H7-TOOL 开发工具

Lua 小程序开发手册



武汉安富莱电子有限公司

1. 概述

1.1. 功能简介

H7-TOOL的Lua小程序功能是一个非常重要的功能。用户编写Lua小程序并下载到H7-TOOL内部，之后就可以脱离PC机执行。H7-TOOL的脱机烧录功能就是结合Lua小程序实现了复杂的业务逻辑。

实现各种小程序，不再限制Flash里面已经下载的程序，就跟手机安装APP差不多，所以在H7-TOOL里面被广泛使用，支持在线调试运行，支持离线运行。

支持USB，以太网和WiFi的在线控制运行和离线控制运行。且局域网和外网也可以控制。

应用场合：

- 生产线批量设置产品参数、成品检测
- 研发人员验证电路板模块功能
- 测试人员测试产品
- 脱机产品检测，日志记录等

主要特点：

- 体积小，节约内存
- 简化人工操作流程，提高工作效率
- 接口功能强大，GPIO输入输出、串口，SPI,I2C，ADC、DAC、RS485、RS232、CAN、RJ45 等均支持
  - 系统控制函数
  - 信号发生器
  - 4-20mA输入输出
  - GUI显示
  - 文件系统
  - 10 路GPIO控制
  - 4 路PWM
  - 脱机烧录控制
  - I2C主机从机
  - SPI主机从机
  - 电压电流采集
  - 串口/232/485/Modbus，支持离线日志
  - CAN/CANFD/CANopen/UDS/J1939
  - NTC热敏电阻
  - 微型数控电源
  - IO扩展
  - 传感器扩展
  - 脉冲测量
  - SWD调试接口函数，读写各种类型数据，内存和寄存器等
  - 支持串口，SPI，I2C和CAN的bootloader脱机更新
  - 扩展驱动AD7606，ADS1256，MCP3421，8 路继电器和 5 路DS18B20
- 提供很多已在生产中批量使用的lua程序范例

1.2. Lua 脚本

Lua是一种强大、高效、轻量级、可嵌入的脚本语言。Lua程序无需编译，直接通过Lua虚拟机解释执行。Lua和Python相比较，Lua占用资源小，执行速度快。Lua的语法非常接近c语言，因此学习成本较低。

Lua的官网：http://www.lua.org/。Lua从 1993 年发布，至今已有 30 年，仍然在迭代更新。

Lua 5.3 于 2015 年 1 月 12 日发布。它的主要新功能是支持 64 位整数、整数bit操作。最后版本 Lua 5.3.6 于 2020 年 9 月 25 日发布。

Lua 5.4 于 2020 年 6 月 29 日发布。它的主要新功能是新一代垃圾回收模式。当前版本是 Lua 5.4.4，于 2022 月 1 月 26 日发布(当前最新版是 5.4.7，发布于 2024.6.25)。其改进如下：

- 新一代的垃圾回收模式
- 等待关闭的变量
- 用户数据可以有多个用户值
- 新方法实现随机数 math.random
- 告警系统
- 关于函数参数和返回的调试信息
- 'for' 循环有新的整数语义
- string.gmatch 可选'init'参数
- 新函数 'lua_resetthread' 、'coroutine.close'
- string-to-number 强制转换被移到string library
- allocation 函数允许失败，当缩小内存块时
- string.format'只支持'%p'
- utf8 library 允许codepoints 最大到 2^31

H7-TOOL 内嵌的Lua版本为 5.3。

Lua参考手册 5.4（官方英文版）：http://www.lua.org/manual/5.4/manual.html

Lua参考手册 5.3（中文版）：https://www.runoob.com/manual/lua53doc/manual.html

2. 使用方法

2.1. 编辑 Lua 脚本

Lua程序编辑器可以用记事本、UltrEdit、NotePad++、VS Code。推荐使用VS Code，它是免费的，支持中文，并自带Lua语法着色，启动速度较快。

下载地址：https://code.visualstudio.com/

简单的lua语句可以直接在H7-TOOL的PC软件中书写。

2.2. 调试 Lua 脚本

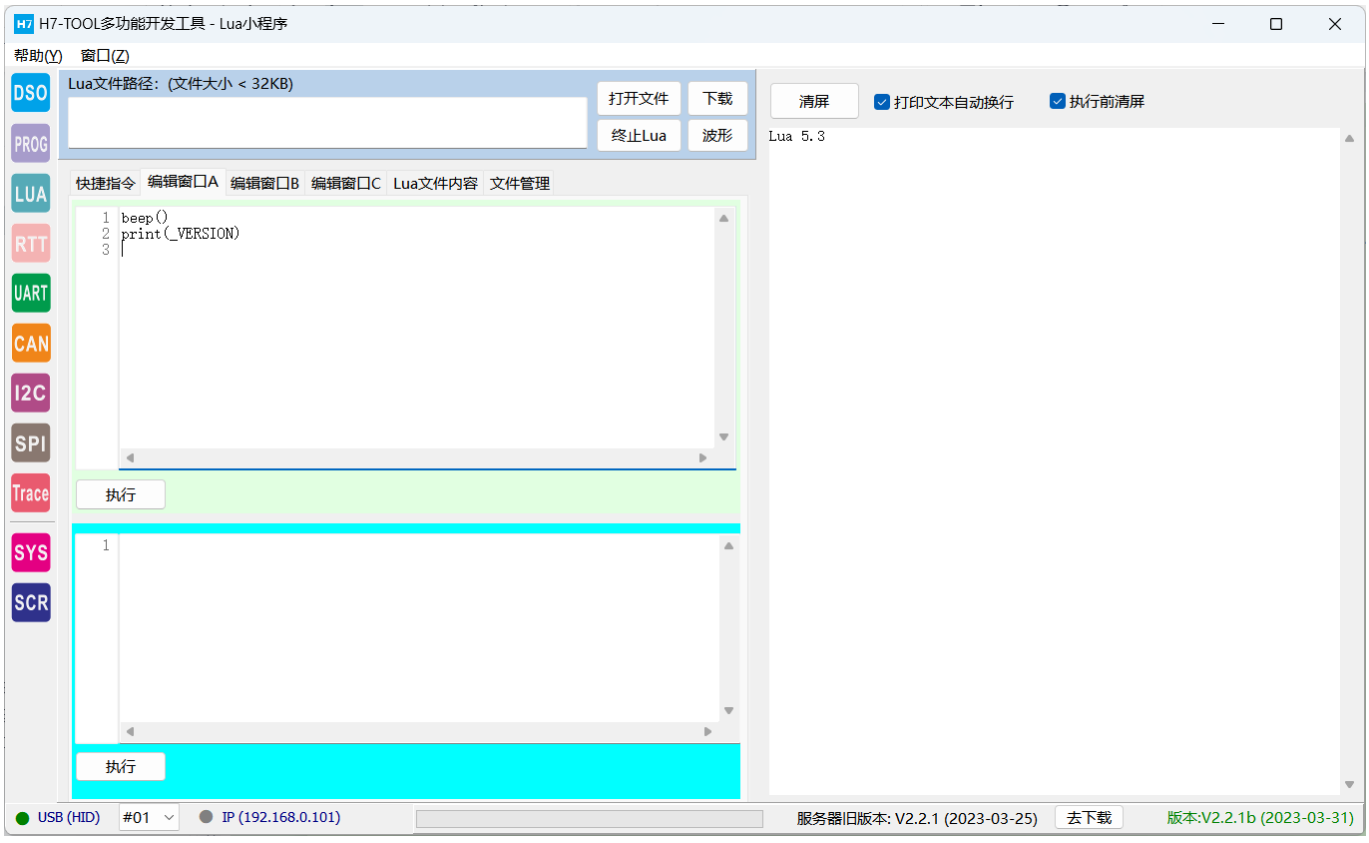
2.2.1. 联机调试步骤

1、用USB线连接H7-TOOL到PC机的USB接口。
2、启动 h7toolPC.exe
3、点左侧Lua图标，切换到小程序界面-编辑窗口A
4、在编辑框中输入Lua语句：

```
beep()  --H7-TOOL 蜂鸣器鸣叫
print(_VERSION)  --打印 Lua 解释器版本
```

--表示Lua语言的行注释。beep()是H7-TOOL扩展的API函数。print()是Lua标准库中的API函数。

5、点执行按钮
6、在右侧窗口可以观察到执行结果。



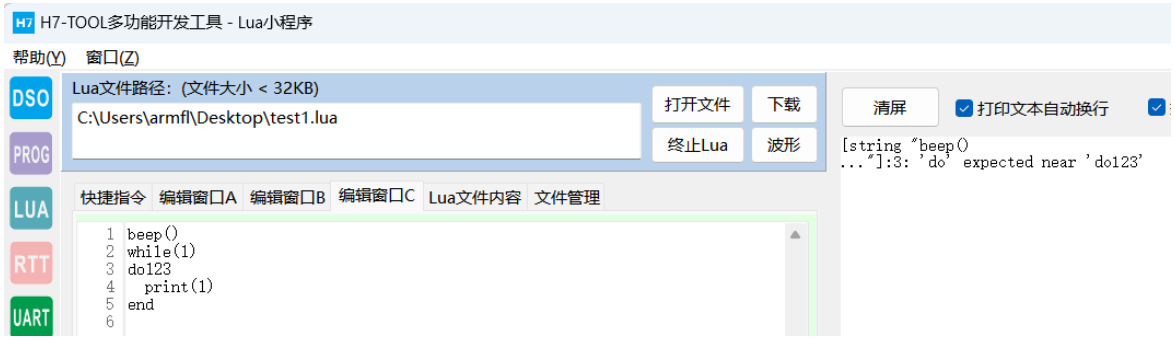
编辑窗口A、B、C共有 6 个文本编辑框，这 6 个窗口的文本会自动保存。

在这些窗口点“执行”，不会初始化Lua虚拟机。执行的代码都会加入到Lua虚拟机，不过不用担心内存溢出问题，因为Lua虚拟机能够自动回收内存。

万一装入一个超大的文本程序或者存在严重bug的程序，Lua无法分配内存，它会打印内存分配错误。

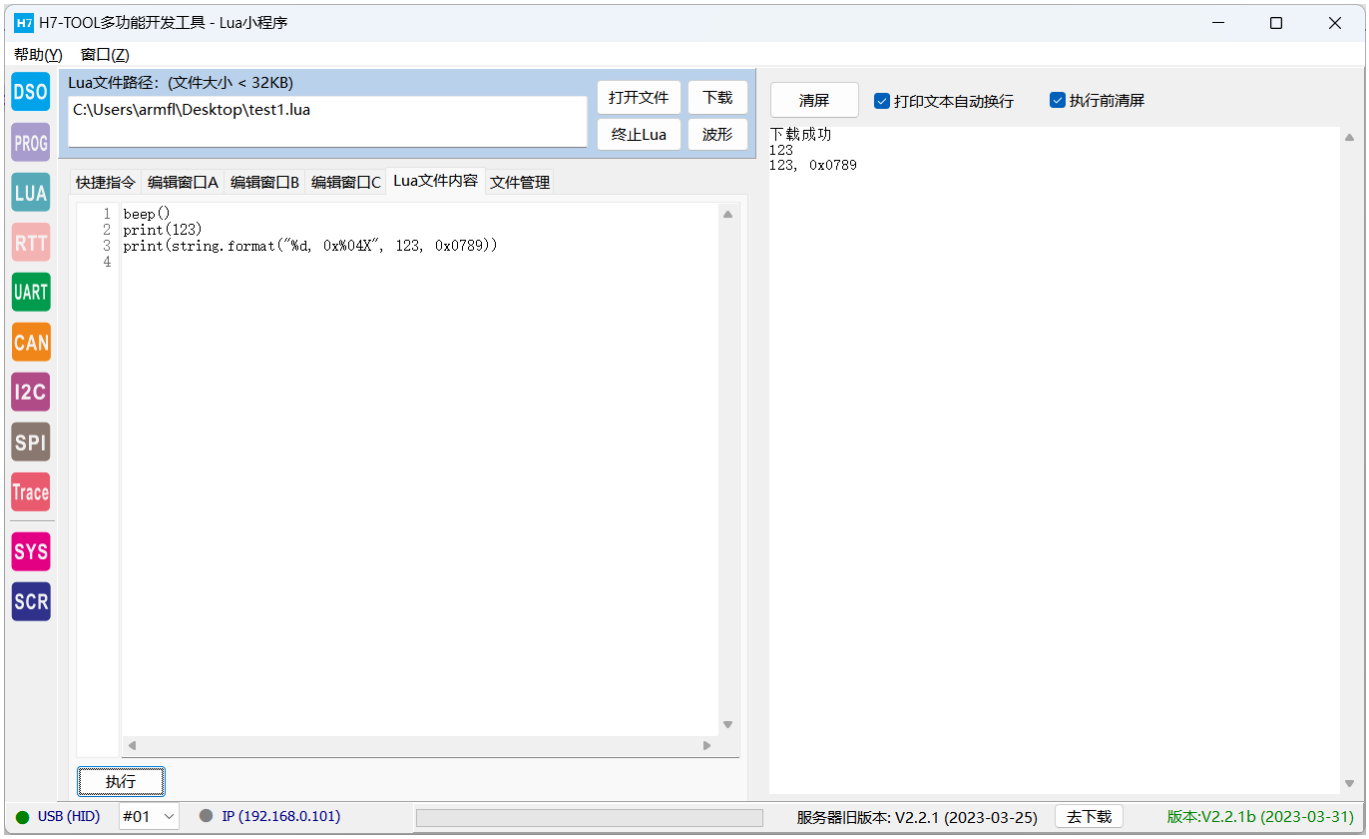
如果你执行的是一个死循环程序，要终止程序，可以点“终止”按钮或者长按H7-TOOL上C键。有些Lua循环程序无等待地持续打印信息，会导致PC软件假死，此时长按C键终止Lua程序总是有效的。

有语法错误的程序在执行时会报错，调试信息会输出到右侧窗口。比如：



2.2.2. 装载 Lua 文件调试

1、点打开文件按钮，弹窗后选择后缀为.lua的文件。
2、点下载按钮，此时Lua文件内容会装入到“Lua文件内容”窗口中，并自动执行一次。



test1.lua 文件内容:

```
beep()
print(123)
print(string.format("%d, 0x%04X", 123, 0x0789))
```

string.format 是Lua string库中的API函数。格式化字符的控制和C语言基本上是一致的。

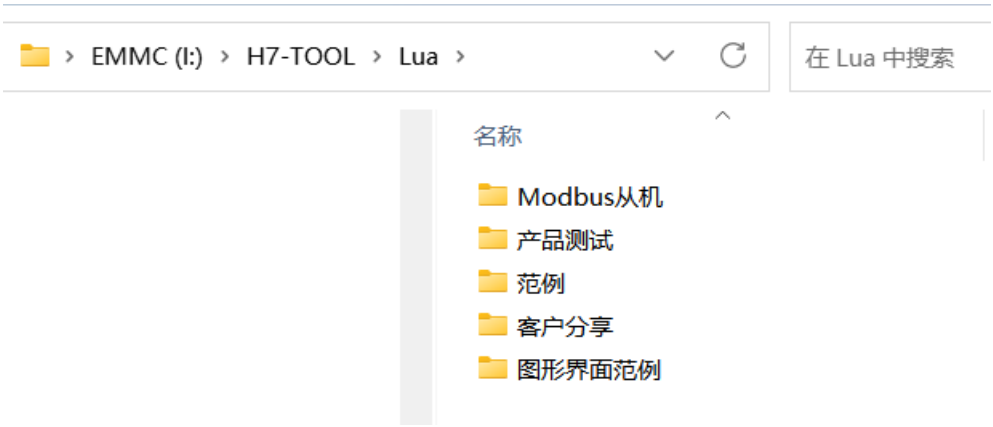
点“下载”和“执行”按钮时，会删除旧的Lua虚拟机，并创建一个新的Lua虚拟机。

如果这个Lua文件将用于脱机执行，推荐让H7-TOOL进入Lua小程序界面，可以同步观测H7-TOOL小屏的显示效果。

2.3. 传送到 H7-TOOL

Lua小程序文件（后缀为.lua）必须手动放到H7-TOOL内置的EMMC磁盘。

让H7-TOOL进入虚拟U盘模式后，可以看到目标目录：\H7-TOOL\Lua



方法 1：进入 U 盘模式，直接复制粘贴



进入系统设置页面，点击“进入U盘模式”按钮，之后Windows会发现一个EMMC的移动磁盘。请将lua文件放置到H7-TOOL\Lua目录。客户可以自由创建子目录，也可以删除多余文件。

