0~15范围
warning_set(0, err_val , 16)
```

5.3.on_parse_warning(id, text, screen_id, control_id, p0, p1, p2, p3)

告警内容动态解析回调函数，`on_parse_warning` 是 HMI 系统提供的**告警文本自定义生成接口**，开发者根据**告警 ID、画面信息**，生成**自定义**的告警描述字符串。

📊 参数说明

参数	类型	说明
`id`	number	*告警 ID** • 对应 `warning_set` 中使用的 `warning_id` • 用于区分不同类型的告警
`text`	string	**原始告警文本** • 此处可以忽略，因为本回调中，拼接新的字符串返回
`screen_id`	number	当前画面 ID
`control_id`	number	当前控件 ID
`p0` ~ `p3`	number	告警参数（0 ~ 3） • 仅在工程中为该告警启用了参数功能时有效 • 实际值由 `on_get_warning_param` 回调提供（见下方说明）

返回值

返回值	类型	说明
`str`	string	**告警描述字符串** • 支持拼接变量、单位、状态等 • 示例：`"电机#3 温度 98°C 超过阈值 95°C! "`，带参配合`on_get_warning_param` `实现

返回值	类型	说明
`encode`	number	**字符编码标识（可选）** • `1`：表示返回字符串为 **UTF-8 编码**（默认） • `0`：**GBK-8 编码**

示例

```
_warningTb = {
--1~96
    '翅片1温度探头故障',
    '翅片2温度探头故障',
    '翅片3温度探头故障',
    '翅片4温度探头故障',
    '回气1温度探头故障',
    '回气2温度探头故障',
    '回气3温度探头故障',
    '回气4温度探头故障',
    '排气1温度探头故障',
    '排气2温度探头故障',
    '排气3温度探头故障',
    '排气4温度探头故障',
    '蒸发1温度探头故障',
    '蒸发2温度探头故障',
    '蒸发3温度探头故障',
    '蒸发4温度探头故障',

    '冷凝1温度探头故障',
    '冷凝2温度探头故障',
    '冷凝3温度探头故障',
    '冷凝4温度探头故障',
    '房内温度探头故障',
    ...
    ...
    ...
    '室内翅片1探头故障',
    '室内翅片2探头故障',
    '室内翅片3探头故障',
    '室内翅片4探头故障',
}

function on_parse_warning(id, text, screen_id, control_id, p0, p1, p2, p3)

--设备1:id0~id95
--...
--...
--...
--设备8:id673~id768

    local slaveId = (id // 96) + 1    --第几个告警
    local msgId = id % 96             --告警类型ID
    local curWarnMsg = ''             --告警描述

    msgId = (msgId == 0) and 1 or (msgId + 1)
```

```
    curWarnMsg = math.ceil(slaveId)..'device : '..warningTb[msgId]

    return curWarnMsg
end
```

5.4.on_get_warning_param(warning_id)

告警参数动态回调函数，`on_get_warning_param(warning_id)` 是 HMI 系统提供的**告警参数设置接口**，当工程告警设置中为**启用"告警参数"功能**后，系统在显示告警详情时自动调用此函数，通过告警 ID，根据具体业务逻辑，返回该告警的参数，支持返回最多 4 个数值/状态参数（`data0` ~ `data3`）；实现**个性化、可读性强的告警提示**

📊 **参数说明**

参数名	类型	说明
`warning_id`	number	告警唯一标识符

📊 **返回值说明**

返回值	类型	必填	说明
`data0`	number	否	告警参数 0 • 用于填充告警设置→中的 `{#0:num1.num2}` • 或者on_parse_warning(...)中的p0
`data1`	number	否	告警参数 1 • 用于填充告警设置→中的 `{#1:num1.num2}` • 或者on_parse_warning(...)中的p1
`data2`	number	否	告警参数 2 • 用于填充告警设置→中 `{#2:num1.num2}` • 或者on_parse_warning(...)中的p2
`data3`	number	否	告警参数 3 • 用于填充告警设置→中 `{#3:num1.num2}` • 或者on_parse_warning(...)中的p3

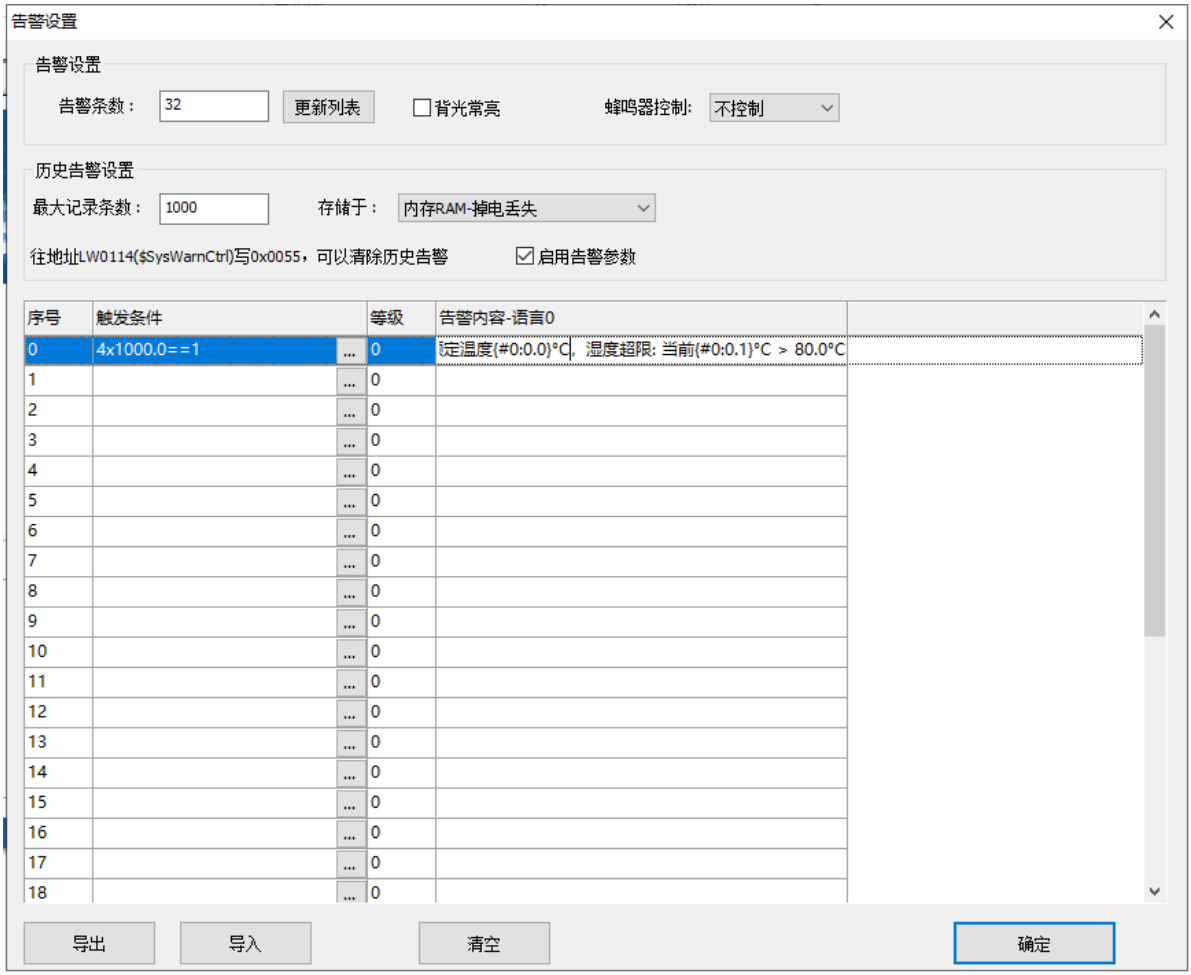
💡 **示例1**：Lua处理

```
-- 假设 warning_id = 0 表示温度告警
function on_get_warning_param(warning_id)
    if warning_id == ALARM_TEMP_OVER then
        local current_temp = get_uint16(VT_LW, ADDR_TEMP_SENSOR)
        local threshold = get_uint16(VT_LW,ADDR_TEMP_THRESHOLD)/10
        return current_temp, threshold  -- data0=current, data1=threshold
    end
    -- 其他告警...
end
function on_parse_warning(id, text,  screen_id, control_id, p0, p1, p2, p3)
```

```
    if id == ALARM_TEMP_OVER and p0 ~= nil p1 ~= nil then
    then
        return "温度超限：当前温度"..p0.."°C > 额定温度26°C，湿度超限： 当前"..p1.."°C > 80.0°C"
    end

end
-- 告警内容举例："温度超限：当前温度32°C > 额定温度26°C，湿度超限： 当前85.3°C > 80.0°C"
```

示例2：告警设置



5.5.on_filter_warning(warning_id)[定制]

告警分类显示过滤回调函数，`on_filter_warning(warning_id)` 是 HMI 系统提供的**告警控制接口**，属于**定制**功能。当一条告警被触发时，系统调用此函数，允许开发者根据 `warning_id` **,当前的画面ID，返回0 or 1（表示允许显示）**。

参数说明

参数	类型	说明
`warning_id`	number	**告警 ID** • 与 `warning_set` 中使用的 ID 一致 • 用于识别告警类型或来源

💡 **示例**

```
function on_filter_warning(warning_id)

    print('on filter warning('..warning_id..')')
    local ret = 0
    local cur_page = get_screen()
    if warning_id < 100 and cur_page == 0
    then
        ret = 1
    end

    if warning_id > 100 and cur_page == 2
    then
        ret = 1
    end
    return ret
end
```

5.6.on_select_warning(screen_id,control_id,warning_id,starttime,stoptime,warning_text)[定制]

告警选中事件回调函数（定制功能），`on_select_warning` 是 HMI 系统提供的**告警交互响应接口**，属于**定制固件专属功能。**当用户在告警列表控件中**点击或选中某条告警记录**时，系统自动调用此函数，向脚本传递该告警的完整上下文信息，用于触发**深度诊断、详细信息**等

⚠️ **重要限制**：

- 仅在**启用定制固件**的设备上可用；
- 若工程中为该告警**启用了"告警参数"功能**，此回调**不会被触发**（二者互斥）。

📊 **参数说明**

参数	类型	说明
`screen_id`	number	**当前画面 ID** • 告警列表所在的页面标识
`control_id`	number	**告警列表控件 ID** • 触发选中事件的具体控件ID
`warning_id`	number	**告警序号（ID）** • 对应工程"告警设置"中的告警编号（从 0 开始）
`starttime`	number	**告警触发时间** • 32 位时间戳（秒）
`stoptime`	number	**告警解除时间** • 32 位时间戳（秒）
`warning_text`	string	告警原始描述文本

6.定时器

6.1.start_timer(timer_id, timeout, countdown, repeat)

启动脚本定时器函数，`start_timer` 是 HMI 系统提供**软件定时器**。当定时器超时后，系统将自动调用全局回调函数 `on_timer(timer_id)`，从而实现周期性任务、延时操作、状态轮询等逻辑。

⚠️ **注意**：此定时器为**应用层软件定时器**，精度受系统负载影响，**不适用于硬实时控制**。

📊 **参数说明**

参数	类型	说明
`timer_id`	number	**定时器 ID** • 取值范围：0 ~ 31（共 32 个独立定时器）
`timeout`	number	**超时时间（毫秒）** • 最小值通常为 10~20ms左右（依平台而定）
`countdown`	number	**计时模式** • `0`：顺计时（从 0 开始累加，超时即触发）• `1`：倒计时（从 `timeout` 倒数至 0 触发） *注：对 `on_timer` 回调行为无影响，仅影响内部显示或调试用途*
`repeat`	number	**重复次数** • `0`：无限重复（周期性触发） • `N > 0`：触发 N 次后自动停止

6.2.stop_timer(timer_id)

停止脚本定时器函数，`stop_timer(timer_id)` 是 HMI 系统提供的**定时器管理接口**，用于**立即停止指定 ID 的软件定时器**，防止其继续触发 `on_timer` 回调。

📊 **参数说明**

参数	类型	说明
`timer_id`	number	**定时器 ID** • 取值范围：0 ~ 31 • 必须与之前通过 `start_timer` 启动的定时器 ID 一致

6.3.on_timer(timer_id)

定时器超时回调函数，`on_timer(timer_id)` 是 HMI 系统中**定时器事件的统一回调入口**。当通过 `start_timer` 启动的任意定时器超时时，系统会自动调用此函数，并传入对应的 `timer_id`，开发者可据此执行对于逻辑。

📊 **参数说明**

参数	类型	说明

参数	类型	说明
`timer_id`	number	**触发超时的定时器 ID** • 取值范围：0 ~ 31 • 用于区分多个并发定时器

💡 示例

如初始化开启1s超时，无限循环的定时器，则每秒输出“hello lua”

```
function on_timer(timer_id)
    if timer_id == 0
    then
        print('hello lua!!!')
    end
end

function on_init()
    start_timer(0, 1000, 0, 0)
end

>hello lua!!!
>hello lua!!!
>...
```

6.4.get_timer_value(timer_id)

获取定时器当前计时值函数，`get_timer_value(timer_id)` 是 HMI 系统提供的**定时器状态查询接口**，用于**实时获取指定软件定时器的当前计时值（单位：毫秒）**。该函数支持顺计时和倒计时两种模式下的读取

📊 参数说明

项目	说明
参数	
`timer_id`	**定时器 ID** • 取值范围：0 ~ 31 • 必须对应一个已启动的定时器
返回值	
`time_ms`	**当前计时值（毫秒）**

7.串口

7.1.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
`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。

7.2.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、FX3U等 等）占用，则主串口 `on_uart_recv` **不会被调用。**

📊 **参数说明**

参数	类型	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）**

7.3.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}`

7.4.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

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、DCBU、XGUS、FX3U 等），系统重新接管主串口通信。

7.5.uart_rxsize(ch)

读取串口接收缓冲区数据字节数函数，`uart_rxsize(ch)` 是 HMI 系统提供的**串口通信状态查询接口**，用于**实时获取指定串口通道接收缓冲区中待读取的数据字节数。**该函数专为**主动式串口协议解析**（即脚本轮询读取），用来实现自定义协议功能，详细参考VisualHMI - 自定义协议(主动)(点击跳转)

📊 **参数说明**

项目	类型	说明
参数		
`ch`	number	串口号 • 通常取值：`0`，具体编号依 HMI 硬件配置而定
返回值		
`size`	number	接收缓冲区中当前可用的字节数 • 返回 `N` 表示有N个字节 • 返回 `0` 表示无新数据

7.6.uart_recv(ch, size)

串口主动接收指定字节数函数，`uart_recv(ch, size)` 是 HMI 系统提供的**串口数据读取接口**，用于从指定串口的接收缓冲区中**主动读取指定数量的字节。**该函数需配合 `uart_rxsize` 使用，构成"查询-读取"主动通信模型，适用于自定义协议解。详细参考VisualHMI - 自定义协议(主动)(点击跳转)

📊 **参数说明**

项目	类型	说明
参数		
`ch`	number	**串口号** • 通常：`0`， 依硬件配置而定
`size`	number	**期望读取的字节数** • 必须 ≥1 • 建议 ≤ `uart_rxsize(ch)` 返回值
返回值		

项目	类型	说明
`data`	table	接收到的原始字节数据 • **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}`

7.7.uart_rxclear(ch)

清空串口接收缓冲区函数，`uart_rxclear(ch)` 是 HMI 系统提供的**串口接收缓冲区管理接口**，用于**立即清空指定串口通道（`ch`）的接收缓冲区中所有未读取的数据。**该函数在自定义串口通信协议开发中至关重要，常用于**通信初始化、错误恢复、协议同步**等场景，避免残留数据干扰后续帧解析。详细参考 VisualHMI - 自定义协议(主动)(点击跳转)

📊 **参数说明**

参数	类型	说明
`ch`	number	**串口号** • 通常：`0`，具体编号依 HMI 硬件配置而定

7.8.calc_crc16(data)

Modbus CRC-16 校验计算函数，`calc_crc16(data)` 是 HMI 系统内置的**标准 Modbus CRC-16（CRC-16-MODBUS）校验算法实现**，用于对指定数据块计算 16 位循环冗余校验码。该函数广泛应用于 **Modbus RTU协议**的帧完整性验证或组帧，确保串口通信中数据的可靠性。

📊 **参数说明**

项目	类型	说明
参数		
`data`	table	**待校验的原始数据** • 以 Lua 表形式传递，**下标从 1 开始。** 示例：`{0xAA, 0x55, 0x01, 0x02}`
返回值		
`crc`	number	**计算得到的 CRC-16 校验值** • 范围：`0 ～ 65535` • 低字节在前，高字节在后（符合 Modbus 小端序）

8.音视频

8.1.play_sound(filename)

播放音频文件函数，`play_sound(filename)` 是 HMI 系统提供的**音频播放接口**，用于在运行时**触发指定音频文件的播放。**该功能常用于操作反馈、告警提示、语音引导等场景，提升人机交互的直观性与用户体验。

📊 **参数说明**

参数	类型	说明
`filename`	string	音频文件路径： 绝对路径

```
--屏内路径播放：
M：play_sound("3:/sound/welcome.wav")

--SD路径播放：
M：play_sound("1:/welcome.wav")

--U盘路径播放：
M：play_sound("2:/welcome.wav")
```

8.2.stop_sound()

停止音频播放函数，`stop_sound()` 是 HMI 系统提供的**音频控制接口**，用于**立即终止当前正在播放的音频文件**

8.3.pause_sound()

暂停音频播放函数,`pause_sound()` 是 HMI 系统提供的**音频播放控制接口**，用于**临时暂停当前正在播放的音频**（由 `play_sound` 启动），并在后续通过 `resume_sound()` 恢复播放。

8.4.resume_sound()

恢复音频播放函数，`resume_sound()` 是 HMI 系统提供的**音频播放控制接口**，用于**恢复由** `pause_sound()` **暂停的音频播放**，从暂停时的位置继续播放。该函数必须与 `pause_sound()` 配套使用。

8.5.play_video(file,left,top,width,height)

视频播放函数，`play_video` 是 HMI 系统提供的**嵌入式视频播放接口**，用于在指定屏幕区域播放符合硬件解码能力的 MP4 视频。该功能适用于**操作引导、产品演示、安全须知播放**等场景，通过富媒体提升人机交互体验

📊 **参数说明**

参数	类型	说明

参数	类型	说明
`file`	string	**视频文件路径** • 绝对路
`left`	number	显示区域左上角 X 坐标（像素）
`top`	number	显示区域左上角 Y 坐标（像素）
`width`	number	视频显示宽度（像素）
`height`	number	视频显示高度（像素）

⚙ 视频格式

项目	要求说明
格式	MP4（`.mp4`）
视频编码	H.264
音频编码	MP3 或 AAC
最大分辨率	1280 × 768 像素（超过将无法播放或严重卡顿）
最大帧率	30 fps（帧/秒）
最大码率	1400 kbps（约 1.4 Mbps）

💡示例