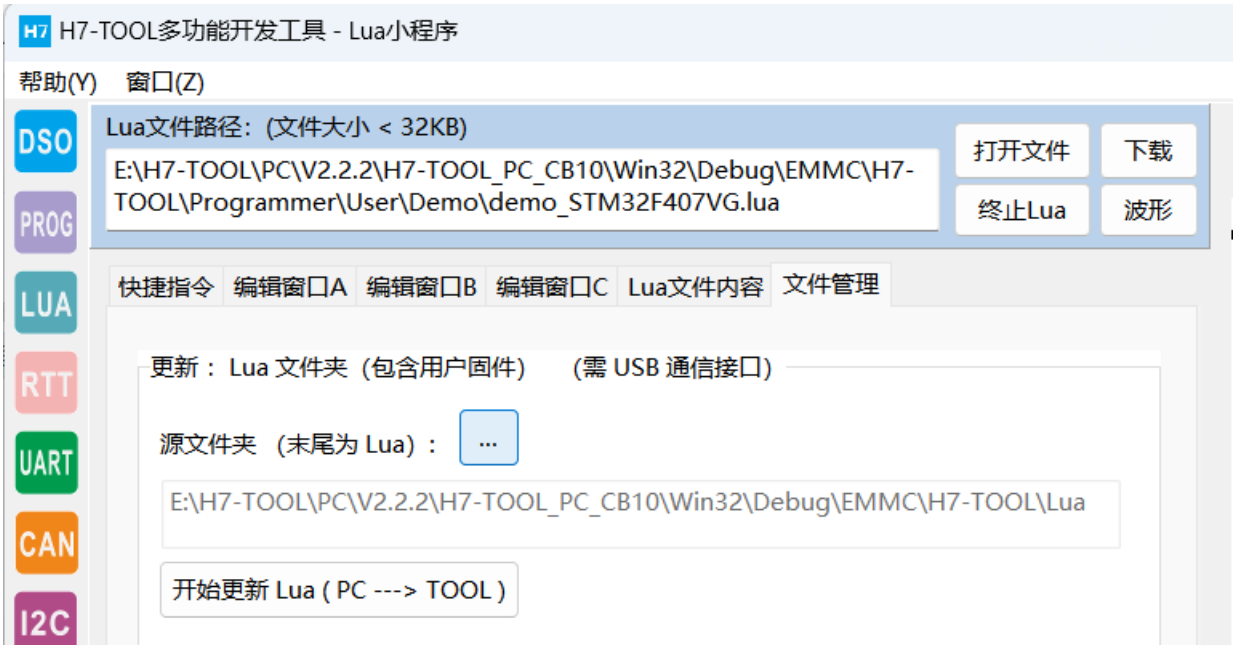
方法 2：在 PC 机维护一个 Lua 文件夹，然后同步到 H7-TOOL

缺省文件夹：h7toolPC_release\EMMC\H7-TOOL\Lua

用户自己创建的lua文件可以放到Lua文件夹，也可以放到其他目录，但是定级目录必须是Lua。

进入Lua页面，切换到“文件管理”，首先选择源目录（末尾必须是Lua），然后点“开始更新Lua（PC----->TOOL）”。

这个更新功能会删除H7-TOOL上多余的文件，相当于文件夹镜像覆盖。



这种模式非常适合工厂批量管理多个H7-TOOL内的数据文件。

2.4. 脱机执行

操作H7-TOOL，无需PC机参与。在缺省 00 界面时长按S键（新版TOOL单击OK键）进入“扩展功能”，然后选择Lua小程序，长按S键（新版TOOL单击OK键）进入文件浏览器，然后选择对应的lua文件，长按S键（新版TOOL单击OK键）开始执行。



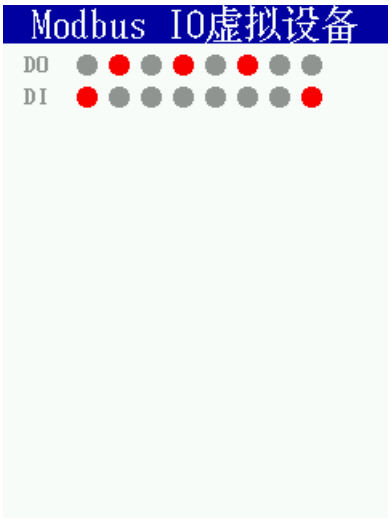






上述是文本界面的Lua小程序执行结果。上方按钮区最多支持 6 个按钮。短按S键（新版TOOL上键和下键）切换按钮，短按C键（新版TOOL点击OK键）执行按钮功能，长按C键退出小程序，长按S键反向切换按钮（一般不用）。

用户也可以创建图形界面的Lua小程序，比如：



3. 程序设计

3.1. 文本界面

文本界面程序的输出结果仅仅通过print函数打印结果，结果会滚动显示。H7-TOOL脱机执行界面支持最多 6 个按钮。显示的按钮个数由用户lua程序决定。



按钮个数和文字通过 lua文件最前面的注释行决定。上述界面的lua代码如下：

```
--F01=RD202-1B820, AutoCalibTest("RD202_18B20")
--F02=RD202-20MA, AutoCalibTest("RD202_20MA")
--F03=RD202-PULSE, AutoCalibTest("RD202_PULSE")
--F04=成品检测, ReleaseTest()
```

--F01 表示有第 1 个按钮，文字内容为RD202-18B20，执行函数为 AutoCalibTest("RD202_18B20")
--F02 表示有第 2 个按钮，文字内容为RD202-20MA，执行函数为 AutoCalibTest("RD202_20MA")
--F03 表示有第 3 个按钮，文字内容为RD202-PULSE，执行函数为 AutoCalibTest("RD202_PULSE")
--F04 表示有第 4 个按钮，文字内容为成品测试，执行函数为 ReleaseTest()

PC软件支持最多 16 个按钮。上面的程序显示结果如下：



3.2. 图形界面

在lua文件开头添加如下行，表示lua输出结果使用图形方式展示。

```
--GUIMODE=1
```

如果没有这行代码，表明是文本输出结果，H7-TOOL小屏上将会展示print结果。如果有这样代码，H7-TOOL小屏上将不会显示print结果。

图形界面的刷新由lua程序自己管理，H7-TOOL提供了GUI相关的API函数，可以用来构建图形元素。请参见API函数章节。

3.3. 读取按键

请参见API函数章节的get_key函数。

4. Lua 基础和常用 API

标准API函数指Lua解释器本身自带的函数。本章节仅仅介绍常用的API函数。多个LUA系统自带API，可以看此贴整理的几个国内网站，整理的都不错：https://www.armbbs.cn/forum.php?mod=viewthread&tid=109728

4.1.1. 注释

用 USB 线连接 H7-TOOL
注释单行

```
--local a = 1
```

注释多行

```
--[[
local a = 1
--]]
```

取消注释多行

```
---[[
local a = 1
--]]
```

4.1.2. 变量

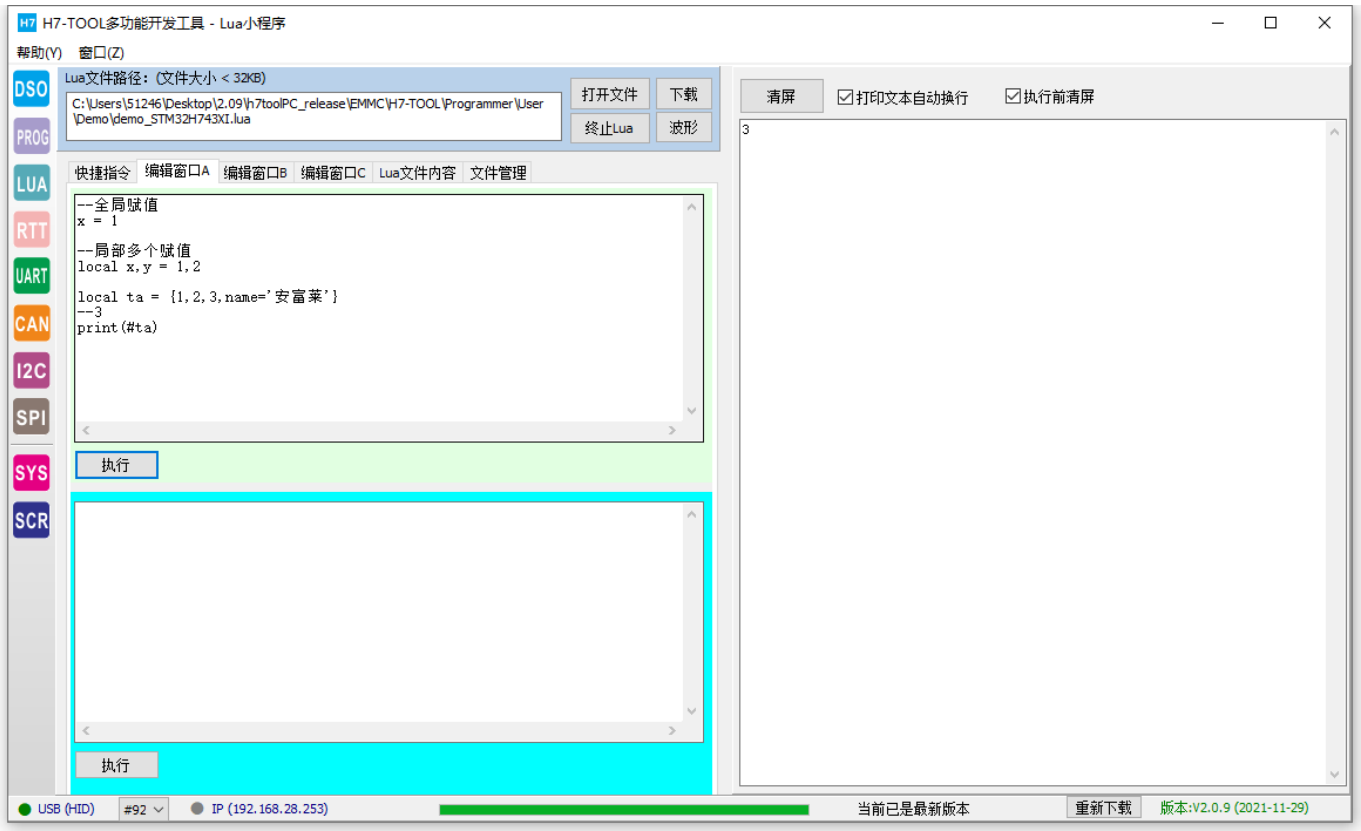
1.变量无需声明
2.变量没声明为 nil,赋值为 nil 等同于删除
3.lua 把 nil,false 视为假,其他都为 true
4.Lua 中的变量全是全局变量,除非用 local 显式声明为局部变量
5.#ta 字符串的长度，也可以获取表格数字索引对应的个数

例子：

```
--全局赋值
x = 1

--局部多个赋值
local x,y = 1,2

local ta = {1,2,3,name='安富莱'}
--3
print(#ta)
```



4.1.3. 条件

支持的条件判断
条件
If
and or not
>= <= == > < ~=

```
if (a >= 0) then   --大于等于
  b = b + 1
else
  b = b - 1
end

if (a ~= 0) then   --不等于
end

if (a==b and c == 0) then --逻辑与

elseif (a~=b) then  --不等于

elseif (not a) then  --逻辑非

else

end
```

4.1.4. 循环

例子

```
local a=10
while (a < 20)
do
   a = a + 1
   print(a)
end

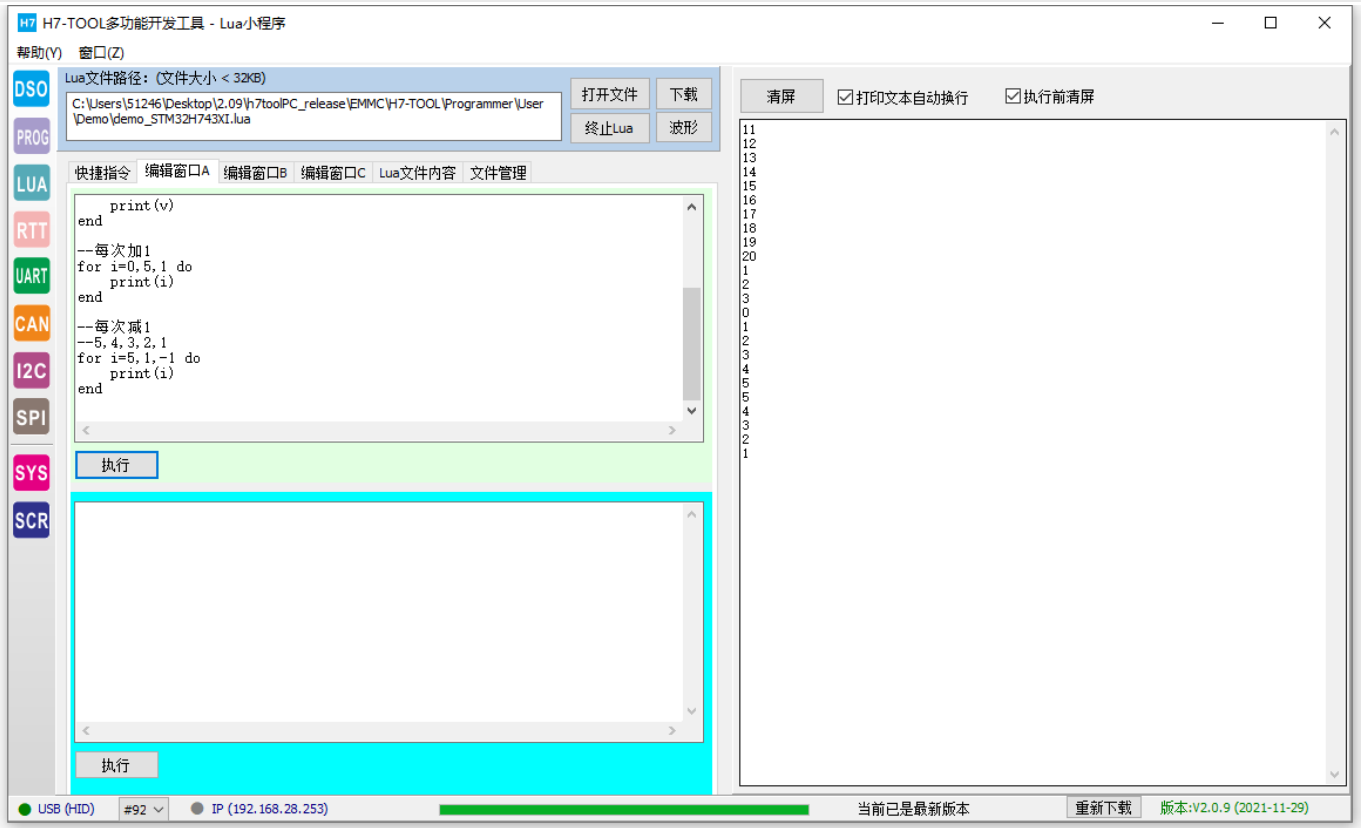
local t = {1,2,3}
for i,v in pairs(t) do
    print(v)
end
```

每次加 1

```
for i=0,5,1 do
    print(i)
end
```

每次减 1

```
--5,4,3,2,1
for i=5,1,-1 do
    print(i)
end
```



4.1.5. 函数

例子：

```
--声明函数，传递 2 个参数，返回 2 个值得和
function add(num1, num2)
   return num1 + num2
end
```

把函数赋值给一个变量，然后可以当参数传递

```
local myprint = function(str)
   print(str)
end

function say(arg1,func)
    func(arg1)
end

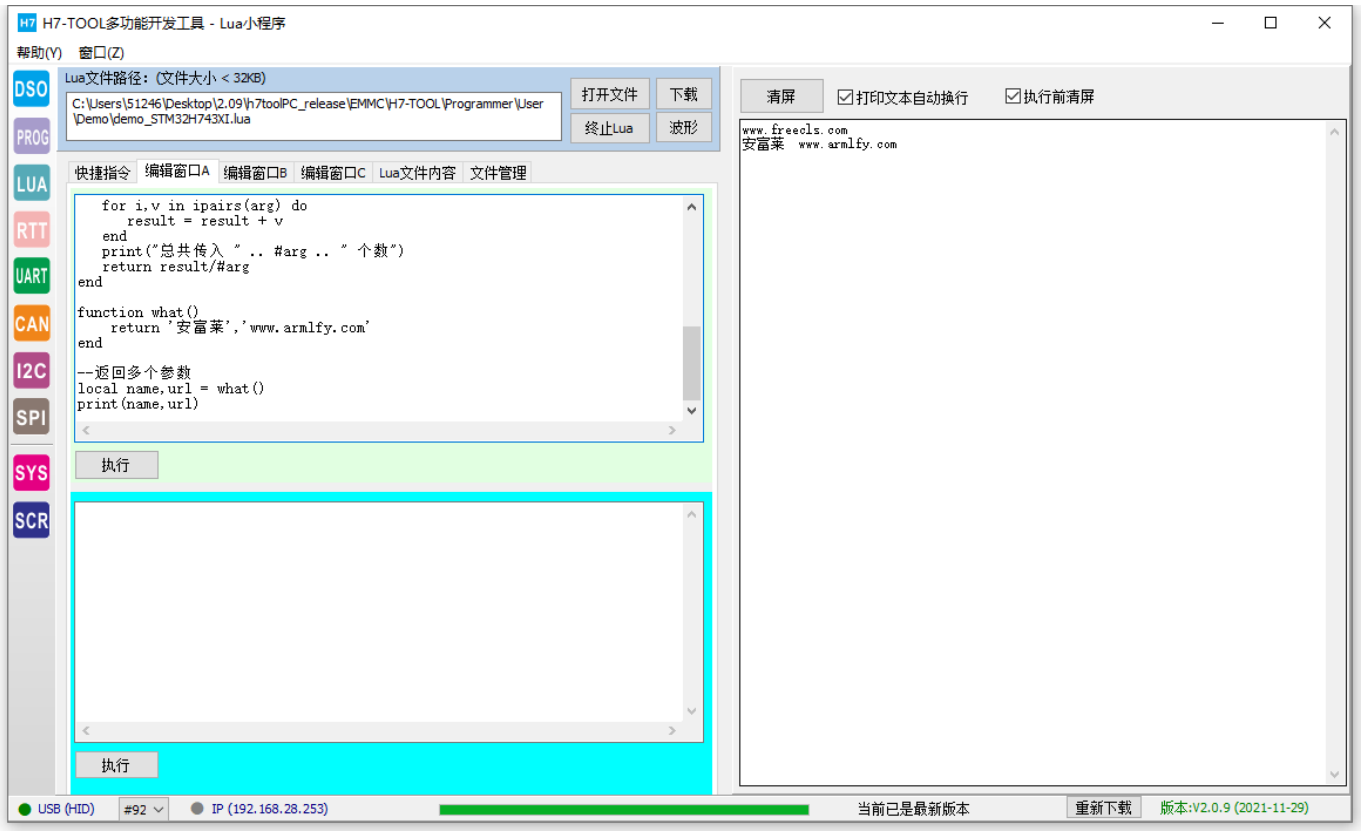
--www.freecls.com
say('www.freecls.com',myprint)
```