```
--屏内路径播放：
M: play_video("3:/video/welcome.mp4", 0, 0 800, 480)

--SD路径播放：
M: play_video("1:/welcome.mp4", 0, 0 800, 480)

--U盘路径播放：
M: play_video("2:/welcome.mp4", 0, 0 800, 480)
```

8.6.pause_video()

暂停视频播放函数，`pause_video()` 是 HMI 系统提供的**视频播放控制接口**，用于**临时暂停当前正在播放的视频**（由 `play_video` 启动），并在后续通过 `resume_video()` 从暂停位置继续播放。该功能适用于需要**用户交互控制、临时中断演示、同步操作步骤**等场景

8.7.resume_video()

恢复视频播放函数，`resume_video()` 是 HMI 系统提供的**视频播放控制接口**，用于**恢复由** `pause_video()` **暂停的视频播放**，从暂停时的精确帧位置继续播放。该函数必须与 `pause_video()` 配套使用，共同实现视频的"暂停/继续"交互逻辑，适用于操作引导、教学演示、安全确认等需要用户临时中断并恢复的场景

8.8.stop_video()

停止视频播放函数，`stop_video()` 是 HMI 系统提供的**视频播放终止接口**，用于**立即停止当前正在播放或已暂停的视频**，释放相关解码资源，并清除屏幕上的视频画面。

8.9.on_video_notify(msg，v1, v2)

视频播放状态回调函数，`on_video_notify(msg, v1, v2)` 是 HMI 系统提供的**视频播放事件回调接口**，用于在视频播放过程中**播放状态与进度信息。**

📊 **参数说明**

参数	类型	说明
`msg`	number	**播放状态标识** • `1`：播放中（周期性通知） • `0`：播放完毕（仅触发一次）
`v1`	number	**当前已播放时长（单位：秒，s）**
`v2`	number	**视频总时长（单位：秒，s）**

8.10.av_init(show,channel,left,top,width,height)

AV 输入初始化函数，`av_init` 是 HMI 系统提供的**模拟输入（AV Input）初始化接口**，用于配置并显示来自外部摄像头或视频源（如工业相机、后视摄像头、CVBS 信号）的实时视频流。该功能广泛应用于**设备监控、视觉引导、安全预览**等需要接入外部模拟视频信号的工业场景。

📊 **参数说明**

参数	类型	说明
`show`	number	**显示控制** • `0`：初始化但隐藏视频画面（后台预览） • `1`：初始化并立即显示视频
`channel`	number	**视频输入通道号** • 通常：`0` = AV1，`1` = AV2 • 具体编号依 HMI 硬件定义
`left`	number	**显示区域左上角 X 坐标（像素）**
`top`	number	**显示区域左上角 Y 坐标（像素）**

参数	类型	说明
`width`	number	**视频显示宽度（像素）**
`height`	number	**视频显示高度（像素）**

8.11.av_get_status()

获取 AV 播放状态函数，`av_get_status()` 是 HMI 系统提供的**AV 输入状态查询接口**，用于检测当前 AV 通道是否**正在正常接收并显示外部模拟信号。**该函数返回一个状态码，明确指示 AV 输入的实时工作状态，适用于摄像头连接检测、信号有效性判断、故障告警等场景。

📊 **参数说明**

参数	类型	说明
返回值	number	• `1`：播放中 —— 已检测到有效的 AV 视频信号（如 PAL/NTSC 同步信号），正常显示 • `0`：未播放 / 无信号 —— 无有效输入信号、摄像头未连接、或未初始化 AV 通道

8.12.av_select(ch)

切换 AV 通道函数，`av_select(ch)` 是 HMI 系统提供的**AV 视频输入通道切换接口**，用于在**多路 AV 输入源之间动态切换**（如通道1 ↔ 通道2）。该功能适用于配备**双路模拟输入**（如双摄像头、前后视系统）的工业或车载 HMI 设备，实现多视角实时监控与切换。

⚠️ **硬件前提**：

- HMI 必须支持**至少 2 路 AV 输入通道**；

📊 **参数说明**

参数	类型	说明
`ch`	number	目标通道编号 • `0`：切换到 通道1（AV1） • `1`：切换到 通道2（AV2）

8.13.av_show(show)

显示/隐藏 AV 播放函数，`av_show(show)` 是 HMI 系统提供的**AV 显示控制接口**，用于**动态显示或隐藏当前已初始化的 AV 视频画面**，而**不中断底层视频信号采集。**该功能适用于需要临时遮蔽视频、切换界面焦点、节省渲染资源等场景，实现“后台持续预览 + 前台按需显示”的灵活控制。

📊 **参数说明**

参数	类型	说明

参数	类型	说明
`show`	number	**显示控制指令** • `0`：隐藏 AV 视频画面（视频仍在后台采集） • `1`：显示 AV 视频画面（恢复到 `av_init` 指定区域）

8.14.video_shoot(filepath, width, height)

视频/AV 画面截图函数。`video_shoot(filepath, width, height)` 是 HMI 系统提供的**实时视频画面捕获接口**，用于将当前**AV 输入或视频播放窗口的内容**截取并保存为 **BMP 格式图片。**

⚠️ **重要限制说明**：视频控件截图截图，特定固件支持。AV输入控件，默认支持

📊 **参数说明**

参数	类型	说明
`filepath`	string	保存路径与文件名 •绝对路径：`"1:/av_20260204.bmp"`
`width`	number	截图宽度（像素）• 建议 ≤ 视频/AV 显示区域宽度 • 超出部分可能填充黑边或裁剪
`height`	number	截图高度（像素）• 建议 ≤ 视频/AV 显示区域高度

8.15.set_color_key(Min_Color,Max_Color,Match)

视频图层色键透明控制函数（M 系列专用），`set_color_key` 是 **M 系列 HMI 专用**的视频底层渲染控制接口，用于解决**视频图层始终置顶导致无法叠加文字/图形**的问题。通过配置一个 **RGB 颜色范围 + 比较规则**，系统会将视频画面中**匹配该颜色范围的像素设为透明**，从而允许下层控件（如按钮、文本、图形）透过视频显示，实现"视频+UI 叠加"的混合渲染效果。

📊 **参数说明**

参数	类型	说明
`Min_Color`	number	**24 位 RGB 颜色最小值** • 格式：`0x00RRGGBB`（高位字节为 0） • 示例：`0x00BFBFBF` 表示 R=0xBF, G=0xBF, B=0xBF（浅灰色）
`Max_Color`	number	**24 位 RGB 颜色最大值** • 格式：`0x00RRGGBB` • 示例：`0x00C8C8C8` 表示 R=0xC8, G=0xC8, B=0xC8
`Match`	number	**6 位颜色分量比较掩码** • 二进制格式：`RRGGGBB`（实际为 6 位：`R[1:0]G[1:0]B[1:0]`） • 每 2 位控制一个通道： • `10` = 启用该通道范围比较 • `00` = 忽略该通道 • 示例：`0x2A` = `0b101010` → R/G/B 均启用

> 注：set_color_key(Min_Color,Max_Color,Match)，此API接口函数必须要放在on_init()系统初始化函数中使用，默认在初始时配置的属性

9.文件系统操作

9.1.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。

```
function on_init()
    dofile('test.lua')
end
```

9.2.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

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**

9.3.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

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）

💡 **示例**

```
function on_list_dir(path, filename, type, fsize)

    local msg = ''
    if type == 0 --文件夹
    then
    else --文件
         msg = path..'/'..file_name --文件路径
    end
end
```

9.4.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

💡 **示例**

```
--打开文件读（没有文件创建文件）：
file_open(path, 0x00)

--打开文件覆盖写（没有文件创建文件）：
file_open(path, 0x01)

--打开文件末尾写（没有文件创建文件）：
file_open(path, 0x02)
```

9.5.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
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

9.6.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

💡 示例

```
local file_fd = file_open(path, 0x00)
local filesize = file_size()
```

9.7.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）

9.8.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。

💡 **示例**

```
local tb = file_read(2048)
```

9.9.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
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。

💡 **示例**

```
local ret = file_write(write_byte_Tb) -- 1 ≤ data 数组大小 ≤ 2048
return ret
```

9.10.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）

💡 **示例**

```
file_delete('1:/1.txt')
```

9.11.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"`
`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

💡 **示例**

```
file_copy('1:/1.txt', '1:/1_copy.txt') --从SD卡目录下，复制1.txt为1.copy.txt
```

9.12.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

💡 示例