可变参数

```
function average(...)
   result = 0
   local arg={...}
   for i,v in ipairs(arg) do
      result = result + v
   end
   print("总共传入 " .. #arg .. " 个数")
   return result/#arg
end

function what()
    return '安富莱','www.armlfy.com'
end

--返回多个参数
local name,url = what()
print(name,url)
```



4.1.6. 常用字符串函数

4.1.6.1. string.byte(s [, i [, j]]) 字符转换成 ascii 数字

string.byte 是用来把字符转换成 ascii 数字，
s 为目标字符串，i 为索引开始位置(从 1 开始)，j 为索引结束位置

```
--默认为第 1 个返回 a 的 ascii 值
local r = string.byte('abcdefg') --97
--从索引 2(b)到索引 4(d)也就是分别返回 bcd 的 ascii 值
local r1,r2,r3 = string.byte('abcdefg',2,4) --98,99,100
```

4.1.6.2. string.char(...) ascii 数值转换成字符

string.char 是把 ascii 数值转换成字符

```
--返回 98 所对应的字符
local r = string.char(98) --a
--返回 98,,99,100 对应的字符并连在一起返回
local r = string.char(98,99,100) --abc
```

4.1.6.3. string.sub (s, i [, j]) 截取字符串

截取字符串(字符串分割，字符串截取)，i 为起始索引，可选参数 j 为结束索引(包含)，都可以为负数，第一个字符索引为 1，最后一个字符为-1

```
local res,s
s = 'www.armfly.com'
res = string.sub(s,5)      --armfly.com
```

```
res = string.sub(s,5,-1)  --armfly.com

--截取后 3 位
res = string.sub(s,-3)    --com

--截取前 3 位
res = string.sub(s,1,3)   --www
```

4.1.6.4. string.find (s, pattern [, init [, plain]]) 字符串查找

字符串查找函数找不到返回 nil，找到了返回开始位置和结束位置， init 为从哪里开始默认为 1，plain 默认为 false 表示利用模式匹配，如果设为 true 则表示纯文本匹配(也就是关闭正则匹配)

```
local str = 'i love programming,11,22,%d+aa'
local s = string.find(str,'222')    --nil
s = string.find(str,'pro')  --8
s = string.find(str,"%d+")    --19(匹配到了,11)
s = string.find(str,"%d+",1,true) --25(由于关闭了模式匹配,所以匹配到了,%d+)
```

4.1.6.5. string.match (s, pattern [, init]) 字符串匹配

它跟 string.find 差不多，只不过能把捕获匹配到的结果并返回

```
local s,res,res1,res2
s = 'http://www.armfly.com'

--由于没有捕获，返回全部匹配
--结果：http://www.armfly.com
res = string.match(s,'http://%a+.%a+.com')

--如果有捕获，则分别返回捕获结果
--结果：www    armfly
res1,res2 = string.match(s,'http://(%a+).(%a+).com')
```

4.1.6.6. string.gsub (s, pattern, repl [, n]) 字符串替换

用来做字符串替换，可选参数 n 代表替换多少次默认全部替换，返回替换后的字符串，也可以指定第二个返回值为替换的次数。

```
local s,res,res1,res2
s = 'http://www.armfly.com'

--结果：http://test.armfly.com，1
res,count = string.gsub(s,'www','test')

--捕获替换
--结果：test.freecls.abc
res = string.gsub(s,'^http://%w+.(%w+).com$','test.%1.abc')
```

```
--w 替换成 t，但是只替换 2 次
--结果：http://ttw.armfly.com
res = string.gsub(s,'w','t',2)
```

4.1.6.7. string.format (formatstring, ・・・) 字符串格式化

字符串格式化类型 c 语言的 sprintf 不说废话以例子来讲解

```
local s = string.format('%d%s',123,'armfly')   --123armlfy

s = string.format('%0.2f',1.234343)     --1.23(保留 2 位)

--转成 16 进制，%X 为大写的 16 进制
local s = string.format('%X',140)       --8C
local s = string.format('%x',140)       --8c
local s = string.format('%04x',140)     --008c
```

4.1.6.8. string.len(s) 返回字符串长度

返回字符串长度=#s

4.1.6.9. string.rep(s,n) 字符串重复 n 次并拼接返回

字符串重复 n 次并拼接返回

4.1.6.10.string.lower(s) 转小写

转小写

4.1.6.11.string.upper(s) 转大写

转大写

4.1.6.12.string.reverse(s) 反转字符串

反转字符串

5. 扩展 API 函数

扩展API函数是专用于H7-TOOL的一组函数，主要包括硬件访问、数制转换、打印显示等函数。

5.1. 系统控制类

系统扩展类主要包括时钟、延时、数值转换、消息传递、打印调试等相关的API函数。

5.1.1. beep 蜂鸣器

函数：

```
beep(a,b,c)
```

说明：

- 控制H7-TOOL内部蜂鸣器鸣叫。非阻塞执行。
- 参数a：鸣叫时间，单位 10ms。0 表示立即停止蜂鸣。
- 参数b：停止时间，单位 10ms。0 表示持续鸣叫。
- 参数c：循环次数。0 表示无限循环。
- 无参数时，响 50ms，停 10ms，循环 1 次。

例子：

```
beep() --蜂鸣器鸣叫一声。
beep(5, 10, 3) --蜂鸣器鸣叫 3 声。响 50ms，停 100ms，循环 3 次。
beep(4, 5, 0) --蜂鸣器鸣叫 3 声。响 40ms，停 50ms，无限循环。
beep(1, 0, 0) --蜂鸣器长鸣。
beep(0, 0, 0) --蜂鸣器停止。
```

5.1.2. delayus 微秒延时

函数：

```
delayus(n)
```

说明：

- 微秒级延时。
- 参数n：延时时间，单位 1us。

例子：

```
delayus(100) --延时 100us。
```

5.1.3. delayms 毫秒延时

函数：

```
delayms(n)
```

说明：

- 毫秒级延时。
- 参数n：延时时间，单位 1ms。

例子：

```
delayms(1000) --延时 1000ms。
```

5.1.4. hex_to_bin HEX 字符串转字符串

函数：

```
hex_to_bin(str)
```

说明：

- 将HEX字符串转换为二进制字符串。
- 参数str: 包含若干HEX整数的字符串
- 字符串中的每个数之间的必须为 1 个空格字符

例子：

```
a = hex_to_bin("12345678 8888 33 444")  --将字符串中的 hex 整数组合为一个二进制字符串
print_hex(a)  --按 HEX 格式打印
```

```
显示结果:
12 34 56 78 88 88 33 44
```

5.1.5. print_hex 打印二进制字符串

函数：

```
print_hex(n1, ...) --第 1 个参数是整数
print_hex(bin, width, addr) --第 1 个参数是字符串
```

说明：

- 可打印整数，也可以打印二进制字符串，显示格式为HEX。
- 第 1 个参数如果是整数，则按整数打印，显示格式为HEX。比如 0x123456 显示为 00123456。最多可带 8 个整数参数。
- 第 1 个参数如果是字符串，则按二进制数组打印。
- bin：二进制字符串，支持中间有 0 值。
- width：每行显示多少字节，缺省 16。
- addr：显示的起始地址，缺省 00000000。

例子：

```
print_hex(0x1234) --打印 1 个整数。显示结果：
00001234

print_hex(1, 2, 0x78901234) --打印 3 个整数。显示结果：
00000001 00000002 78901234

print_hex("\x01\x02\x001234567890a1891124d48512121") --打印字符串，每行 16 字节。显示结果：
01 02 00 31 32 33 34 35 36 37 38 39 30 61 31 38
39 31 31 32 34 64 34 38 35 31 32 31 32 31

print_hex("\x01\x02\x001234567890a1891124d48512121", 8) --打印字符串，每行 8 字节。显示结果：
01 02 00 31 32 33 34 35
36 37 38 39 30 61 31 38
39 31 31 32 34 64 34 38
35 31 32 31 32 31

print_hex("\x01\x02\x001234567890a1891124d4851212189", 8, 0x08000000) --打印字符串，每行 8 字节，显示偏移地址。显示结果：
08000000 : 01 02 00 31 32 33 34 35
08000008 : 36 37 38 39 30 61 31 38
08000010 : 39 31 31 32 34 64 34 38
08000018 : 35 31 32 31 32 31 38 39
```

5.1.6. hex_to_table HEX 字符串转换为数组

函数：

```
hex_to_table(str)
```

说明：

- 将HEX字符串转换为数组。
- 参数str: 包含若干HEX整数的字符串
- 字符串中的每个数之间的必须为 1 个空格字符

例子：

```
a = {hex_to_table("12345678 8888 33 444")}  --将字符串中的 hex 整数分离出来
print_hex(a[1], a[2], a[3], a[4])  --按 HEX 格式打印
print(a[1], a[2], a[3], a[4])  --按十进制格式打印

显示结果:
12345678 00008888 00000033 00000444
305419896.0    34952.0 51.0    1092.0
```

5.1.7. write_clock 修改时钟

函数：

```
write_clock(year, mon, day, hour, min, sec)
```

说明：

- 修改H7-TOOL内部时钟。
- 参数 year：年，2000 以上。
- 参数 mon：月份，1-12。
- 参数 day：日，1-31。
- 参数 hour：小时，0-23。
- 参数 min：分钟，0-59。
- 参数 sec：秒，0-59。
- 函数无返回值

例子：

```
write_clock(2023, 12, 8, 0, 1, 30) --修改日期时间为 2023 年 12 月 8 日 00:01:30
```

5.1.8. read_clock 读时钟

函数：

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

说明：

- 读H7-TOOL内部时钟，返回年、月、日、星期、小时、分钟、秒、星期。
- 星期值是 1-7

例子：

```
year,mon,day,hour,min,sec,week = read_clock()  --读时钟赋值给年月日时分秒星期变量
print(year,mon,day,hour,min,sec,week) --打印这些变量
```

```
显示结果: 2023 年 12 月 8 日 00:50:55 星期五
2023.0 12.0 8.0 0.0 50.0 55.0 5.0
```

5.1.9. get_runtime 读运行时长(ms)

函数：

```
d = get_runtime()
```

说明：

- 获得H7-TOOL复位后的运行时长，单位ms
- 返回值d ： 运行时长，单位ms。int32 类型。
- 最长可以表示 2147483647ms = 24.85 天。如果你的产品连续运行时间超过这个数，则必须考虑溢出问题。

例子：

```
n = get_runtime()
print(n)
```

```
显示结果: 表示已运行 741461ms
```

```
741461.0
```

5.1.10. check_runtime 判断上电运行时刻是否到

函数：

```
d = check_runtime(t0)
```

说明：

- 判断上电运行时刻是否到了。ms单位。
- 返回值d：当前时间和给定时间t0 的差值。
- 参数t0：上一个运行时刻，单位ms。

例子：

```
print("每秒蜂鸣一声。这是一个死循环例子，长按 C 键可终止 lua 程序")
n = 0
t0 = get_runtime()
while(1) do
   if (check_runtime(t0) > 1000) then  --比较当前时刻和 t0 之间的差值是否大于 1000ms
      t0 = get_runtime() --获得当前运行时刻（ms）
      beep()  --每秒蜂鸣一声
      print(n) --显示执行次数
      n = n + 1
   end
end
```

```
执行结果:
每秒蜂鸣一声。这是一个死循环例子，长按 C 键可终止 lua 程序
0
1
2
3
```

5.1.11. get_runtime_us 获取运行时间

函数：

```
d = get_runtime_us()
```

说明：

- 获得H7-TOOL复位后的运行时长，单位us
- 返回值d ： 运行时长，单位us。int64 类型。
- 因为是 64 位整数，可表示时间非常大，可以不用考虑溢出问题。

例子：

```
n = get_runtime_us()
print(n)
```

```
显示结果: 表示已运行 601700747us （601 秒）
601700747.0
```

5.1.12. check_runtime_us 判断上电运行时刻是否到

函数：

```
d = check_runtime_us(t0)
```

说明：

- 判断上电运行时刻是否到了。us单位。
- 返回当前时间和给定时间t0 的差值。
- 参数t0：上一个运行时刻，单位ms。

例子：

```
if (check_runtime_us(0) > 5000000) then
   print("上电已运行 5 秒")
else
   print("上电启动中")
end
```

```
执行结果:
上电已运行 5 秒
```

5.1.13. put_key 发送按键事件

函数：

```
put_key("s")   --字符串形参
put_key(a)   --整数形参
```

说明：

- 产生一个按键事件存放到按键FIFO。V2.25 固件之前FIFO大小为 10 个，之后调整为 50 个。
- 形参：取值如下：

字符串键值	整数键值	功能	备注
"PressDown_UP"	1	↑键按下	第 1 代H7-TOOL S键 第 2 代H7-TOOL 上键
"Release_UP" "up"	2	↑键释放	
"LongPress_UP"	3	↑键长按	
"LongRelease_UP"	4	↑键长按之后的释放	
"AutoResend_UP"	5	↑键长按之后自动重发键值	
"DoubleClick_UP"	6	↑键双击	
"LongPressIdle_UP"	7	键盘空闲 1 秒之后，首次长按	
"PressDown_DOWN"	8	↓键按下	第 1 代H7-TOOL C键 第 2 代H7-TOOL 下键
"Release_DOWN"	9	↓键释放	

"down"			
"LongPress_DOWN"	10	↓键长按	
"LongRelease_DOWN"	11	↓键长按之后的释放	
"AutoResend_DOWN"	12	↓键长按之后自动重发键值	
"DoubleClick_DOWN"	13	↓键双击	
"LongPressIdle_DOWN"	14	键盘空闲 1 秒之后，首次长按	
"PressDown_OK"	15	OK键按下	第 2 代H7-TOOL才有独立的OK按键
"Release_OK" "ok"	16	OK键释放	
"LongPress_OK"	17	OK键长按	
"LongRelease_OK"	18	OK键长按之后的释放	
"AutoResend_OK"	19	OK键长按之后自动重发键值	
"DoubleClick_OK"	20	OK键双击	
"LongPressIdle_OK"	21	键盘空闲 1 秒之后，首次长按	
"PressDown_CANCEL"	22	Cancel键按下	第 2 代H7-TOOL才有独立的Cancel按键
"Release_CANCEL" "cancel"	23	Cancel键释放	
"LongPress_CANCEL"	24	Cancel键长按	
"LongRelease_CANCEL"	25	Cancel键长按之后的释放	
"AutoResend_CANCEL"	26	Cancel键长按之后自动重发键值	
"DoubleClick_CANCEL"	27	Cancel键双击	
"LongPressIdle_CANCEL"	28	键盘空闲 1 秒之后，首次长按	
"PressDown_HOME"	29	Home键按下	第 2 代H7-TOOL才有独立的Home按键
"Release_HOME" "home"	30	Home键释放	
"LongPress_HOME"	31	Home键长按	
"LongRelease_HOME"	32	Home键长按之后的释放	
"AutoResend_HOME"	33	Home键长按之后自动重发键值	
"DoubleClick_HOME"	34	Home键双击	
"LongPressIdle_HOME"	35	键盘空闲 1 秒之后，首次长按	

5.1.14. clear_key 清除按键缓冲区

函数：

```
clear_key()
```

说明：

- 清除按键缓冲区。无返回值。
- 当执行一个较长的操作后，有可能需要清按键缓冲区，忽略用户等待过程中产生的无效按键。

例子：

```
clear_key()
```

5.1.15. get_key 获取实际按键消息

函数：

```
get_key()
```

说明：

- 用于获取按键消息

旧款 TOOL 按键消息（TOOL 上的 C 键长按用于终止 lua 程序，不得在 lua 中使用）：

```
beep()
print("请按键，长按 c 退出")
while(1)
do
    k = get_key()
    if  (k > 0) then
        print(k)
    end
end
```



新款 TOOL 按键消息（TOOL 上的 C 键长按用于终止 lua 程序，不得在 lua 中使用）：

```
beep()
print("请按键，长按 c 退出")
while(1)
do
    k = get_key()
    if  (k > 0) then
        print(k)
    end
end
```



5.1.16. put_msg 发送消息

函数：

```
put_msg(code, param)
```

说明：

- 发送一个消息。lua程序通知应用程序执行某些操作。具体用法参考例子。
- 形参 code ： 消息代码
- 形参 param ： 消息参数

例子：

```
put_msg(0x00001004, 0x0000)  --选择烧录模式为单路 （以下案例仅在脱机烧录界面才会生效）
put_msg(0x00001004, 0x0100)  --选择烧录模式为一拖四模式，烧录第 1-4 路
put_msg(0x00001004, 0x0101)  --选择烧录模式为一拖四模式，烧录第 1 路
put_msg(0x00001004, 0x0102)  --选择烧录模式为一拖四模式，烧录第 2 路
put_msg(0x00001004, 0x0103)  --选择烧录模式为一拖四模式，烧录第 3 路
put_msg(0x00001004, 0x0104)  --选择烧录模式为一拖四模式，烧录第 4 路

put_msg(0x00001004, 0x0105)  --选择烧录模式为一拖四模式，烧录第 1-2 路
put_msg(0x00001004, 0x0106)  --选择烧录模式为一拖四模式，烧录第 1-3 路
put_msg(0x00001004, 0x0107)  --选择烧录模式为一拖四模式，烧录第 1-4 路
put_msg(0x00001004, 0x1001)  --选择烧录模式为 1 拖 16 模式，只有第 1 个通道有效
put_msg(0x00001004, 0x1002)  --选择烧录模式为 1 拖 16 模式，只有第 1-2 个通道有效
put_msg(0x00001004, 0x1003)  --选择烧录模式为 1 拖 16 模式，只有第 1-3 个通道有效
put_msg(0x00001004, 0x1004)  --选择烧录模式为 1 拖 16 模式，只有第 1-4 个通道有效
put_msg(0x00001004, 0x1005)  --选择烧录模式为 1 拖 16 模式，只有第 1-5 个通道有效
put_msg(0x00001004, 0x1006)  --选择烧录模式为 1 拖 16 模式，只有第 1-6 个通道有效
put_msg(0x00001004, 0x1007)  --选择烧录模式为 1 拖 16 模式，只有第 1-7 个通道有效
put_msg(0x00001004, 0x1008)  --选择烧录模式为 1 拖 16 模式，只有第 1-8 个通道有效
put_msg(0x00001004, 0x1009)  --选择烧录模式为 1 拖 16 模式，只有第 1-9 个通道有效
put_msg(0x00001004, 0x100A)  --选择烧录模式为 1 拖 16 模式，只有第 1-10 个通道有效
put_msg(0x00001004, 0x100B)  --选择烧录模式为 1 拖 16 模式，只有第 1-11 个通道有效
put_msg(0x00001004, 0x100C)  --选择烧录模式为 1 拖 16 模式，只有第 1-12 个通道有效
```

```
put_msg(0x00001004, 0x100D)  --选择烧录模式为 1 拖 16 模式，只有第 1-13 个通道有效
put_msg(0x00001004, 0x100E)  --选择烧录模式为 1 拖 16 模式，只有第 1-14 个通道有效
put_msg(0x00001004, 0x100F)  --选择烧录模式为 1 拖 16 模式，只有第 1-15 个通道有效
put_msg(0x00001004, 0x1010)  --选择烧录模式为 1 拖 16 模式，只有第 1-16 个通道有效
```

5.1.17. get_rng 获得随机数

函数：

```
rng = get_rng(len)
```

说明：

- 获取一个指定长度随机数。返回值为二进制字符串。
- 形参 len ： 随机数字节个数。值阈 1 - 1024。
- 返回值 rng ： 计算结果，32 位整数

例子：

```
rng = get_rng(12)
print_hex(rng)
```

```
执行结果:
AA 4E 12 24 7D D6 82 AB 94 DC 33 20
```

5.1.18. crc32_stm32 CRC32 校验

函数：

```
crc = crc32_stm32(bin, initvalue)
```

说明：

- 按照STM32 的CRC32 硬件算法对数据进行计算。不足 4 字节补 0，小端模式传递数据。
- 形参 bin ： 二进制字符串。
- 形参 initvalue ： 初值，缺省是 0xFFFFFFFF

例子：

```
crc = crc32_stm32("\x12\x34\x56\x78\x00\x01\x02\x03", 0xFFFFFFFF)
print_hex(crc)
crc = crc32_stm32("\x12\x34\x56\x78\x00\x01", 0xFFFFFFFF)
print_hex(crc)
```

```
执行结果:
36BCC99F
557A22FF
```

STM32 的硬件 CRC 算法和通用的 CRC32 算法不同之处：

在其他平台如果想要得到和STM32 一样的结果，需要将输入的数据按字节反转，再将字节存放次序反转，4字节为一组,然后再按常规CRC32 算法计算。以 0x41 为例：将 0x41 写成二进制为：0100 0001,将其反转为：1000 0010 ,反转后的数十六进制为：0x82 。同理,将［0x41, 0x42, 0x43, 0x44］依次反转便为［0x82, 0x42, 0xC2, 0x22］,同时字节的存放也需要反转,即［0x22, 0xC2, 0x42, 0x82］。

C语言函数：

```
// 按 STM32 方式计算 CRC32，di 是 32 字节整数数组，len 是 4 字节整数的个数（字节个数/4)
uint32 t CRC32_STM32(uint32_t *di, uint32_t len)
{
  uint32_t crc_poly = 0x04C11DB7;
  uint32_t data = 0xFFFFFFFF;

  for (uint32_t i = 0: i < len; i++)
  {
    data ^= di[i];
    for (uint8_t j = 0; j < 32; j++)
    {
     if (data & 0x80000000)
     {
       data = (data << 1) ^ crc_poly;
     }
     else
     {
       data <<= 1;
     }
    }
  }
  return (data);
}
```

lua语言函数：

```
function CRC32_STM32(di)
  local i,j
  local crc_poly = 0x04C11DB7
  local data = 0xffffffff
  local d0
  local len

  --不足 4 字节，末尾补 0
  len = #di
  for i = 1, len % 4, 1 do
    di = di.."\x00"
  end

  len = #di
  for i = 1, len / 4, 1 do
```

```
    d0 = (string.byte(di, (i - 1) * 4 + 1) << 0) +
         (string.byte(di, (i - 1) * 4 + 2)  << 8) +
         (string.byte(di, (i - 1) * 4 + 3)  << 16) +
         (string.byte(di, (i - 1) * 4 + 4) << 24)

    data = data ~ d0
    for j = 1,32,1 do
      if ((data & 0x80000000) == 0x80000000) then
        data = ((data << 1) & 0xFFFFFFFF) ~ crc_poly
      else
        data = (data << 1) & 0xFFFFFFFF
     end
    end
  end
  return (data)
end
```

5.1.19. crc16_modbus CRC16 校验

函数：

```
crc = crc16_modbus(bin)
```

说明：

- 按照 Modbus RTU 协议规范计算CRC16 校验值。
- 形参 bin ： 二进制字符串。
- 返回值 crc ： 计算结果，uint16_t。

例子：

```
crc = crc16_modbus("\x01\x10\x00\x02\x00\x02\x04\x12\x34\x56\x78")
print_hex(crc)
```

```
执行结果:
00000942  --低 16 字节有效
```

5.2. DAC 模拟量输出，信号发生器，0-20mA 输出

5.2.1. dac_on/dac_off 打开/关闭 DAC 电源

函数：

```
dac_on() --打开 DAC 电源
dac_off()--关闭 DAC 电源
```

说明：

- 这两个函数都是直接调用即可。由于信号发生器和 0-40mA输出都是通过DAC实现的，所以调用这两个函数可以实现信号发生器和 0-40mA输出开关。

例子：

见 5.2.4

5.2.2. dac_write 写 DAC 数值

函数：

```
dac_write(参数 1)
```

说明：

- 参数 1：范围 0-4095，用于写DAC值，因为是 12bit DAC，所以范围是 0-4095

例子：

见 5.2.4

5.2.3. dac_volt2dac 电压值换算为 DAC 值

函数：

```
dac_volt2dac(参数 1)
```

说明：

- 参数 1：单位mV，支持正负 10V，对应的数值范围是-10000 到 10000，即-10000mV到 10000mV，用于电压值换算为DAC值。

例子：

见 5.2.4

5.2.4. dac_curr2dac 电流值换算为 DAC 值

函数：

```
dac_on() --打开 DAC 电源
dac_off()--关闭 DAC 电源
```

说明：

- 参数 1：单位uA，支持 0-20mA，对应的参数范围是 0 到 20000，即 0uA到 20000uA，用于电流值换算为DAC值。

例子：，

设置电流输出 10mA：

```
local value

dac_on()
value = dac_curr2dac(10000)
dac_write(value)
```

设置输出电压 5V：

```
local value

dac_on()
value = dac_volt2dac(5000)
dac_write(value)
```

5.2.5. dac_volt 写电压值

函数：

```
dac_volt(参数 1)
```

说明：

- 参数 1：写电压值，单位mV，支持正负 10V，对应的数值范围是-10000 到 10000，即-10000mV到 10000mV

例子：

见 5.2.6

5.2.6. dac_curr 写电流值

函数：

```
dac_curr(参数 1)
```

说明：

- 参数 1：写电流值，单位uA，支持 0-20mA，对应的参数范围是 0 到 20000，即 0uA到 20000uA

例子：

函数dac_volt等效于dac_volt2dac + dac_write， 比如设置输出电压 5V，那么我们就可以：

```
local value

dac_on()
dac_volt(5000)
```

函数dac_curr等效于dac_curr2dac + dac_write，比如设置电流输出 10mA，那么我们就可以：

```
local value

dac_on()
dac_curr(10000)
```

使用另外一个TOOL的 4-20mA输入采集，实际电流效果：



使用台式万用表测量实际值



5.2.7. write_tvcc_dac 设置 TVCC 的 DAC 值

函数：

```
write_tvcc_dac(参数 1)
```

说明：

- 参数 1：用于设置TVCC的DAC值，范围 0 -127

例子：

见 5.2.8

5.2.8. write_tvcc_volt 设置 TVCC 的电压

函数：

```
write_tvcc_volt(参数 1)
```

说明：

- 参数 1：用于设置TVCC的电压，单位mV，范围 1.2V - 5.0V，即 1200mV - 5000mV

例子：

推荐使用函数 write_tvcc_volt 设置电压值，比如设置输出 3.3V，那么就是 write_tvcc_volt(3300)

5.2.9. dac_wave_start 输出波形

函数：

```
lua_DacWaveStart(参数 1, 参数 2, 参数 3)
```

说明：

- 参数 1：波形数据缓冲

  注意这个缓冲是每两个字节组成一个DAC值，值得范围是 0-4095，　并且数值是以字符串形式填写才行，为了方便大家使用，需要使用函数dac_volt2dac做转换即可。
- 参数 2：是输出一个波形数据点的速度
- 参数 3：0 表示无限输出，1 表示输出一次。

例子：

比如周期性输出 8 个数值的波形，1123mV，　556mV，2445mV，　5334mV，　-3334mV，　-5556mV,，　7890mV，　3231mV。

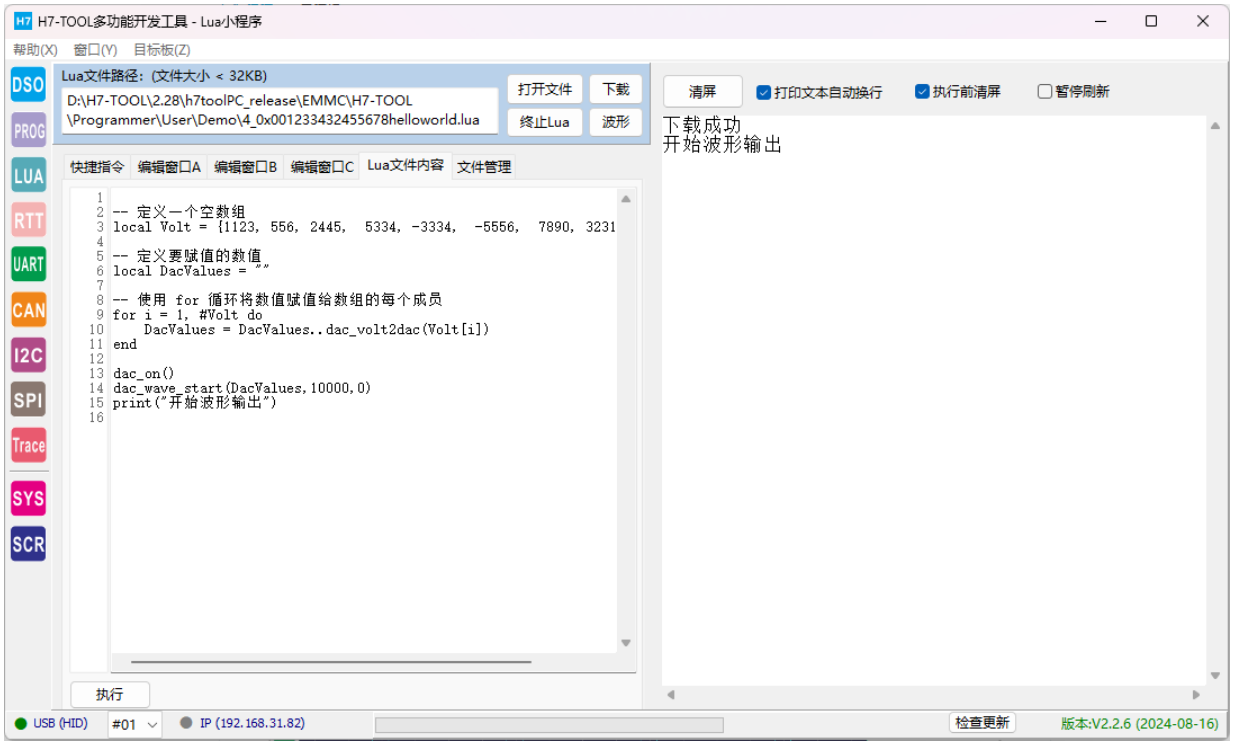
代码实现如下：

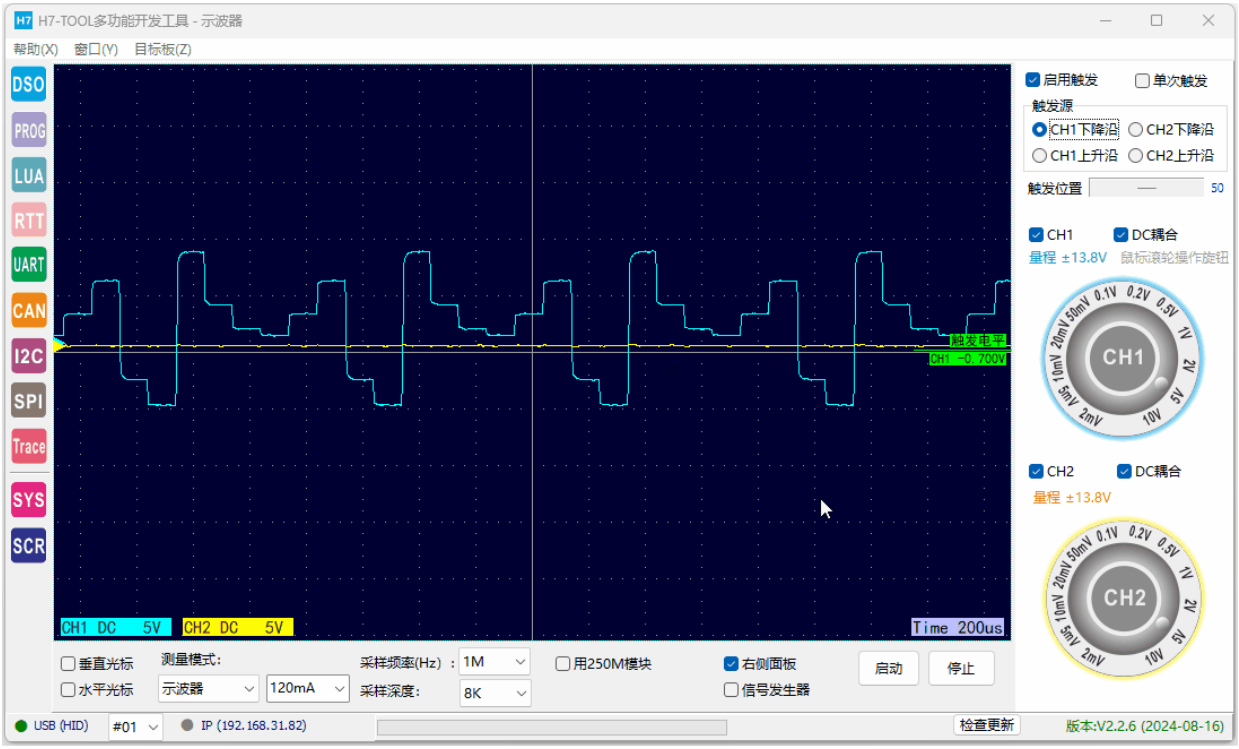
```
-- 定义一个空数组
local Volt = {1123, 556, 2445,  5334, -3334,  -5556,  7890, 3231}

-- 定义要赋值的数值
local DacValues = ""

-- 使用 for 循环将数值赋值给数组的每个成员
for i = 1, #Volt do
    DacValues = DacValues..dac_volt2dac(Volt[i])
end

dac_on()
dac_wave_start(DacValues,10000,0)
print("开始波形输出")
```