```
function on_copy_file_process(status,filesize,transfersize)

    local progress        = 0

    if status == 0 -- 复制失败
    then

    elseif status == 1 --复制中
    then
        local prg = ((transfersize*100) // filesize) --计算百分比进度

    elseif status == 2 -- 复制成功
    then
    end

    refresh_screen()-- 刷新界面
end
```

9.13.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

💡扩展：删除目录

在嵌HMI 的文件操作系统中，**不支持递归删除目录**，仅仅支持**删除非空目录**，必须按照下面流程操作：

🚥 流程

步骤	操作	描述

步骤	操作	描述
1. 目录遍历	调用 `list_dir(path)`	通过系统回调 `on_list_dir()` 获取其直接子项（包括文件与子目录）。
2. 内容清理	递归删除子项	对每个子项： • 若为**普通文件**（`type == 1`），直接调用 `file_delete()`； • 若为**子目录**（`type == 0`），**递归执行本流程**
3. 目录删除	调用 `file_delete(dir_path)`	待目录变为空（无任何子项）后， 调用 `file_delete()` **删除空目录元数据**）

⚠️ 关键约束

- `file_delete()` **仅支持删除空目录**：
  若目录非空，调用 `file_delete()` 将**静默失败或返回错误**（具体行为取决于 HMI 平台实现，但绝不会自动递归删除）。
- 所有删除操作均为**物理删除**，不可恢复。

💡扩展：重命名目录

HMI 系统（如基于 FAT/FAT32 的 VisualHMI 平台）中，**文件系统层不支持对目录项直接重命名操作。**因此，实现"目录重命名""需通过 **模拟重命名** 的多步流程完成。

> **由于底层文件系统不支持 `rename()` 系统调用对目录的直接重命名，必须通过"创建新目录 → 递归迁移内容 → 删除原目录"的三阶段事务性操作来实现逻辑重命名。**

流程

阶段	操作	技术描述
1. 目标目录创建	调用 `mkdir(new_path)`	在目标路径创建一个**空的新目录**，作为重命名后的容器。
2. 内容递归迁移	遍历原目录 + 文件复制	对原目录执行**遍历**： • 遇到**子目录**：递归创建对应新子目录，并继续迁移其内容； • 遇到**普通文件**：调用 `file_copy(src, dst)` 将文件从 `old_path/...` 复制至 `new_path/...`。
3. 原目录清理与删除	递归删除原目录	待所有内容成功迁移后，对原目录执行**递归删除** •先清空其全部子项（文件与子目录） •再调用 `file_delete(old_path)` 移除**空目录。**

10.CAN 接口

10.1.canbus_open(index,baudrate,listen_mode,loop_back)

CAN 总线接口打开函数，`canbus_open(index, baudrate, listen_mode, loop_back)` 是 HMI 系统提供的**CAN 总线初始化接口**，用于配置并启用指定的 CAN 控制器。该函数完成波特率设置、工作模式选择及环回测试使能等关键参数配置，是进行后续 CAN 报文收发（如 `canbus_send()`、`on_canbus_rx()`）的前提。

📊 **参数说明**

参数	类型	说明
`index`	number	**CAN 通道索引号** • 默认：`0`,可配置0或1，根据具体硬件配置
`baudrate`	number	**通信波特率（单位：Kbps）** • 仅支持以下标准值：`125`（125 kbps），`250`（250 kbps），`500`（500 kbps），`1000`（1 Mbps）
`listen_mode`	number	**监听模式** • `0`：正常模式（可收发） • `1`：只读模式（禁止发送，仅监听总线，不参与 ACK 应答） • 常用于总线监控或避免干扰
`loop_back`	number	**自发自收模式** • `0`：正常模式（报文发送至总线） • `1`：环回模式

10.2.canbus_close(index)

CAN 总线接口关闭函数，`canbus_close(index)` 是 HMI 系统提供的**CAN 总线资源释放接口**，用于**关闭并禁用指定索引的 CAN 控制器**，停止其收发功能并释放相关硬件或驱动资源。

📊 **参数说明**

参数	类型	说明
`index`	number	CAN 通道索引号 • 有效值：`0` 或 `1` • 对应硬件 CAN0 / CAN1

10.3.canbus_write(index,identifier,dlc,rtr,ide,data)

CAN 报文发送函数，`canbus_write(index, identifier, dlc, rtr, ide, data)` 是 HMI 系统提供的**CAN 总线报文发送接口**，用于向指定 CAN 通道发送一帧标准或扩展格式的 CAN 报文。该函数支持数据帧与远程帧，是实现**设备控制、状态请求、数据上报**等 CAN 通信功能的核心操作。

📊 **参数说明**

参数	类型	说明

参数	类型	说明
`index`	number	**CAN 通道索引号** • 有效值：`0` 或 `1` • 必须对应已打开的 CAN 通道
`identifier`	number	**报文标识符（ID）** • 标准帧（`ide = 0`）：范围 `0x000 ～ 0x7FF`（11 位） • 扩展帧（`ide = 1`）：范围 `0x00000000 ～ 0x1FFFFFFF`（29 位）
`dlc`	number	**数据长度码** • 有效值：`0 ～ 8` • 表示实际有效数据字节数
`rtr`	number	**远程帧标志** • `0`：数据帧（携带 `data`） • `1`：远程帧（不携带数据，请求其他节点发送同 ID 报文） • 远程帧时 `data` 参数可忽略或传空表
`ide`	number	**帧格式标志** • `0`：标准帧（11 位 ID） • `1`：扩展帧（29 位 ID）
`data`	table	**数据字段（仅数据帧有效）** • 元素为 `0～255` 的整数（字节值） • 下标从 `1` 开始连续排列 • 表长度必须等于 `dlc`，且 ≤ 8 字节 • 示例：`{0x12, 0x34, 0x56, 0x78}`

10.4.on_canbus_recv(index,identifier,dlc,rtr,ide,data)

CAN 报文接收回调函数，`on_canbus_recv(index, identifier, dlc, rtr, ide, data)` 是 HMI 系统在**成功接收到有效 CAN 报文后自动触发的异步回调函数。**

📊 **参数说明**

参数	类型	说明
`index`	number	**接收通道索引号** • 值为 `0` 或 `1` • 表示报文来自哪个物理 CAN 接口
`identifier`	number	**报文标识符（ID）** • 标准帧（`ide = 0`）：11 位，范围 `0x000 ～ 0x7FF` • 扩展帧（`ide = 1`）：29 位，范围 `0x00000000 ～ 0x1FFFFFFF`
`dlc`	number	**数据长度码（DLC）** • 表示有效数据字节数，范围 `0 ～ 8` • 即使实际数据字段为 8 字节，`dlc` 仍反映真实有效长度

参数	类型	说明
`rtr`	number	**远程帧标志** • `0`：数据帧（携带有效 `data`） • `1`：远程帧（请求数据，`data` 内容无效）
`ide`	number	**帧格式标志** • `0`：标准帧（11 位 ID） • `1`：扩展帧（29 位 ID）
`data`	table	**数据字段（仅数据帧有效）** • 元素为 `0～255` 的整数（字节值） • 下标从 `1` 开始连续排列 • 表长度等于 `dlc`（≤ 8）• 远程帧时内容未定义，不应使用

💡 示例

以下是例子为楼宇对讲、电梯联动、智能门禁等应用，实现**地面站PLC**与**楼层HMI**分机之间的双向通信控制

PLC	CAN_ID	长度 (byte)	名称	类型	描述
地面站	`0x201`	2	楼呼分机地址	U16	楼呼分机地址，对应的楼呼分机需要应答。
		2	预留	U16	保留字段。
		2	楼层左门控制字节	U16	楼层左门控制字节，按位解析，如下： 第0位：打开对应层门（置位）； 第1位：关闭对应层门（置位）； 第3位：卷帘门置位，电磁锁复位 其它：预留
		2	预留	U16	保留字段。
HMI（应答帧）	0x200	2	应答对应的 CAN_ID	U16	应答对应的设备 ID（如 `0x201` 或 `0x202`）。
		2	楼呼分机地址	U16	对应的楼呼分机地址，需应答。
		2	按数据位解析	U16	按数据位解析，如下： 第0位：取消上行(置位)； 第1位：取消下行(置位)； 其它：暂不定义。

PLC	CAN_ID	长度(byte)	名称	类型	描述
		1	左门数据位解析	U8	左门数据按位解析，如下： 第0位：上限位触发(置位)，未触发(复位)； 第1位：下限位触发(置位)，未触发(复位)； 其它：暂不定义。
		1	右门数据位解析	U8	右门数据按位解析，如下： 第0位：上限位触发(置位)，未触发(复位)； 第1位：下限位触发(置位)，未触发(复位)； 其它：暂不定义。

> 适用范围：VisualHMI - Can接口的产品
>
> 例程下载链接：ViusalHMI - can(点击下载)

11.设置窗口

API函数中的screen、control参数均表示为目标画面ID、目标组态控件ID。控件ID需要用户编号，默认为0，需要设置非0才生效

11.1.set_screen(screen)

画面切换函数，`set_screen(screen)` 是 HMI 系统提供的**画面管理核心接口**，用于**立即切换当前显示的画面**至指定 ID 的目标画面。

📊 **参数说明**

参数	类型	说明
`screen`	number	**目标画面**

11.2.get_screen()

获取当前画面ID函数，`get_screen()` 是 HMI（人机交互）系统提供的一个用于**查询当前显示画面**的接口。

📊 **参数说明**

项目	说明
成功返回	**当前画面的 ID**

11.3.show_dialog(screen, x, y, alpha)

弹出对话框，调用 `show_dialog(screen, x, y, alpha)` ，在当前画面之上**叠加显示指定 ID 的对话框画面**，或**关闭已显示的对话框**。该函数支持自定义位置与透明度。

📊 **参数说明**

参数	类型	说明
`screen`	number	**对话框画面 ID** • `≥ 0`：要显示的对话框画面 ID（必须已在工程中定义） • `-1`：关闭当前对话框（无论其 ID 为何）
`x`	number	**对话框左上角 X 坐标（像素）**
`y`	number	**对话框左上角 Y 坐标（像素）**
`alpha`	number	**对话框整体透明度（百分比）** • 取值范围：`0 ～ 100` • `100`：完全不透明 • `50`：半透明 • `0`：完全透明

💡 **示例**