5.2.10. dac_wave_stop 停止波形

函数：

```
lua_DacWaveStop()
```

说明：

- 无形参和返回值，直接调用即可，用于停止信号发生器输出。

5.2.11. 范例：Matlab 生成任意波形数据给 TOOL 使用

https://www.armbbs.cn/forum.php?mod=viewthread&tid=111384

5.3. GUI 显示

5.3.1. RGB565 RGB888 转 RGB565

函数：

```
re = RGB565(r, g, b)
```

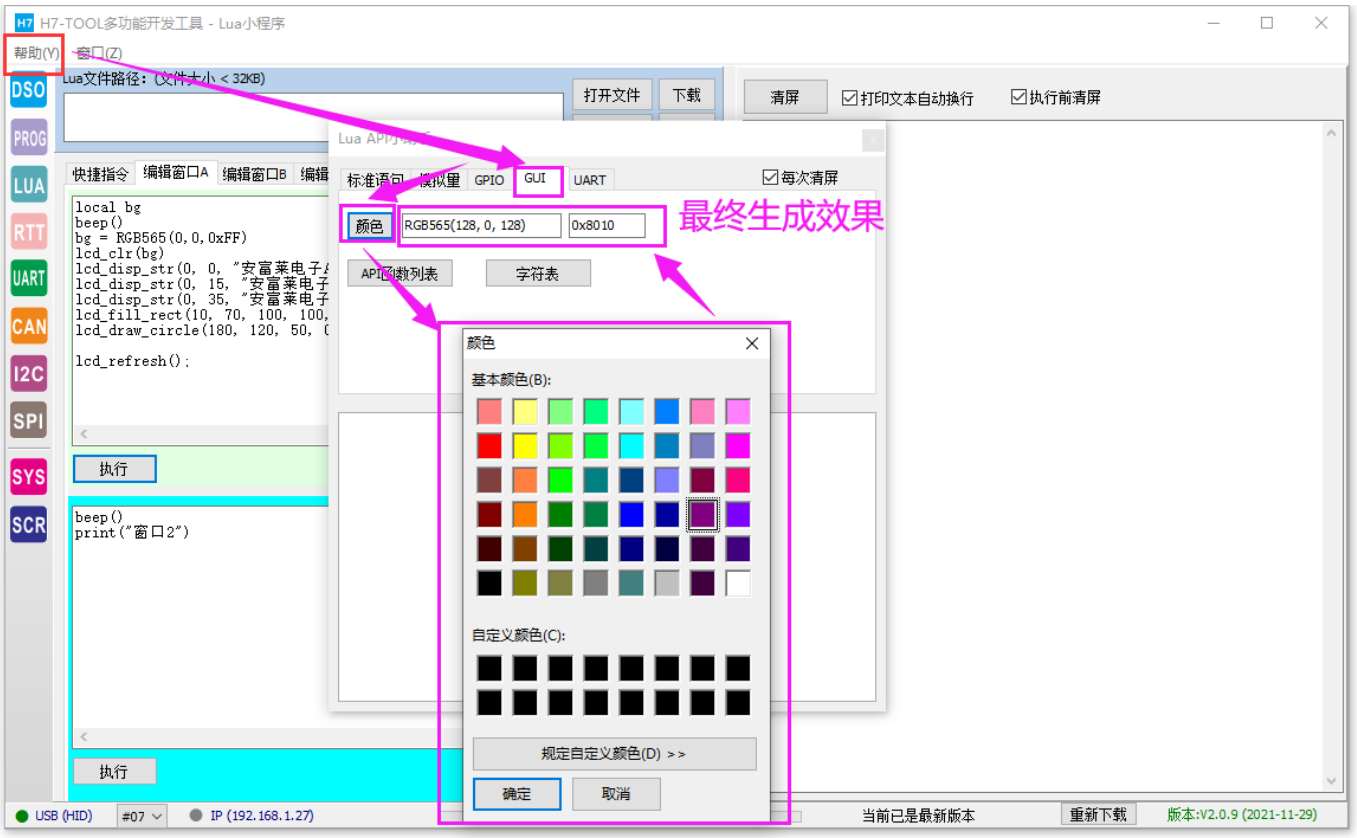
说明：

- 将RGB888 颜色转换为RGB565，当前TOOL显示屏是采用的RGB565 颜色格式。
- 参数r，g，b分别代表RGB888 颜色支持的Red 8bit，Green 8bit和Blue 8bit
- 返回值是RGB888 转换RGB565 的最终值

例子：

```
lcd_clr(RGB565(0xFF, 0xFF, 0xFF)) --清屏，白色
```

RGB888 的颜色取值效果，可以看这里



5.3.2. lcd_refresh 刷新页面

函数：

```
lcd_refresh()
```

说明：

- 无形参和返回值，用于刷新页面。

例子：

5.3.3. lcd_clr 清屏

函数：

```
lcd_clr(参数 1)
```

说明：

- 用于将整个屏幕清为指定颜色
- 参数 1 是RGB565 颜色值

例子：

```
lcd_clr(RGB565(0xFF, 0xFF, 0xFF))
lcd_clr(0x1234)
```

5.3.4. lcd_disp_str 显示字符串

函数：

```
lcd_disp_str(x, y, str, fontsize, front_color, back_color, width, align)
```

说明：

- 用于字符显示
- x, y ：显示坐标, 像素单位, 0-239
- str ：字符串, 支持GBK编码中文
- fontsize ：字体大小. 12=12 点阵，16=16 点阵，24=24 点阵，32 表示 32 点阵
- front_color ：字体颜色, 可以用 RGB(0,0,0)
- back_color ：背景颜色
- width ：宽度
- align ：对齐. 0=左对齐, 1=居中, 2=右对齐

例子：

见 5.3.9

5.3.5. lcd_fill_rect 填充矩形窗口

函数：

```
lcd_fill_rect(x, y, h, w, color)
```

说明：

- 用于
- x, y ：矩形左上角, 像素单位, 0-239
- h, w ：矩形高度和宽度, 像素单位, 1-240
- color ：填充颜色

例子：

见 5.3.9

5.3.6. lcd_draw_rect 绘制矩形边框

函数：

```
lcd_fill_rect(x, y, h, w, color)
```

说明：

- 用于
- x，y ：矩形左上角，像素单位，0-239
- h，w ：矩形高度和宽度，像素单位，1-240
- color ：填充颜色

例子：

见 5.3.9

5.3.7. lcd_draw_circle 绘制圆圈

函数：

```
lcd_draw_circle(x, y, r, color)
```

说明：

- 用于绘制圆圈
- x，y ：圆心坐标，像素单位，0-239
- r ：圆半径，像素单位
- color ：边框颜色

例子：

见 5.3.9

5.3.8. lcd_draw_line 绘制直线

函数：

```
lcd_draw_line(x1, y1, x2, y2, color)
```

说明：

- 用于绘制直线
- x1，y1 ：第 1 个点坐标，像素单位，0-239
- x2，y2 ：第 2 个点坐标，像素单位，0-239
- color ：直线颜色

例子：

见 5.3.9

5.3.9. lcd_disp_label 绘制标签

函数：

```
lcd_disp_label(x, y, h, w, str, align, fontzize, front_color, back_color)
```

说明：

- 用于绘制标签
- x, y ：矩形左上角，像素单位，0-239
- h, w ：矩形高度和宽度，像素单位，1-240
- str ：字符串
- align：对齐方式，0 表示左对齐，1 表示居中，2 表示右对齐
- fontzize ：字体大小 12 16 24 32
- front_color ：字体颜色
- back_color ：字体背景颜色

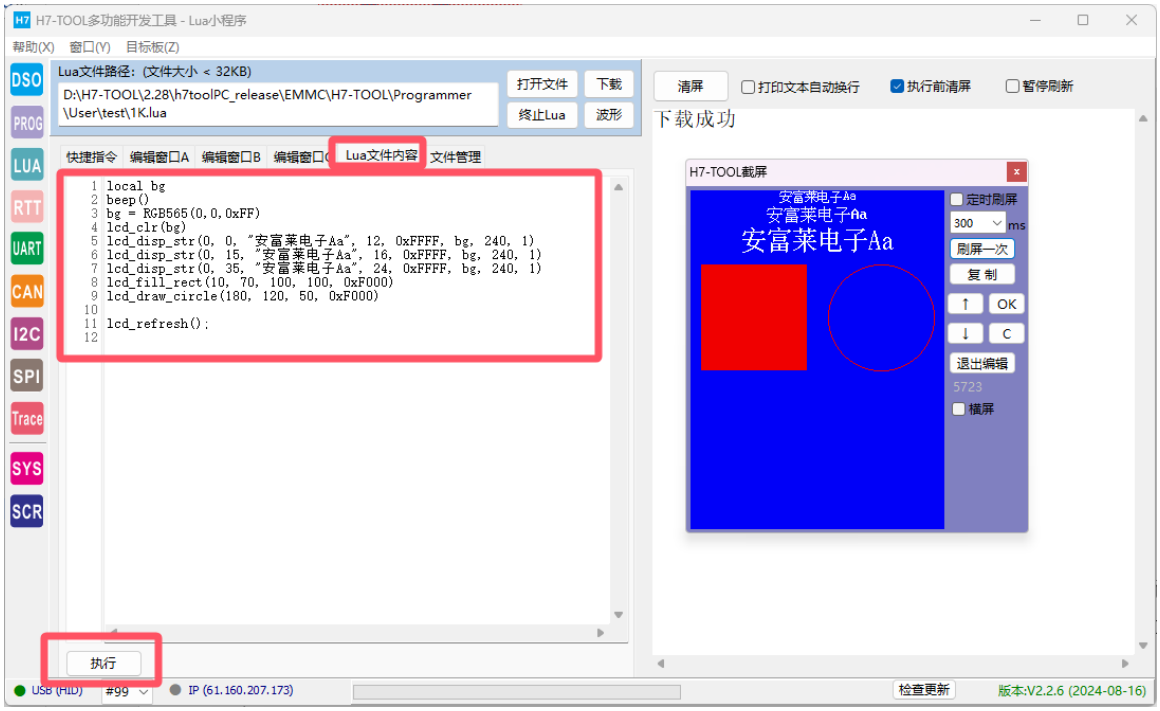
例子：

（1）测试的时候，将 TOOL 运行在静态的界面，没有内容刷新的，防止原 APP 程序定时更新影响了 LUA 界面 API 的执行。

（2）如果大家的 LUA 小程序里面有 delayms 或者死循环之类的操作，下次更新 LUA，需要长按 TOOL 的 C 键退出当前 LUA 的执行，因为 LUA 是阻塞式的执行方式。

```
local bg
beep()
bg = RGB565(0,0,0xFF)
lcd_clr(bg)
lcd_disp_str(0, 0, "安富莱电子 Aa", 12, 0xFFFF, bg, 240, 1)
lcd_disp_str(0, 15, "安富莱电子 Aa", 16, 0xFFFF, bg, 240, 1)
lcd_disp_str(0, 35, "安富莱电子 Aa", 24, 0xFFFF, bg, 240, 1)
lcd_fill_rect(10, 70, 100, 100, 0xF000)
lcd_draw_circle(180, 120, 50, 0xF000)

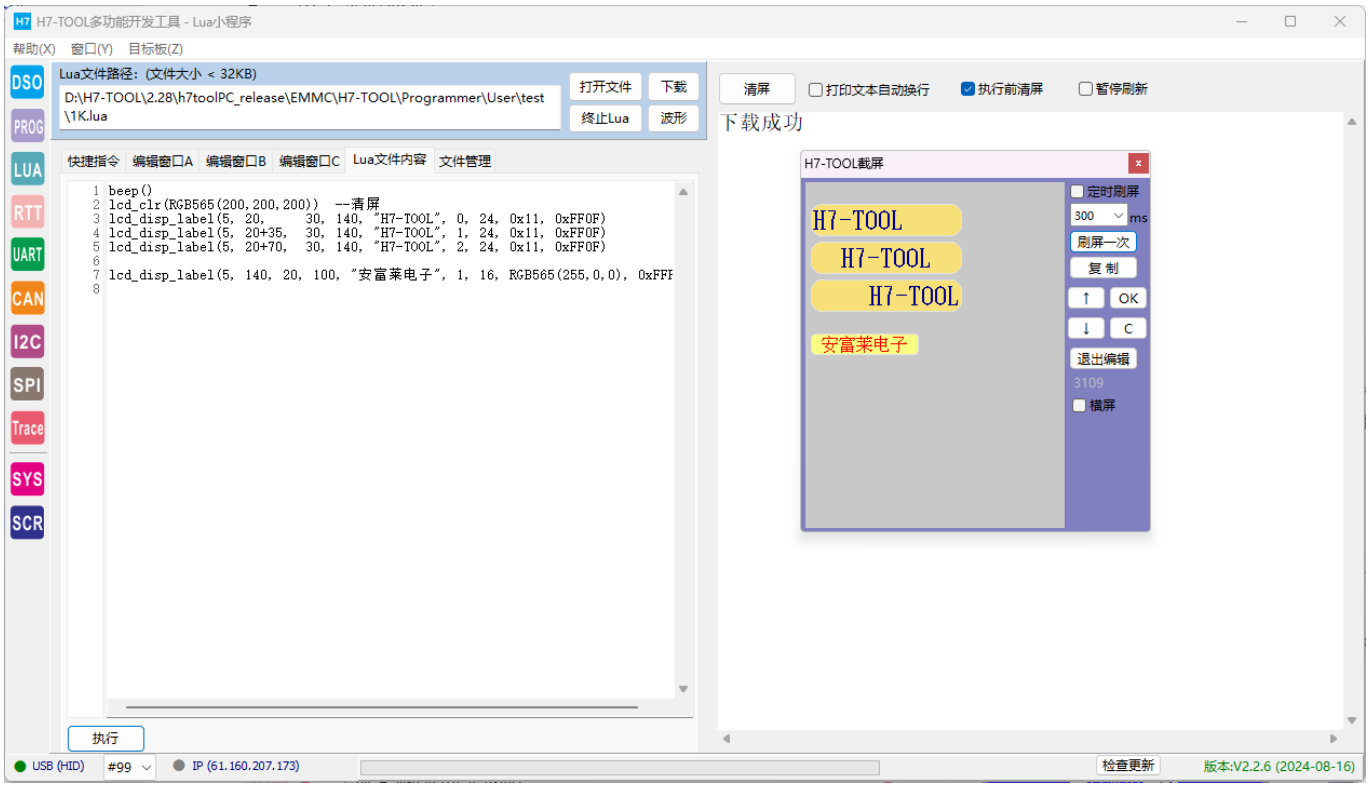
lcd_refresh();
```




```
beep()
lcd_clr(RGB565(200,200,200))  --清屏
lcd_disp_label(5, 20,      30, 140, "H7-TOOL", 0, 24, 0x11, 0xFF0F)
```

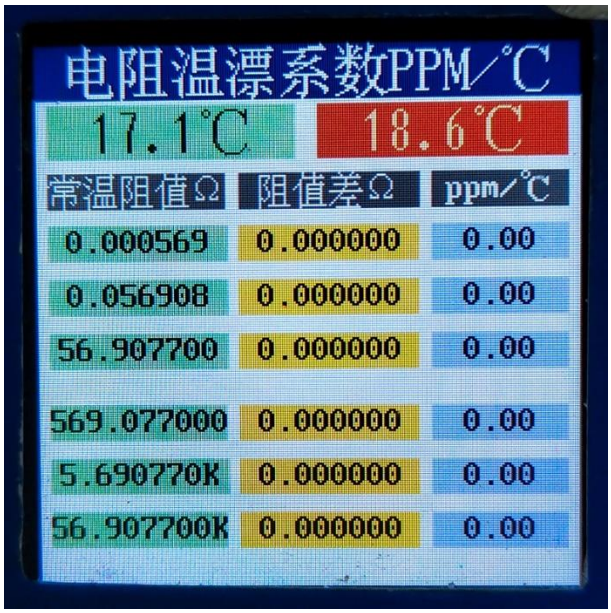
```
lcd_disp_label(5, 20+35,  30, 140, "H7-TOOL", 1, 24, 0x11, 0xFF0F)
lcd_disp_label(5, 20+70,  30, 140, "H7-TOOL", 2, 24, 0x11, 0xFF0F)

lcd_disp_label(5, 140, 20, 100, "安富莱电子", 1, 16, RGB565(255,0,0), 0xFFF0)
```