```
show_dialog(0, 184, 108, 50) --显示对话框，坐标（184,105），50%的透明度
show_dialog(-1, 184, 108, 50) ---1，表示关闭对话框，后面参数任意
```

11.4.wgt_set_pos(screen,control,x,y,w,h)

设置控件的位置，`wgt_set_pos()` 是一个用来控件的位置的接口。通过提供目标画面 ID、控件 ID 以及期望的坐标和尺寸信息，用户可以精确地控制 UI 元素在屏幕上的布局。

📊 **参数说明**

参数	类型	说明
`screen`	number	**目标画面 ID**
`control`	number	**目标控件 ID**
`x`	number	**左上角 X 坐标（像素）**
`y`	number	**左上角 Y 坐标（像素）**
`w`	number	**原控件宽度（像素）**
`h`	number	**原控件高度（像素）**

💡 **示例**

```
--设置画面0控件ID10显示在（476,70）位置，大小为120,36
wgt_set_pos(0,10,476,70,120,36)
```

11.5.wgt_set_fcolor(screen,control, color)

设置控件前景色，`wgt_set_fcolor()` 是一个用来调整 控件前景色的接口。

📊 **参数说明**

参数	类型	说明
`screen`	number	**目标画面 ID**
`control`	number	**目标控件 ID**
`color`	number	**颜色值（RGB565格式）** • 使用16位表示颜色，其中红色占5位，绿色占6位，蓝色占5位。 • 示例：红色可以表示为 `0xF800`

💡 **示例**

```
--设置画面0控件ID10的前景色
wgt_set_fcolor(0,10,0xF800) --红色
wgt_set_fcolor(0,10,0x07E0) --绿色
wgt_set_fcolor(0,10,0xFFE0) --黄色
```

11.6.wgt_set_bcolor(screen,control, color)

设置控件的背景色，`wgt_set_bcolor()` 是一个用来调整 控件背景色的接口。

📊 **参数说明**

参数	类型	说明
`screen`	number	**目标画面 ID**
`control`	number	**目标控件 ID**
`color`	number	**颜色值（RGB565格式）** • 使用16位表示颜色，其中红色占5位，绿色占6位，蓝色占5位。 • 示例：红色可以表示为 `0xF800`

💡 **示例**

```
--设置画面0控件ID10的背景色
wgt_set_bcolor(0,10,0xF800) --红色
wgt_set_bcolor(0,10,0x07E0) --绿色
wgt_set_bcolor(0,10,0xFFE0) --黄色
```

11.7.wgt_set_param(screen,control, param,value)

控件属性设置扩展接口，`wgt_set_param(screen, control, param, value)` 是 HMI 系统提供的**通用控件属性配置接口**，用于动态设置特定控件的扩展属性。该函数通过 **参数标识码（`param`）** 与 **数值（`value`）** 的组合，支持对不同控件类型进行精细化控制，是实现**数据记录翻页、曲线坐标轴调整、GIF动画控制**等高级功能的关键通道。

📊 **参数说明**

参数	类型	说明
`screen`	number	**目标画面 ID**
`control`	number	**目标控件 ID**
`param`	number	**属性标识码** • 决定要设置的具体功能 • 已知有效值见下表
`value`	number	**属性值** • 含义由 `param` 决定

已定义 `param` 标识码说明

`param` (Hex)	名称	适用控件	`value` 含义说明
`0x28`	数据记录控件通道翻页	数据记录控件	通道页码索引（从 0 开始）

`param` (Hex)	名称	适用控件	`value` 含义说明
`0x31`	曲线控件 Y 轴最小值	历史曲线控件	Y 轴显示范围下限
`0x32`	曲线控件 Y 轴最大值	历史曲线控件	Y 轴显示范围上限
`0x33`	GIF 播放速度	GIF 控件	播放帧间隔（单位：毫秒），值越小播放越快

💡 示例

```
--设置画面0，控件ID10的曲线控件最大值、最小值
wgt_set_param(0,10, 0x32,100) --最大值100
wgt_set_param(0,10, 0x31,0)   --最小值10
wgt_set_param(0,11, 0x28,-400)   --表格向左滚动400个像素
wgt_set_param(0,11, 0x28,400)   --表格向右滚动400个像素
wgt_set_param(0,12, 0x33,100)   --帧间隔为100ms
```

11.8.on_wgt_event(screen_id,widget_id,event,value)

控件事件回调函数，是 HMI 系统在**用户与控件发生交互或控件状态发生变化时自动触发的全局事件回调函数。**

📊 参数说明

参数	类型	说明
`screen_id`	number	**当前画面 ID**
`widget_id`	number	**触发事件的控件 ID，保证 ≠ 0**
`event`	number	**事件类型码**
`value`	number	**事件关联数据**

11.9.on_wgt_text_color(screen,control)

控件文字颜色动态回调函数，`on_wgt_text_color(screen, control)` 是 HMI 系统提供的**控件文字颜色动态设置接口**，用于根据特定条件批量设置控件的文字颜色。

📊 参数说明

参数名	类型	说明
`screen`	number	**当前画面 ID：** 用于区分多页面中的同名控件

参数名	类型	说明
`control`	number	**目标控件 ID：** •仅当 `control ≠ 0` 时触发回调

📊 **返回值说明**

返回值	类型	说明
`color`	number	**RGB565 格式的 16 位颜色值**

11.10.refresh_screen()

立即刷新画面

12.简易数据库

⚠ 简易数据库 Flash 地址规划注意事项

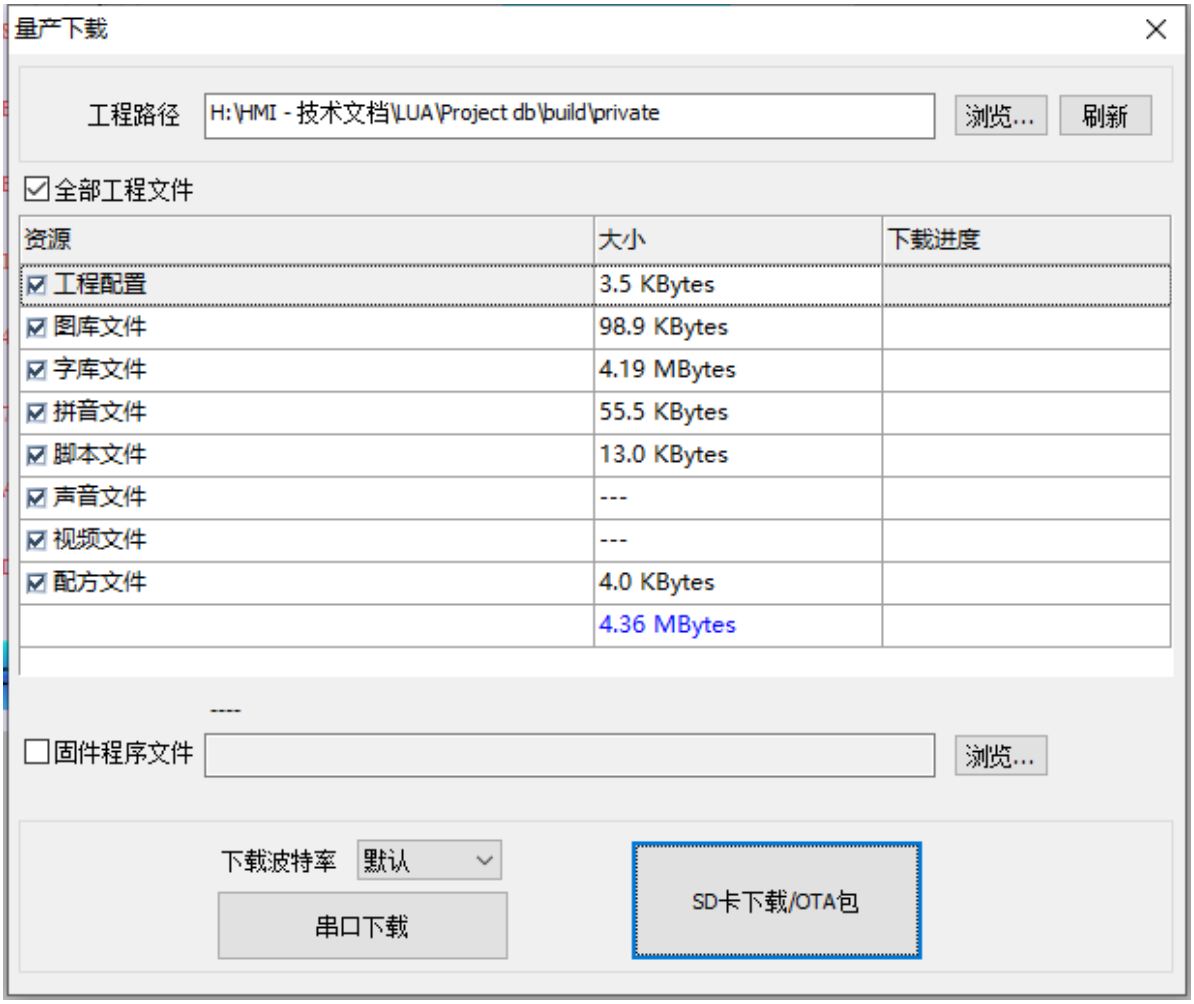
ViusualHMI 的数据库是以字符串写入flash，一般屏幕默认为16M。本简易数据库直接操作裸 Flash 存储区域，**不具备文件系统隔离机制**，因此其 `flashaddr` 起始地址必须严格避开系统其他关键功能所占用的存储区块，避免数据覆盖、程序异常或设备变砖。特别需注意以下四类高风险区域：

⚠. **工程资源存储区**

⚠. **资料采样块存储区**

⚠. **操作记录块存储区**

⚠. **OTA 升级备份区域。**



⚠本章节所有数据库操作，都封装在`db.lua`中，均采用**句柄驱动**（Handle-Based）的设计范式。调用 `db.open → easydb_open` 成功后，将返回一个非零整数作为数据库操作的唯一有效句柄。该句柄应被**声明为全局变量**（如 `g_db_handle`），并在整个应用生命周期内由主逻辑脚本统一持有。

所有对简易数据库的读写操作（包括添加、查询、修改、删除、清空等）**必须显式地将该句柄作为首个关键形参传入**，模块内部不维护任何全局状态或隐式上下文。此设计确保了接口的无状态性、线程安全性及多实例兼容性。

⚠ **重要使用说明：理解设计意图，避免机械复制**，本模块章节旨在**传递清晰的架构思想与工程实践原则**，而非提供可直接粘贴复用的"代码片段"。请用户在集成或参考时，务必注意。

12.1.easydb_open(flashaddr,createIfNotExist,dataRowSize,maxCount)

数据库打开函数，`easydb_open()` 是 HMI 系统提供的**嵌入式轻量级数据库接口**，用于在内部 Flash 存储器中**打开或创建一个结构化数据表**。该数据库以**固定长度记录**方式存储，适用于**参数配置、事件日志、运行记录**等掉电存数据读写场景。

📊 **参数说明**

参数	类型	说明
`flashaddr`	number	**Flash 起始地址（字节对齐）**

参数	类型	说明
`createIfNotExist`	number	**创建策略标志** • `0`：仅打开已有数据库，若不存在则失败 • `1`：若数据库不存在，则按后续参数创建新库
`dataRowSize`	number	**单条记录大小（字节）** • 必须 ≥ 1
`maxCount`	number	**最大记录数量** • 决定数据库总占用空间 = `dataRowSize × maxCount`

💡 示例