5.3.10. 范例：电阻温漂系数测量

https://www.armbbs.cn/forum.php?mod=viewthread&tid=101814



5.4. 文件系统

5.4.1. f_dir 列举目录文件

函数：

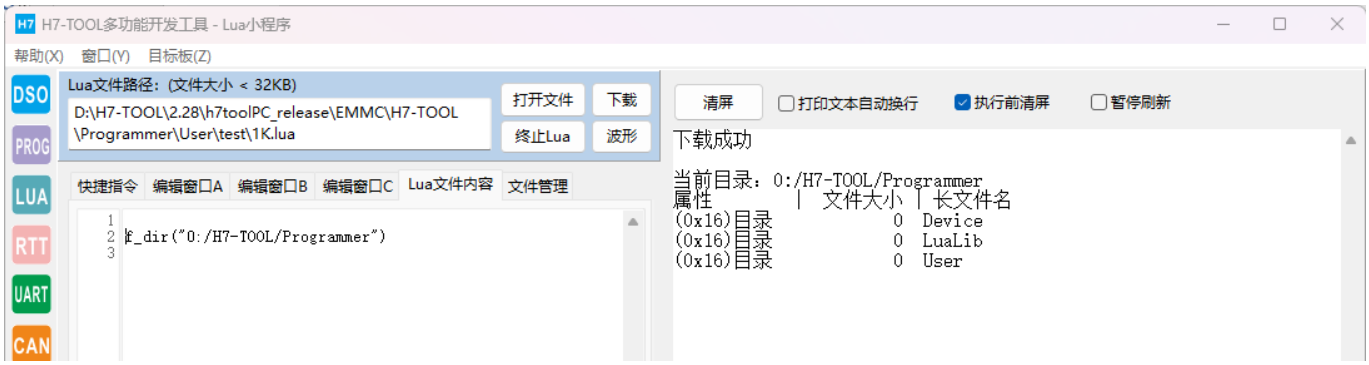
```
f_dir(path)
```

说明：

- 用于列目录文件
- path：完整的目录字符串

例子：

```
f_dir("0:/H7-TOOL/Programmer")
```



5.4.2. f_read 读文件内容

函数：

```
re = f_read(filename， offset，  read_bytes）
```

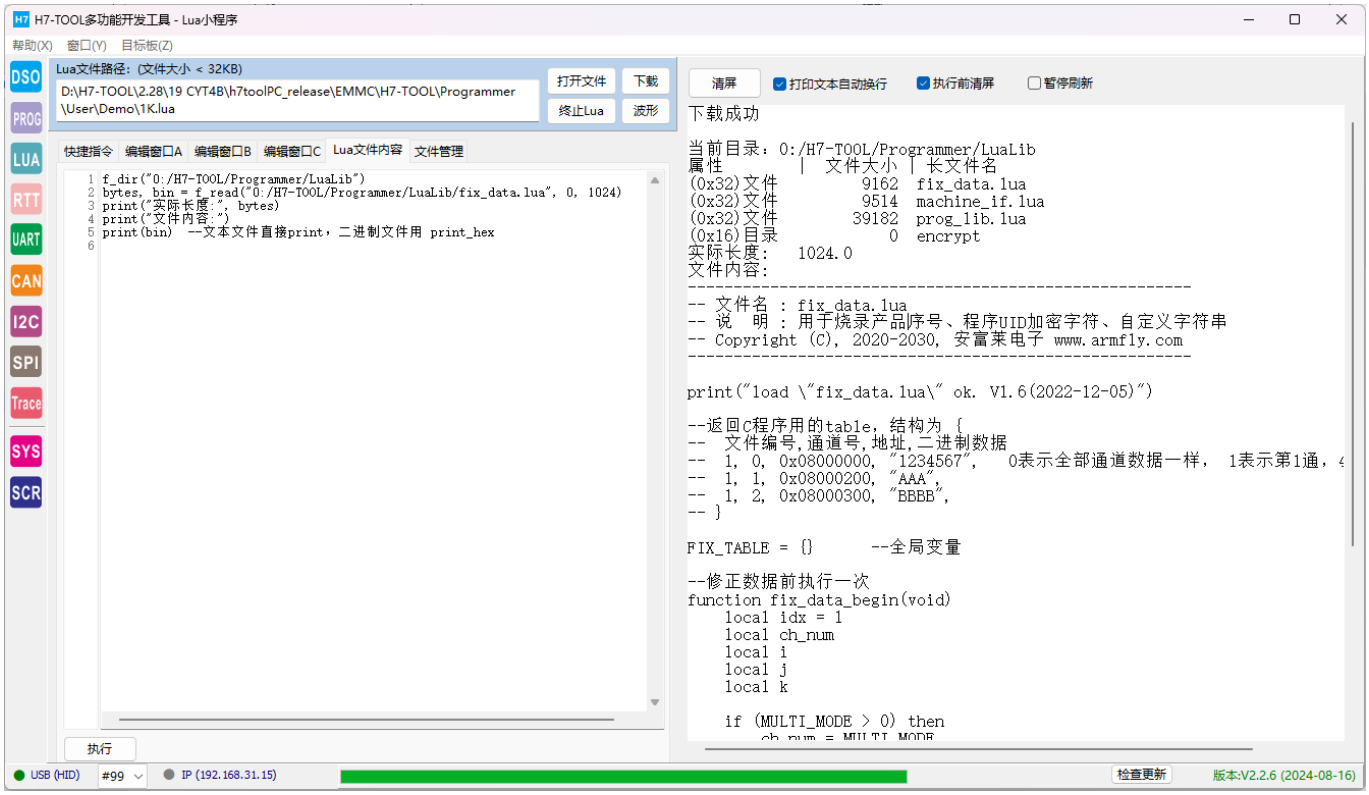
说明：

- 用于读取文件内容，一次最大读取 16KB数据。与fatfs文件系统的f_read不同的是不需要f_open，此函数已经封装了open功能。
- filename：文件路径，需要完整的路径
- offset：偏移地址
- read_bytes:读取长度
- re：返回实际读取的文件长度

例子：

查看 fix_data.lua 文件的内容：

```
f_dir("0:/H7-TOOL/Programmer/LuaLib")
bytes, bin = f_read("0:/H7-TOOL/Programmer/LuaLib/fix_data.lua", 0, 16 * 1024)
print("实际长度:", bytes)
print("文件内容:")
print(bin)  --文本文件直接 print，二进制文件用 print_hex
```



5.4.3. f_write 写文件内容

函数：

```
re = f_read(filename， offset， data)
```

说明：

- 用于写文件内容，一次最大读取 16KB数据。对于不存在的文件，此函数会创建对应的文件。与fatfs文件系统的f_write不同的是不需要f_open，此函数已经封装了open功能。
- filename：文件路径，需要完整的路径
- offset：偏移地址
- data：待写入的文件数据（可以是ascii字符串、二进制数组（包括 0x00））
- re：返回值 1 表示成功，0 表示失败，如果TOOL磁盘已加密。则会直接返回错误。
  磁盘加密后需要先执行 lua_f_password("123456") --输入密码
  密码出错 5 次，则禁止文件访问了，需要重新上电。

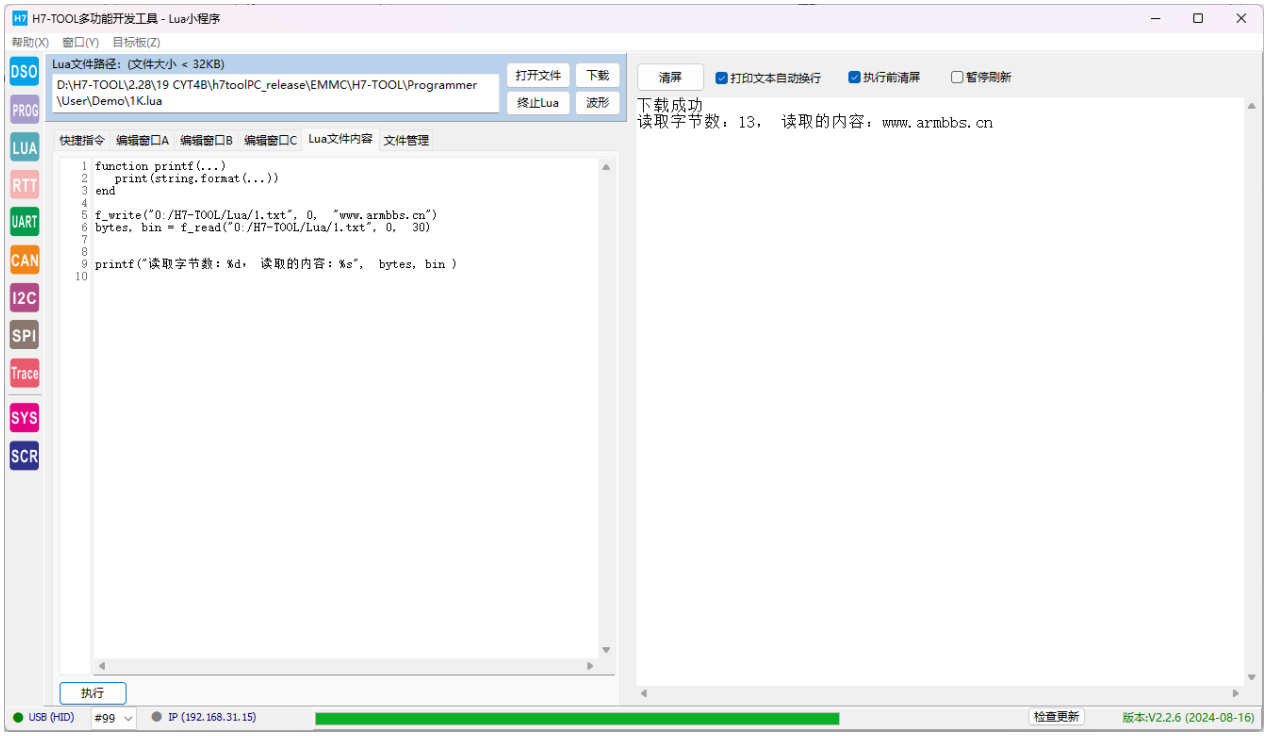
例子：

创建一个 1.txt 文件，写入内容 www.armbbs.cn，再读出来

```
function printf(...)
   print(string.format(...))
end

f_write("0:/H7-TOOL/Lua/1.txt", 0, "www.armbbs.cn")
bytes, bin = f_read("0:/H7-TOOL/Lua/1.txt", 0, 30)


printf("读取字节数：%d， 读取的内容：%s", bytes, bin )
```



5.4.4. f_size 获取文件大小

函数：

```
re = f_size(filename)
```

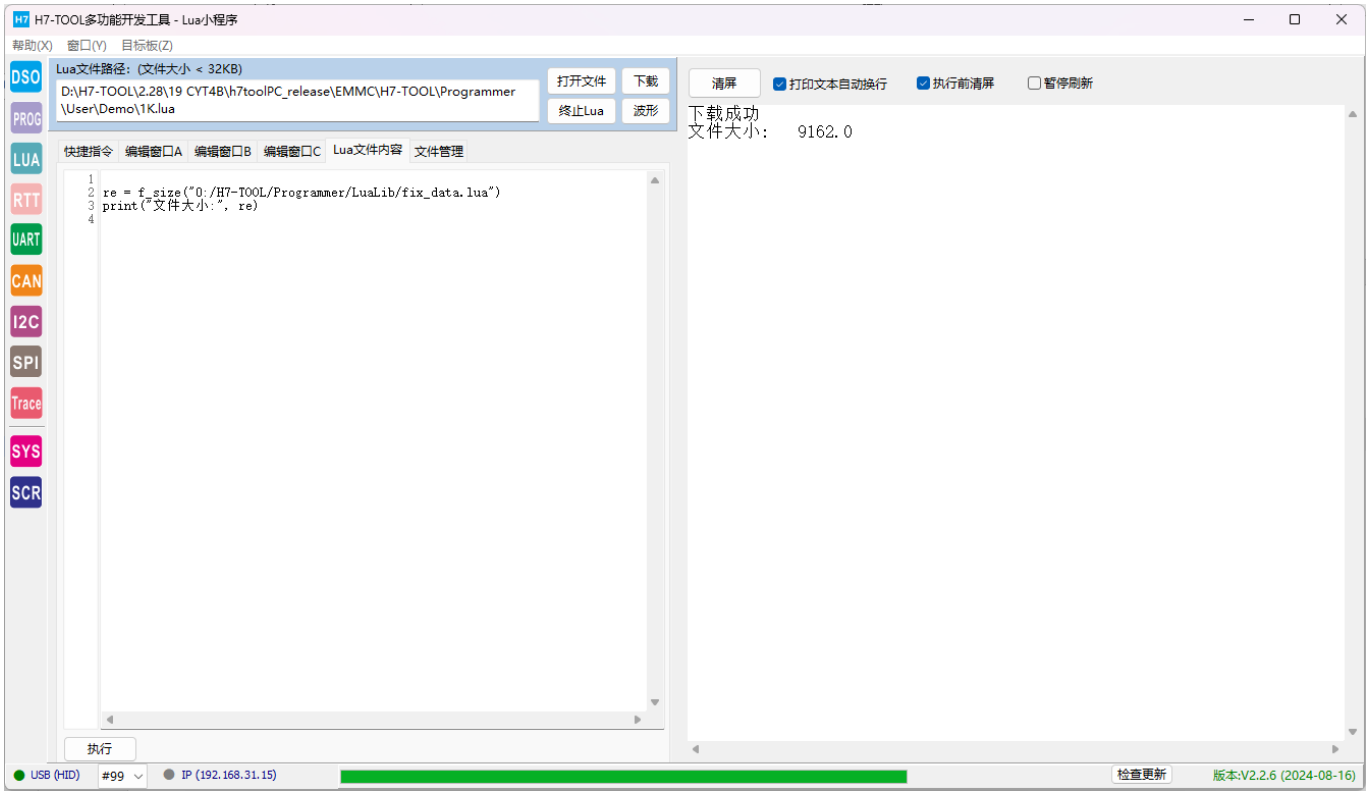
说明：

- 用于获取文件大小。
- filename：文件路径，需要完整的路径

例子：

比如获取文件 fix_data.lua 的大小：

```
re = f_size("0:/H7-TOOL/Programmer/LuaLib/fix_data.lua")
print("文件大小:", re)
```



5.4.5. 范例 1：离线串口日志

存储到 TOOL 的 4GB 容量 eMMC 磁盘

https://www.armbbs.cn/forum.php?mod=viewthread&tid=122322

5.4.6. 范例 2：单片机 bootloader 的 CANFD，I2C，SPI 和串口方式更新 APP

https://www.armbbs.cn/forum.php?mod=viewthread&tid=114491

5.5. GPIO 10 路 IO 控制输入输出

5.5.1. gpio_cfg 通用 GPIO 配置

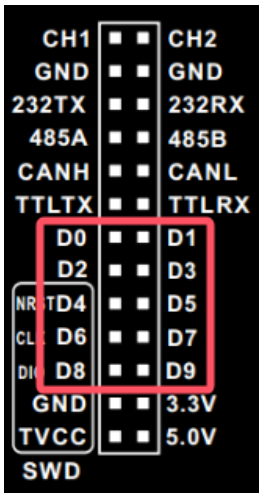
5.5.2. gpio_write GPIO 写操作

5.5.3. gpio_read GPIO 读操作

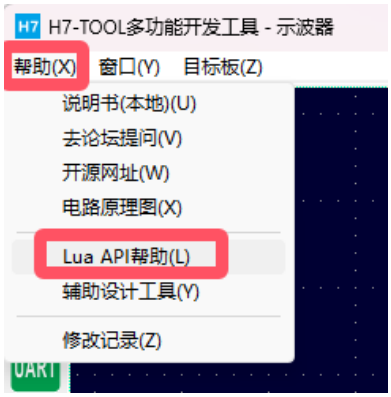
这三个函数涉及到的配置项比较多，不做具体介绍了，可以使用上位机带的一个小软件直接配置生成