```
--- 打开数据库，返回操作句柄。
-- @param flashaddr        (number) Flash 存储起始地址（单位：字节），例如
10*1024*1024
-- @param createIfNotExist (number) 是否在数据库不存在时自动创建（0: 否, 1: 是）
-- @param dataRowSize      (number) 单条记录最大字节数（建议 ≤ 1024）
-- @param maxCount         (number) 数据库最多可存储的记录条数（建议 ≤ 10000）
-- @return handle          (number) 数据库句柄；非0表示成功，0表示失败
function db.open(flashaddr, createIfNotExist, dataRowSize, maxCount)
    local handle = easydb_open(flashaddr, createIfNotExist, dataRowSize,
maxCount)
    return handle
end
```

12.2.easydb_close(db)

数据库关闭函数，`easydb_close(db)` 是 HMI 系统提供的**嵌入式数据库资源释放接口**，用于**关闭已打开的 Flash 数据库**，释放其占用的内存上下文或驱动句柄

📊 参数说明

参数	类型	说明
`db`	number	**数据库句柄** • 由 `easydb_open()` 成功调用时返回数据

💡 示例

```
--- 关闭数据库，释放资源。
-- @param handle (number) 由 db.open() 返回的数据库句柄
function db.close(handle)
    if handle and handle ~= 0 then easydb_close(handle) end
end
```

12.3.easydb_get_count(db)

获取数据库当前有效记录数，`easydb_get_count(db)` 是 HMI 系统提供的**轻量级 Flash 数据库查询接口**，用于**获取指定数据库中当前已写入的有效记录数量。**

📊 参数说明

参数	类型	说明
`db`	number	**有效的数据库句柄** • 必须是由 `easydb_open()` 成功返回的句柄

💡 示例

```
--- 获取当前数据库中的记录总数。
-- @param handle (number) 数据库句柄
-- @return count (number) 记录数量；若句柄无效，返回 -1
function db.getCnt(handle)
    if not handle or handle == 0 then return -1 end
    return easydb_get_count(handle)
end
```

12.4.easydb_get(db,idx)

读取数据库指定记录，`easydb_get(db, idx)` 是 HMI 系统提供的**轻量级 Flash 数据库读取接口**，用于**从已打开的数据库中读取指定索引位置的完整数据记录。**

📊 参数说明

参数	类型	说明
`db`	number	**有效的数据库句柄** • 由 `easydb_open()` 成功返回的句柄
`idx`	number	**记录索引（行号）** • 从 `0` 开始计数 • 有效范围：`0 ≤ idx < 当前记录总数`

📊 返回值说明

项目	说明
成功返回	字符串
失败返回	`nil`

💡 示例

```lua
--- 根据索引读取一条记录（索引从 0 开始）。
-- @param handle (number) 数据库句柄
-- @param idx    (number) 记录索引（0 ~ getCnt()-1）
-- @return ret   (number) 0 表示成功，-1 表示句柄无效，-2 表示索引越界
function db.get(handle, idx)

    if not handle or handle == 0 then return nil end

    local count = easydb_get_count(handle)
    if idx < 0 or idx >= count then return -2 end -- 索引越界

    return 0, easydb_get(handle, idx)
end
```

12.5.easydb_add(db,dataset)

向数据库追加新记录，`easydb_add(db, dataset)` 是 HMI 系统提供的**轻量级 Flash 数据库写入接口**，用于**在指定数据库末尾追加一条新记录。**

📊 **参数说明**

参数	类型	说明
`db`	number	**有效的数据库句柄** • 由 `easydb_open()` 成功返回的句柄
`dataset`	string	**要写入的数据内容**

💡 **示例**

```lua
--- 向数据库末尾追加一条新记录。
-- @param handle (number) 数据库句柄
-- @param msg    (string) 要写入的字符串数据（长度 ≤ dataRowSize
function db.add(handle, msg)
    if not handle or handle == 0 then return -1 end
    return easydb_add(handle, msg)
end
```

12.6.easydb_update(db,idx,dataset)

更新数据库指定记录，`easydb_update(db, idx, dataset)` 是 HMI 系统提供的**轻量级 Flash 数据库更新接口**，用于**修改已存在记录的指定索引位置的数据内容。**

📊 **参数说明**

参数	类型	说明
`db`	number	**有效的数据库句柄** • 由 `easydb_open()` 成功返回的句柄

参数	类型	说明
`idx`	number	**要更新的记录索引（行号）** • 从 `0` 开始计数 • 有效范围：`0 ≤ idx < easydb_get_count(db)`
`dataset`	string	**新的数据内容** • 长度必须严格等于创建时指定的 `dataRowSize`

💡 示例

```lua
--- 修改指定索引的记录内容（索引从 0 开始）。
-- @param handle (number) 数据库句柄
-- @param idx    (number) 要修改的记录索引（0 ~ getCnt()-1）
-- @param msg    (string) 新的数据内容
-- @return ret   (number) 0 表示成功，-1 表示句柄无效，-2 表示索引越界
function db.modify(handle, idx, msg)

    if not handle or handle == 0 then return -1 end

    local count = easydb_get_count(handle)
    if idx < 0 or idx >= count then return -2 end -- 索引越界

    print(string.format("db.modify(%d, '%s').....", idx, msg))
    return 0, easydb_update(handle, idx, msg)
end
```

12.7.easydb_del(db,idx)

删除数据库指定记录，`easydb_del(db, idx)` 是 HMI 系统提供的**轻量级 Flash 数据库删除接口**，用于**删除指定索引位置的数据记录**

📊 参数说明

参数	类型	说明
`db`	number	**有效的数据库句柄** • 由 `easydb_open()` 成功返回的句柄
`idx`	number	**要删除的记录索引（行号）** • 从 `0` 开始计数 • 有效范围：`0 ≤ idx < easydb_get_count(db)`

💡 示例

```lua
--- 删除指定索引的记录（索引从 0 开始）。
-- @param handle (number) 数据库句柄
-- @param idx    (number) 要删除的记录索引（0 ~ getCnt()-1）
-- @return ret   (number) 0 表示成功，-1 表示句柄无效，-2 表示索引越界
function db.del(handle, idx)

    if not handle or handle == 0 then return -1 end

    local count = easydb_get_count(handle)
    if idx < 0 or idx >= count then return -2 end -- 索引越界

    return 0, easydb_delete(handle, idx)
end
```

12.8.easydb_clear(db)

清空数据库所有记录，`easydb_clear(db)` 是 HMI 系统提供的**轻量级 Flash 数据库清空接口**，用于**一次性删除数据库中的所有有效记录**，将数据库恢复至初始空状态。

📊 **参数说明**

参数	类型	说明
`db`	number	**有效的数据库句柄** • 由 `easydb_open()` 成功返回的句柄

💡 **示例**

```lua
--- 清空数据库中所有记录。
-- @param handle (number) 数据库句柄
-- @return ret   (number) 0 表示成功，-1 表示失败
function db.clear(handle)
    if not handle or handle == 0 then return -1 end
    return easydb_clear(handle)
end
```

13.GPIO控制

GPIO控制，需要看硬件是否引出IO引脚，以HMI48272KM043 硬件为例，PD20、PD21，IO定义如下:

16进制的高八位表示IO组，低八位表示引脚

```lua
LED1 = 0x0314
LED2 = 0x0315
```

13.1.gpio_set_in (pin)

GPIO 引脚设为输入模式，`gpio_set_in(pin)` 是 HMI 或嵌入式系统提供的 **GPIO配置接口**，用于将指定的物理引脚（PIN）**配置为输入模式。**

📊 **参数说明**

参数	类型	说明
`pin`	number	目标 GPIO 引脚编号 • 必须为平台支持的有效引脚，如PD20=0x0314

💡 **示例**