说明：

用到的 D0 -D9 引脚是指的这几个引脚：
注意 TVCC 电平决定 D0 - D9 电平范围，支持 1.2V - 5.0V



例子：
比如配置 D2 输出高电平



配置 D2 输出：



配置输出高电平：

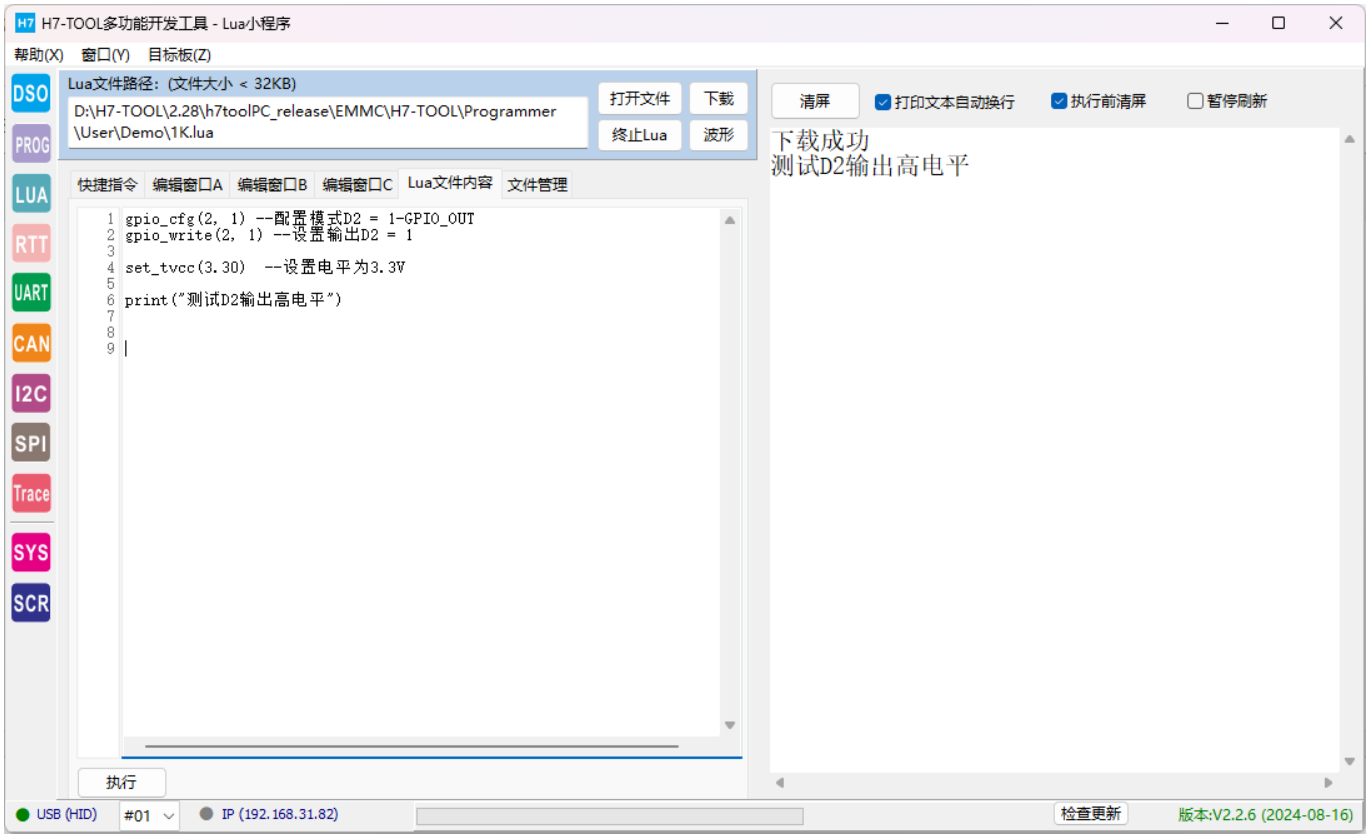


测试代码：

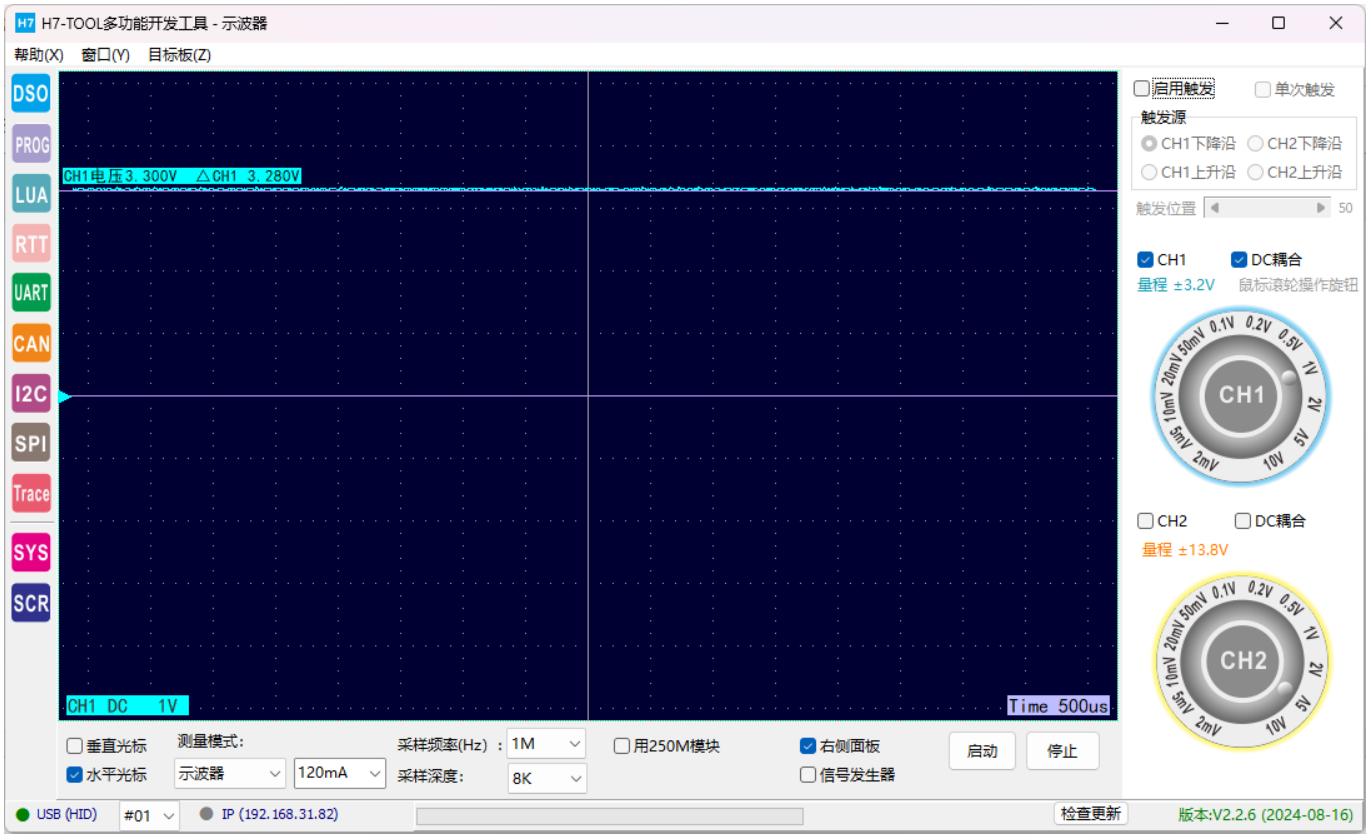
```
gpio_cfg(2,  1)  --配置模式 D2 = 1-GPIO_OUT
gpio_write(2,  1)  --设置输出 D2 = 1

set_tvcc(3.30)     --设置电平为 3.3V

print("测试 D2 输出高电平")
```



示波器测量 D2 输出高电平



5.6. PWM

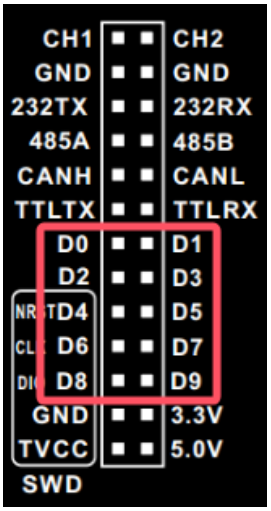
5.6.1. gpio_pwm_out 4 路 PWM 输出

函数：

```
gpio_pwm_out(pin, freq, duty)
```

说明：

- pin : 端口号，0、1、3、4，分别对应硬件的上的D0，D1，D3 和D4 端口



- freq : 频率，Hz单位，整数
- duty : 占空比，百分比，0 - 100.0%
- 推挽输出，电平由TVCC电压决定（可设置），四路共用的一个TIM，因此频率必须相同。占空比可以不同。

用法：

```
f_dir("0:/H7-TOOL/Programmer")
gpio_pwm_out(0, 1000, 50.00)       --D0 口输出 1KHz，占空比 50.00%的方波
gpio_pwm_out(1, 1000, 40.00)       --D1 口输出 1KHz，占空比 40.00%的方波
gpio_pwm_out(3, 1000, 30.00)       --D3 口输出 1KHz，占空比 30.00%的方波
gpio_pwm_out(4, 1000, 10.00)       --D4 口输出 1KHz，占空比 10.00%的方波

gpio_pwm_out(0, 0, 0)              --D0 口 PWM 关闭，输出 0 电平
gpio_pwm_out(1, 0, 0)              --D1 口 PWM 关闭，输出 0 电平
gpio_pwm_out(3, 0, 100)        --D3 口 PWM 关闭，输出 1 电平
gpio_pwm_out(4, 0, 100)        --D4 口 PWM 关闭，输出 1 电平

set_tvcc(3.30)     --设置电平为 3.3V
```

如果需要小于 1Hz 的波形，要在上位机设置，支持范围 0.05Hz 到 40MHz



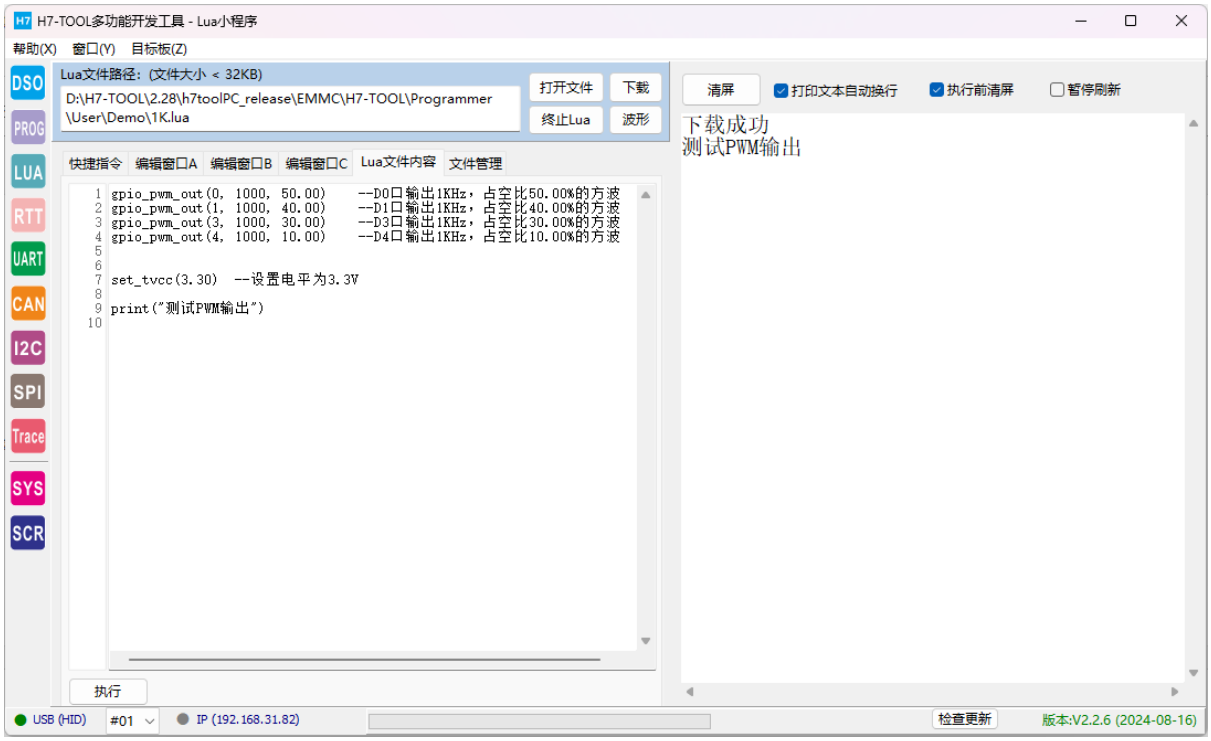
例子：

D0,D1，D3 和 D4 都输出 1KHz，占空比分别是 50%，40%，30%，20%，10%

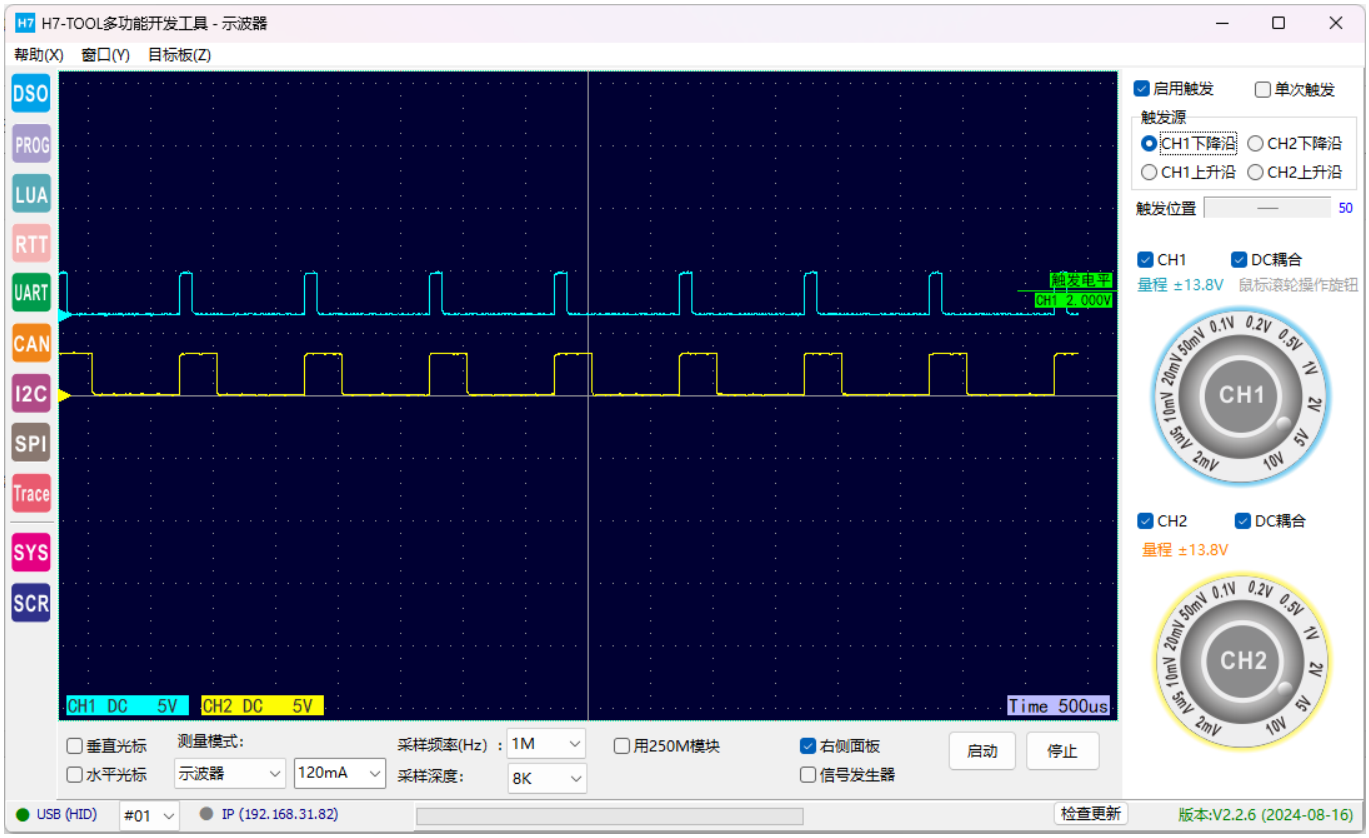
```
gpio_pwm_out(0,   1000,   50.00)          --D0 口输出 1KHz，占空比 50.00%的方波
gpio_pwm_out(1,   1000,   40.00)          --D1 口输出 1KHz，占空比 40.00%的方波
gpio_pwm_out(3,   1000,   30.00)          --D3 口输出 1KHz，占空比 30.00%的方波
gpio_pwm_out(4,   1000,   10.00)          --D4 口输出 1KHz，占空比 10.00%的方波

set_tvcc(3.30)     --设置电平为 3.3V

print("测试 PWM 输出")
```



示波器测试 D3 和 D4 的输出效果：


H7-TOOL多功能开发工具 - 示波器
帮助(X)
窗口(Y)
目标板(Z)
DSO
PROG
LUA
RTT
UART
CAN
I2C
SPI
Trace
SYS
SCR
启用触发
单次触发
触发源
CH1下降沿
CH2下降沿
CH1上升沿
CH2上升沿
触发位置
50
CH1
DC耦合
量程 ±13.8V
鼠标滚轮操作旋钮
CH1
CH2
DC耦合
量程 ±13.8V
CH2
触发电平
CH1 2.000V
CH1 DC 5V
CH2 DC 5V
Time 500us
垂直光标
水平光标
测量模式:
示波器
120mA
采样频率(Hz) :
1M
采样深度:
8K
用250M模块
右侧面板
信号发生器
启动
停止
USB (HID)
#01
IP (192.168.31.82)
检查更新
版本:V2.2.6 (2024-08-16)

5.7. I2C 总线主机

5.7.1. 功能特性

1、主机最高速度 800KHz。
2、IO 电平范围支持 1.2V-5.0V。
3、支持波形展示

5.7.2. 硬件接线

I2C 控制器引脚定义
D9 : I2C_SCL, 开漏内部集成上拉电阻 2K 欧
D7 : I2C_SDA, 开漏内部集成上拉电阻 2K 欧



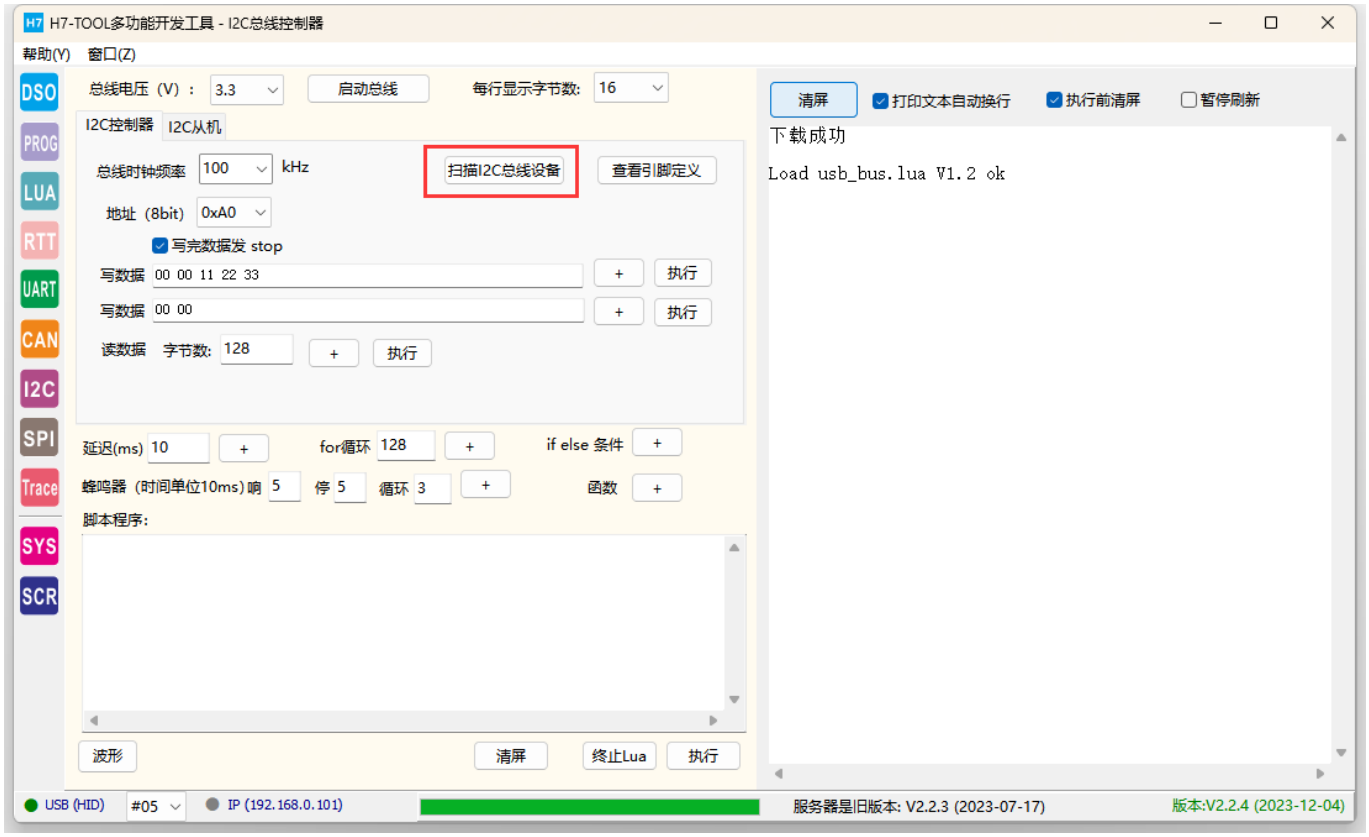
5.7.3. i2c_bus("init", 参数) 主机初始化

函数：

```
i2c_bus("init", 参数 1)
```

说明：

- 参数 1：表示I2C速度，单位Hz，最高 800KHz
- 此函数可以单独LUA调用，也可以在I2C控制器界面调用
  上位机页面对应的是”扫描总线设备“按钮：