```
gpio_set_in(LED1)--LED1 = 0x0314
gpio_set_in(LED2)--LED2 = 0x0315
```

13.2.gpio_set_out (pin)

GPIO 引脚设为输出模式，`gpio_set_out(pin)` 是 HMI 或嵌入式系统提供的 **GPIO配置接口**，用于将指定的物理引脚（PIN）**配置为输出模式**

📊 **参数说明**

参数	类型	说明
`pin`	number	目标 GPIO 引脚编号 • 必须为平台支持的有效引脚，如PD20=0x0314

💡 **示例**

```
gpio_set_out (LED1)--LED1 = 0x0314
gpio_set_out (LED2)--LED2 = 0x0315
```

13.3.gpio_set_value (pin,value)

设置 GPIO 引脚逻辑输出，`gpio_set_value(pin, value)` 是嵌入式 HMI 系统提供的 **GPIO 输出控制接口**，用于**设置已配置为"输出模式"的引脚的电平状态**

📊 **参数说明**

参数	类型	说明
`pin`	number	目标 GPIO 引脚编号 • 必须为平台支持的有效引脚，如PD20=0x0314
`value`	number	目标逻辑 • `0` or 1

💡 **示例**

```
gpio_set_value(LED1,0) --IO输出逻辑0
gpio_set_value(LED2,1) --IO输出逻辑1
```

13.4.gpio_get_value (pin)

读取 GPIO 输入引脚电平，`pio_get_value(pin)` 是嵌入式 HMI 系统提供的 **GPIO 输入状态读取接口**，用于**获取已配置为"输入模式"的引脚当前的逻辑电平**

📊 参数说明

参数	类型	说明
`pin`	number	目标 GPIO 引脚编号 • 必须为平台支持的有效引脚，如PD20=0x0314

📊 返回说明

项目	说明
成功返回	`0` 或 `1`

💡 示例

```
local val = gpio_get_value (LED1)

> val = 1 --逻辑1
> val = 0 --逻辑0
```

13.4.set_rotate_decoder(index, pinA, pinB, ktype)

旋钮编码器初始化配置，`set_rotate_decoder()` 是 HMI 系统提供的**增量式旋转编码器（Rotary Encoder）硬件接口配置函数**，用于**绑定物理引脚、指定编码器类型并分配通道索引。**配置后需配合 `start_rotate_decoder()` 启动解码服务，系统将自动解析 A/B 相位信号。

📊 参数说明

参数	类型	说明
`index`	number	**编码器通道索引** • 范围：`0 ～ 3`（支持最多 4 路独立编码器）
`pinA`	number	*编码器 A 相 GPIO 引脚编号 ** • 必须为有效输入引脚
`pinB`	number	**编码器 B 相 GPIO 引脚编号**
`ktype`	number	**编码器类型标识** • `0`：标准 2 相增量编码器（如 EC11，每 detent 产生 4 脉冲） • 其他值：保留扩展（如带按键编码器需额外配置）

13.5.start_rotate_decoder()

启动旋钮编码器解码， `start_rotate_decoder()` **是 HMI 系统提供的**旋钮编码器启动接口，**调用** `set_rotate_decoder()` **将启动** 配置好的指定通道编码器。系统将开始监听该编码器 A/B 相引脚的电平变化

13.6.set_sw_pwm(index, pin, mode, freq, duty)[定制]

软件 PWM 输出配置接口，`set_sw_pwm()` 是 HMI 或嵌入式系统提供的 **软件模拟 PWM（脉宽调制）输出接口**，用于在**任意 GPIO 引脚上生成指定频率和占空比的方波信号**

📊 **参数说明**

参数	类型	说明
`index`	number	**PWM 通道索引** • 范围通常为 `0 ～ N-1`（如 0、1，视硬件而定） • 用于区分多路独立 PWM 输出
`pin`	number	**目标 GPIO 引脚编号**
`mode`	number	**工作模式** • `0`：强制输出低电平（关闭 PWM，引脚=0） • `1`：强制输出高电平（关闭 PWM，引脚=1） • `2`：启用 PWM 模式（按 `freq`/`duty` 生成波形）
`freq`	number	**PWM 频率（Hz）** • 单位：赫兹（Hz） • 有效范围依平台而定（如 1 Hz ～ 10 kHz） • 频率越高，CPU 开销越大
`duty`	number	占空比（%） • 范围：`0 ～ 100` • `0` = 始终低电平，`100` = 始终高电平

14.OTA升级

详细参考VisualHMI - OTA升级(主动)(点击跳转)

📌OTA 升级 Flash 使用规范

以 HMI80480M070 为例

1. **Flash 总容量**
    - 标准品 Flash 大小：**128 Mbit = 16 MB**（字节）。
2. **系统固件占用**
    - M 系列系统文件（底层 OS + 驱动）占用：**1 MB**（DH 系列为 2 MB，本例为 M 系列，按 1 MB 计）。
3. **当前工程文件大小**

- 下载文件 private 大小：**4.38 MB。**

4. **已用 Flash 空间总计**
   - 已用 = 系统（1 MB） + 工程（4.38 MB） = **5.38 MB。**
   - 剩余可用空间 = 16 MB − 5.38 MB = **≈10.62 MB。**
5. **OTA 升级起始地址要求**
   - **必须从 Flash 的后半部分开始**，即使**剩余可用空间 =10.62MB**，也要从≥ **8 MB 地址处开始**（16 MB ÷ 2 = 8 MB）。
   - 目的：避免新旧工程在 Flash 中交叉覆盖，确保升级过程中当前系统仍可正常运行。
6. **OTA 区域需连续且空闲**
   - OTA 写入区域必须是**连续的未使用地址段**；
   - 若用户已通过"块地址"功能占用了部分高地址区（如 10~12 MB）
7. **OTA 文件大小限制**
   - OTA 升级包（新工程）大小 **Flash 一半**，某升级包大小为 ≈10.62 MB，**不可升级**，；
   - OTA 升级包（新工程）大小=5MB，块地址占用8M~12M， **不可升级**；

14.1.ota_init(md5, filesize, addr)

OTA 升级初始化接口，`ota_init(md5, filesize, addr)` 是 HMI 或嵌入式系统提供的 **OTA升级初始化函数**，用于**配置本次升级的目标参数**，包括固件大小、写入地址。调用成功后，系统将准备接收新固件数据并写入指定 Flash 区域，为后续的 `ota_write()` 和 `ota_finish()` 流程奠定基础。

📊 **参数说明**

参数	类型	说明
`md5`	string	**MD5 校验字符串（固定值）** ： `"0123456789abcdef"`
`filesize`	number	**待升级固件文件大小（字节）** • 单位：Byte • 必须与实际传输的 `.bin` 文件大小严格一致 • 系统据此预分配空间并验证最终写入长度
`addr`	number	**OTA 固件写入起始地址（Flash 物理地址）** • 必须 ≥ Flash 总容量的一半（例如：2MB Flash → `addr ≥ 0x100000`）

14.2.ota_write(writeTb)

OTA 固件数据写入接口，`ota_write(writeTb)` 是 HMI 或嵌入式系统提供的 **OTA升级数据写入函数**，用于将**分块的固件数据写入预先通过 `ota_init()` 配置的 Flash 地址区域。**

📊 **参数说明**

参数	类型	说明

参数	类型	说明
`writeTb`	table	**待写入的字节数据块** • Lua 表形式，下标从 `1` 开始 • 有效数据长度 ≤ 2048 字节 • 若 `< 2048`，系统自动在末尾补 `0x00` 至 2048 字节

14.3.ota_check_upgrade(state)

OTA 升级校验与解压执行接口，`ta_check_upgrade(state)` 是 HMI 或嵌入式系统提供的 **OTA 升级最终确认与执行函数。**在用户通过 `ota_init()` 和 `ota_write()` 完整传输新固件（`ota.bin`）后，调用此函数将**触发系统对已写入 Flash 的固件数据进行完整性校验、解压缩，并完成升级流程。**

📊 **参数说明**

参数	类型	说明
`state`	number	**升级控制状态码** • 必须传入 `1`，表示“开始校验并执行升级”

14.4.ota_destory()

清除 OTA 升级残留数据，`ota_destroy()` 是 HMI 或嵌入式系统提供的 **OTA升级清理接口**，用于**擦除已写入 Flash 的 OTA 固件数据**

14.5.on_ota_progress(status, value)

OTA 升级过程状态回调函数，`on_ota_progress(status, value)` 是 HMI 系统在执行 `ota_check_upgrade(1)` 后**自动触发的全局回调函数**，用于向应用层实时反馈 OTA 固件的**校验与解压进度及结果。**开发者可通过实现此函数，在 UI 上显示进度条、提示信息或处理升级失败逻辑

📊 **参数说明**

`status`	**含义**	`value` **说明**
`1`	校验过程开始	固定为 `0`，表示校验阶段已启动
`2`	校验结果	• `0`： 校验失败（固件损坏、MD5/CRC 不匹配等） • `1`： 校验成功
`3`	解压过程	**解压进度百分比** 范围 `0 ～ 100`（如 `value=50` 表示解压完成 50%）
`4`	解压结果	• `0`： 解压失败（数据格式错误、空间不足等） • `1`： 解压成功 → 即将重启

15.其他

15.1.get_platform()

获取设备平台与芯片信息，`get_platform()` 是 HMI 或嵌入式系统提供的**设备识别接口**，用于**获取当前运行设备的硬件平台标识字符串。**

```
local device = get_platform()
```

15.2.get_version()

获取固件版本号，`get_version()` 是 HMI 或嵌入式系统提供的**固件版本查询接口**，用于**获取当前设备运行的固件版本标识**

```
local ver= get_version()
```

15.3.get_device_uuid()

获取设备唯一标识符，`get_device_uuid()` 是 系统提供的**设备身份认证接口**，用于**获取当前设备的全局唯一标识符（UUID）**。该字符串由设备出厂时固化或首次启动时生成，具有**不可变、不重复**

📊 **返回参数**

类型	说明
string	20 位字符串

💡 **示例**

```
uuid = get_device_uuid()
```

15.4.reboot()

屏幕复位

15.5.feed_dog()

看门狗定时器喂狗接口，`feed_dog()` 是嵌入式 HMI 系统提供的**看门狗函数**，用于**在长时间任务执行过程中重置看门狗计时器**，防止系统因“假死”或“卡死”而被强制复位。当某段操作（如文件处理、网络通信、复杂计算）**预计耗时超过 5 秒**时，必须周期性调用此函数以告知系统“程序仍在正常运行”。

💡 **示例**

```
for i = 0, 10000 --某循环体大
do
    feed_dog()
    --do user code
end
```

15.6.delay_ms(ms)

🔴**毫秒级延时函数**，`delay_ms(ms)` 是 HMI 或嵌入式系统提供的**阻塞式延时接口**，用于让当前任务**暂停执行指定的毫秒数。**

📊 **参数说明**

参数	类型	说明
`ms`	number	延时时间（毫秒） • 推荐范围：`1 ～ 5000` • `ms > 5000`：🔴高风险！可能导致系统重启,屏幕卡顿

15.7.set_run_cycle(cycle)

设置 `on_run` 回调执行周期，`set_run_cycle(cycle)` 是 HMI 系统提供的**主循环回调频率控制接口**，用于**设定全局函数 `on_run(screen_id)` 的自动调用间隔。**该函数通常在 `on_init()` 中调用一次，以配置后台任务的 `on_run` 的执行。

⚠️ **核心说明**:

- 单位为 **毫秒（ms）**，最小值依平台而定（通常 ≥ 10 ms）；
- **严禁在 `on_run()` 内部调用**（可能导致调度器死锁或未定义行为）；

📊 **参数说明**

参数	类型	说明
`cycle`	number	`on_run` 回调周期（毫秒）

💡 **示例**

```
function on_init()
    set_run_cycle(1000) --on_run(screen)  1s回调一次
end
```

15.8.update_system()

`update_system()` 是 HMI 系统提供的**关键系统参数存储函数**，用于**将当前系统参数（如音量、语言、背光、配方设置等）立即写入非Flash**。调用后**掉电不丢失**。

💡 示例：加载系统语言

```
function on_init()
    set_uint16(VT_LW, 0x011B, (1<<1)) -- 选择需要加载的掩码，bit1,多语言
    set_uint16(VT_LW, 0x011A, 0x5502) -- 加载选中的系统参数
    update_system()
end

function on_update(slave,vtype,addr)
    if vtype == VT_LW
    then
        if addr == 0x119
        then
            set_uint16(VT_LW, 0x011B, (1<<1)) -- 选择需要加载的掩码，bit1,多语言
            set_uint16(VT_LW, 0x011A, 0x5501) -- 保存选中的系统参数
            update_system()
        end
    end
end
```

💡 示例：配方设置

```
for i = 0, 41
do
    feed_dog()
    set_uint16(VT_LW, 0x1000, i)
    update_system()--立刻加载

    local recoveryTb = {}
    for j = 1, 32
    do
        recoveryTb[j] = 0
    end

    set_array(VT_LW, 0x1001, recoveryTb)
    set_uint16(VT_LW, 0x1100, 2)
    update_system()--立刻写入
end
```

15.9.get_date_time ()

获取当前系统日期与时间，`get_date_time()` 是 HMI 或嵌入式系统提供的**实时时钟（RTC）查询接口**，用于**获取当前日期与时间。**该函数返回完整的年、月、日、时、分、秒及星期信息。

📊 返回参数

变量	类型	范围与说明

变量	类型	范围与说明
`year`	number	年份，如 `2026`
`month`	number	月份，`1 ～ 12`
`day`	number	日期，`1 ～ 31`（依月份自动适配）
`hour`	number	小时，`0 ～ 23`（24 小时制）
`min`	number	分钟，`0 ～ 59`
`sec`	number	秒，`0 ～ 59`
`week`	number	星期

💡 示例

```
local year,mon,day,hour,min,sec,week = get_date_time()
```

15.10.set_date_time (year,mon,day,hour,min,sec)

设置系统日期与时间，`set_date_time()` 是 HMI 或嵌入式系统提供的**实时时钟（RTC）写入接口**，用于**手动设置本地系统时间。**

📊 参数说明

参数	类型	范围与要求
`year`	number	年份，建议范围：`2000 ～ 2099`
`mon`	number	月份，`1 ～ 12`
`day`	number	日期，`1 ～ 31`，需符合实际日历（如 2023-02-29 无效）
`hour`	number	小时，`0 ～ 23`（24 小时制）
`min`	number	分钟，`0 ～ 59`
`sec`	number	秒，`0 ～ 59`

💡 示例

```
set_date_time(2023,8,8,8,8,8)
```

15.11.make_datetime(timestamp)

时间戳转本地日期时间， HMI 或嵌入式系统提供的**时间戳解析函数**，用于将 **Unix 时间戳（秒）转换为年、月、日、时、分、秒。**

📊 返回参数

变量	类型	范围与说明
`year`	number	年份，如 `2026`
`month`	number	月份，`1 ～ 12`
`day`	number	日期，`1 ～ 31`
`hour`	number	小时，`0 ～ 23`（24 小时制）
`min`	number	分钟，`0 ～ 59`
`sec`	number	秒，`0 ～ 59`

💡 **示例**

```
local year,mon,day,hour,min,sec = make_datetime(timestamp)
```

15.12.make_timestamp(year,mon,day,hour,min,sec)

本地日期时间转 Unix 时间戳，`ake_timestamp()` 是 HMI 或嵌入式系统提供的**本地时间编码函数**，用于将 **年、月、日、时、分、秒组成的日期时间转换为 Unix 时间戳（秒）**。

📊 **参数说明**

参数	类型	范围与要求
`year`	number	年份，2000 ~ 2099
`mon`	number	月份，`1 ～ 12`
`day`	number	日期，`1 ～ 31`
`hour`	number	小时，`0 ～ 23`（24 小时制）
`min`	number	分钟，`0 ～ 59`
`sec`	number	秒，`0 ～ 59`

💡 **示例**

```
local timestamp = make_timestamp(year,mon,day,hour,min,sec)
```

15.13.set_pwd(level, pwd，len)

设置用户等级密码，`set_pwd(level, pwd, len)` 是 HMI 系统提供的**多级用户密码管理接口**，用于**指定安全等级（0～7）设置数字密码。**该函数支持**前导零处理**，确保密码位数固定（如 4 位密码 `0123` 不被存储为 `123`。

📊 **参数说明**

参数	类型	必填	说明
`level`	number	是	**用户等级** • 范围：`0 ～ 7` • `0` 通常为最高权限（管理员）
`pwd`	number	是	**数字密码** • 仅支持数字（0-9） • 示例：`123`，`0`（表示全零密码）
`len`	number	否	**密码总位数（用于前导零补全）** • 若省略，则按 `pwd` 实际位数存储 • 若指定（如 `4`），则密码将视为 `0123`（当 `pwd=123`

💡 **示例：**无前导零，设置修改1级密码并存储，如下所示：

```
set_pwd(0,123)
set_uint16(VT_LW, 0x011B, (1<<4))-- 选择用户密码系统参数
set_uint16(VT_LW, 0x011A, 0x5501)-- 保存密码
```

💡 **示例：**前导零，设置修改1级密码并存储，如下所示：

```
set_pwd(0,123,4)
set_uint16(VT_LW, 0x011B, (1<<4)) --选择用户密码系统参数
set_uint16(VT_LW, 0x011A, 0x5501) --保存密码
```

💡 **示例：**上电加载安全等级密码

```
function on_init()
    set_uint16(VT_LW, 0x011B, (1<<4)) --选择用户密码系统参数
    set_uint16(VT_LW, 0x011A, 0x5502) --加载密码
    update_system()
end
```

15.14.set_stage_pwd(level,pwd)

设置分期使用密码，`set_stage_pwd(level, pwd)` 是 HMI 系统提供的**设备分期使用密码管理接口**，用于为指定*分期等级（0～9）设置数字解锁密码*

📊 **参数说明**

参数	类型	必填	说明
`level`	number	是	分期等级 • 范围：`0 ～ 9`
`pwd`	number	是	数字密码 • 仅支持纯数字（0-9）

💡 **示例：**设置修改1级密码并存储，如下所示：

```
set_stage_pwd(0,1234)
set_uint16(VT_LW, 0x011B, (1<<5))-- 选择分期密码系统参数
set_uint16(VT_LW, 0x011A, 0x5501)-- 保存密码
```

💡 **示例：**上电加载分期密码

```
function on_init()
    set_uint16(VT_LW, 0x011B, (1<<5)) --选择分期密码系统参数
    set_uint16(VT_LW, 0x011A, 0x5502) --加载密码
    update_system()
end
```

15.15.md5(text, key)

带密钥的 MD5 消息认证，`md5(text, key)` 是 HMI 或嵌入式系统提供的**带密钥的MD5函数**，用于生成 **基于 MD5 算法的消息认证码。**该函数通过结合**目标消息**与**预共享密钥**，生成一个 **32 位十六进制字符串**.

📊 **参数说明**

参数	类型	必填	说明
`text`	string	是	**待认证的原始消息** • 可为任意字符串
`key`	string	是	**预共享密钥（Secret Key）** • 双方必须使用相同密钥