此按钮对调用的函数对 i2c_bus("init", 参数)做了封装，原始定义位于上位机软件包路径 h7toolPC_release\USBBus 里面的 usb_bus.lua 文件中

5.7.4. i2c_bus("start") 启动函数

5.7.5. i2c_bus("stop") 停止函数

函数：

```
i2c_bus("start")
i2c_bus("stop")
```

说明：

- 这两个函数可以单独LUA调用，也可以在I2C控制器界面调用，不过I2C控制器界面对这两个函数进行了封装，由函数I2C_ReadData和I2C_WriteData调用，这两个函数的实现在上位机软件包路径h7toolPC_release\USBBus里面的usb_bus.lua文件中

5.7.6. i2c_bus("send", 参数) 发送函数

函数：

```
返回值 = i2c_bus("send",参数 1)
```

说明：

- 参数 1：表示要发送的地址或者数据。如果是发送地址，表示 8bit地址，高 7bit地址+1bit读写控制位，读写控制位 = 0 表示写操作，=1 表示读操作
- 返回值：返回 0 表示应答，返回 1 表示无应答。

操作说明：

1、 比如对地址 0x20 进行写操作，就是i2c_bus("send", 0x20|0)，而读操作的话，就是i2c_bus("send", 0x20|1)

2、 发送数据可以直接发送字符串，比如i2c_bus("send", ”123456789“)

发送实际数值可以使用转义字符i2c_bus("send", "\x00\x12\x34\x56")

上位机界面对应实现：



上位机调用的函数I2C_WriteData对函数i2c_bus("start")， i2c_bus("send", 参数) 和 i2c_bus("stop")进行了封装，原定义在上位机软件包路径h7toolPC_release\USBBus里面的usb_bus.lua文件中

5.7.7. i2c_bus("recive", 参数) 接收函数

函数：

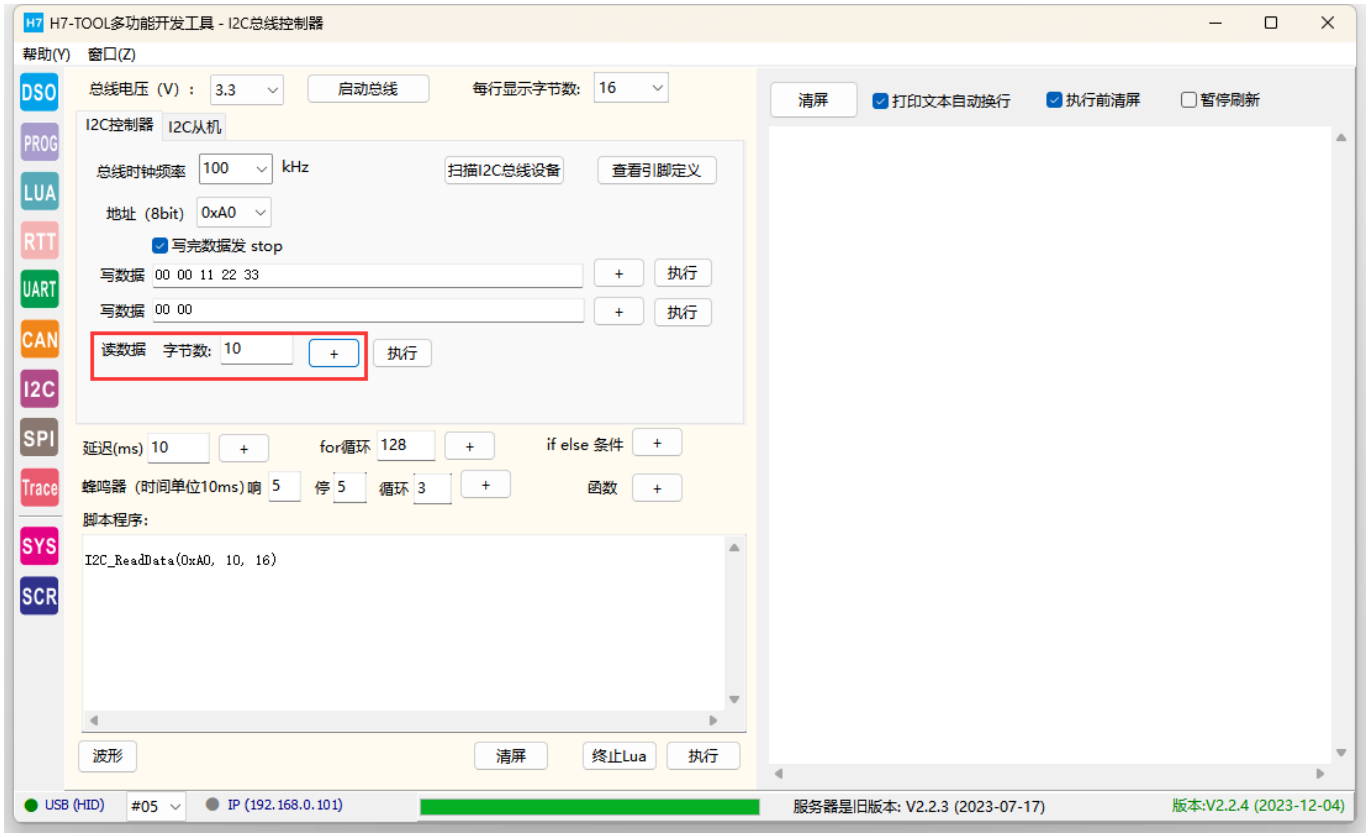
```
返回值 = i2c_bus("recive", 参数 1)
```

说明：

- 参数 1：表示读取的字节个数
- 返回值：返回的数字是字符串方式的，如果要数值方式可以使用LUA函数string.byte转换下（操作案例里面有使用）

操作说明：

比如读取 10 个字节数据就可以rd = i2c_bus("recive", 10)，rd是返回的数据，可以单独LUA函数调用，上位机操作对应实现如下：



上位机调用的函数 I2C_ReadData 对函数 i2c_bus("start"), 2c_bus("recive", 参数)和 i2c_bus("stop")进行了封装，原定义在上位机软件包路径 h7toolPC_release\USBBus 里面的 usb_bus.lua 文件中。

5.7.8. 范例 1：驱动 EEPROM 操作说明

支持 AT24C02/04/08/16/32/64/128，仅需读写两个函数，随意访问

https://www.armbbs.cn/forum.php?mod=viewthread&tid=109261

5.7.9. 范例 2：驱动光照传感器 BH1750 操作说明

https://www.armbbs.cn/forum.php?mod=viewthread&tid=109265

5.7.10. 范例 3：驱动陀螺仪 MPU6050 操作说明

https://www.armbbs.cn/forum.php?mod=viewthread&tid=109264

5.7.11. 范例 4：驱动气压传感器 BM085/BM180 操作说明

https://www.armbbs.cn/forum.php?mod=viewthread&tid=109266

5.7.12. 范例 5：驱动 18bit 分辨率 ADC 芯片 MCP3421 操作说明

https://www.armbbs.cn/forum.php?mod=viewthread&tid=109658

5.7.13. 范例 6：驱动 I2C 从机操作说明

https://www.armbbs.cn/forum.php?mod=viewthread&tid=112463

5.7.14. 范例 7：驱动 I2C 从机实现操作说明

https://www.armbbs.cn/forum.php?mod=viewthread&tid=113998

5.8. I2C 总线从机

5.8.1. 功能特性

1、从机是采用硬件 I2C 实现，开启了 Fast-mode Plus，最高支持 1MHz，从机的 I2C 速度是由主机决定的，使用时，不需要大家配置。

2、IO 电平范围支持 1.2V-5.0V。

3、从机不能主动发起操作，对从机的读写操作都是由主机发起的。

4、支持波形展示。

5.8.2. 硬件接线

I2C 控制器引脚定义

D9 : I2C_SCL, 开漏内部集成上拉电阻 2K 欧

D7 : I2C_SDA, 开漏内部集成上拉电阻 2K 欧



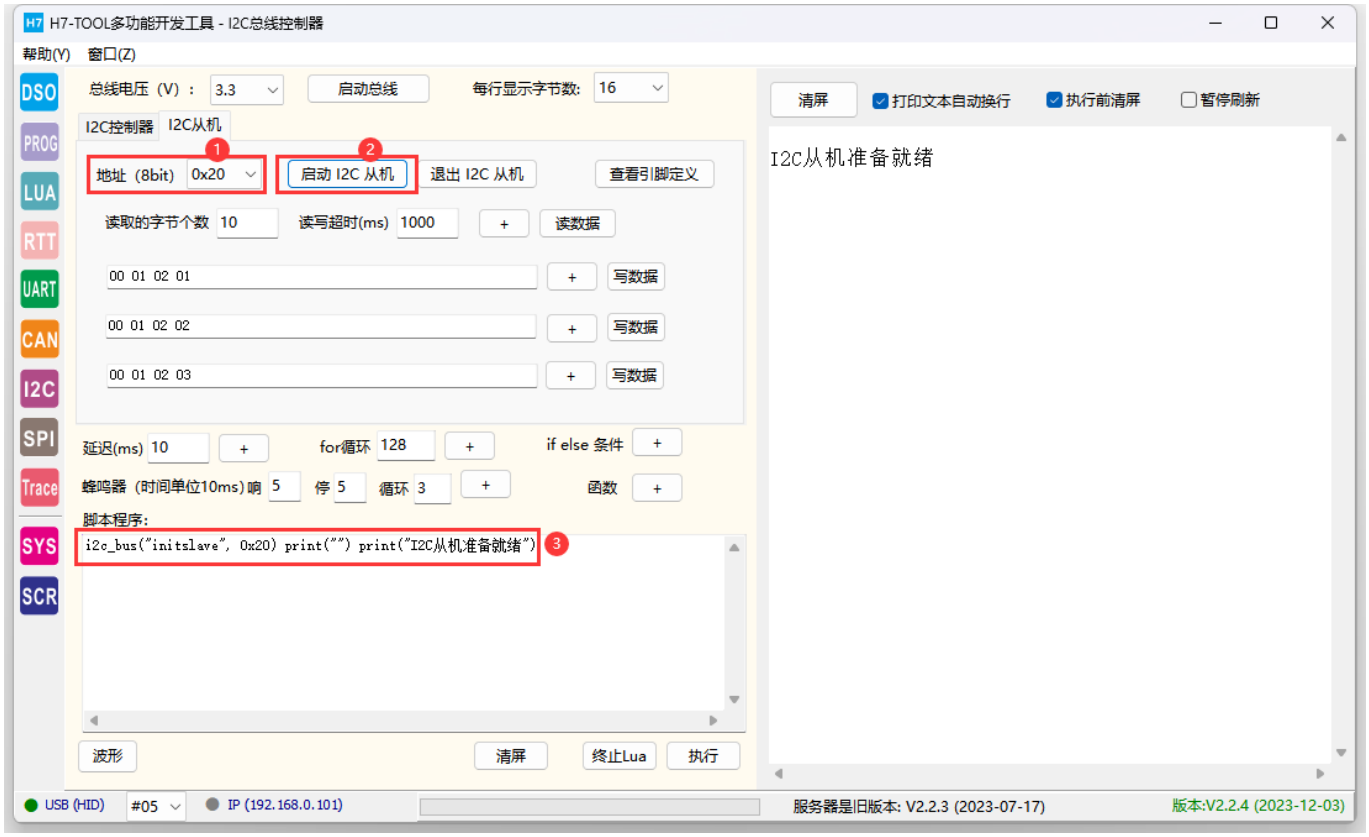
5.8.3. i2c_bus("initslave", 从机地址) 初始化

函数：

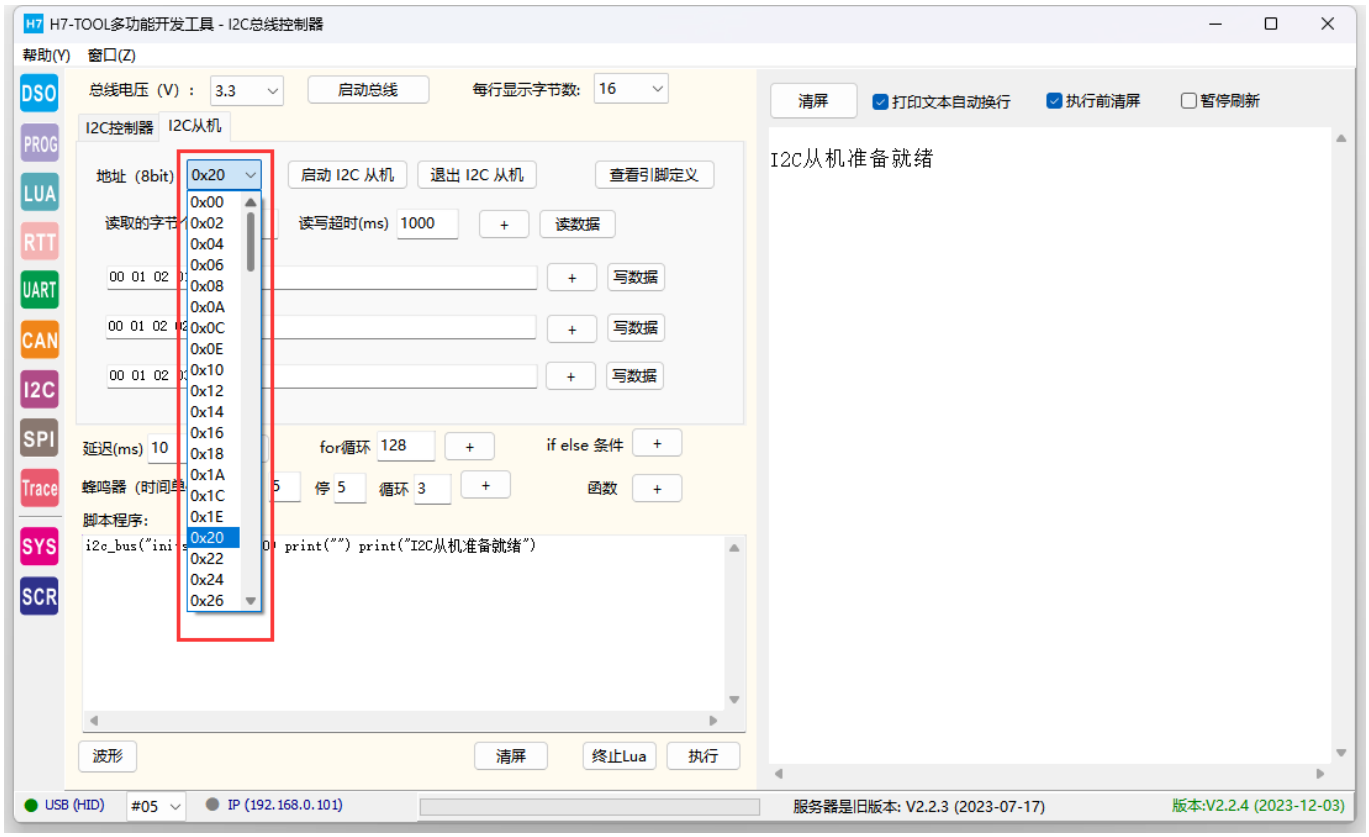
```
i2c_bus("initslave", 从机地址)
```

说明：

- 从机地址：从机地址采用 7bit模式，但这里实际填入的值是 = 7bit地址 << 1 ， 左移了一位
- 从机函数可以单独LUA调用，也可以TOOL上位机I2C从机界面控制。



大家可以看到从机地址都是偶数，就是因为左移了一位，空出来的 bit0 是 I2C 的读写控制位，硬件 I2C 会自动管理。



5.8.4. i2c_bus("reciveslave", 参数 1, 参数 2) 接收函数

函数：

返回值 1, 返回值 2 = i2c_bus("reciveslave", 参数 1, 参数 2)

说明：

- 返回值 1 ：表示实际接收到的字节数。
- 返回值 2 ：返回接收的数值
- 参数 1 ：等待接收字节数
- 参数 2 ：等待超时时间，单位ms，等待的时间内没有接收完成将立即返回

比如接收 10 个数据，超时时间 3000ms，那么就可以这样调用：len, str = i2c_bus("reciveslave", 10, 3000)



5.8.5. i2c_bus("sendslave", 参数 1, 参数 2) 发送函数

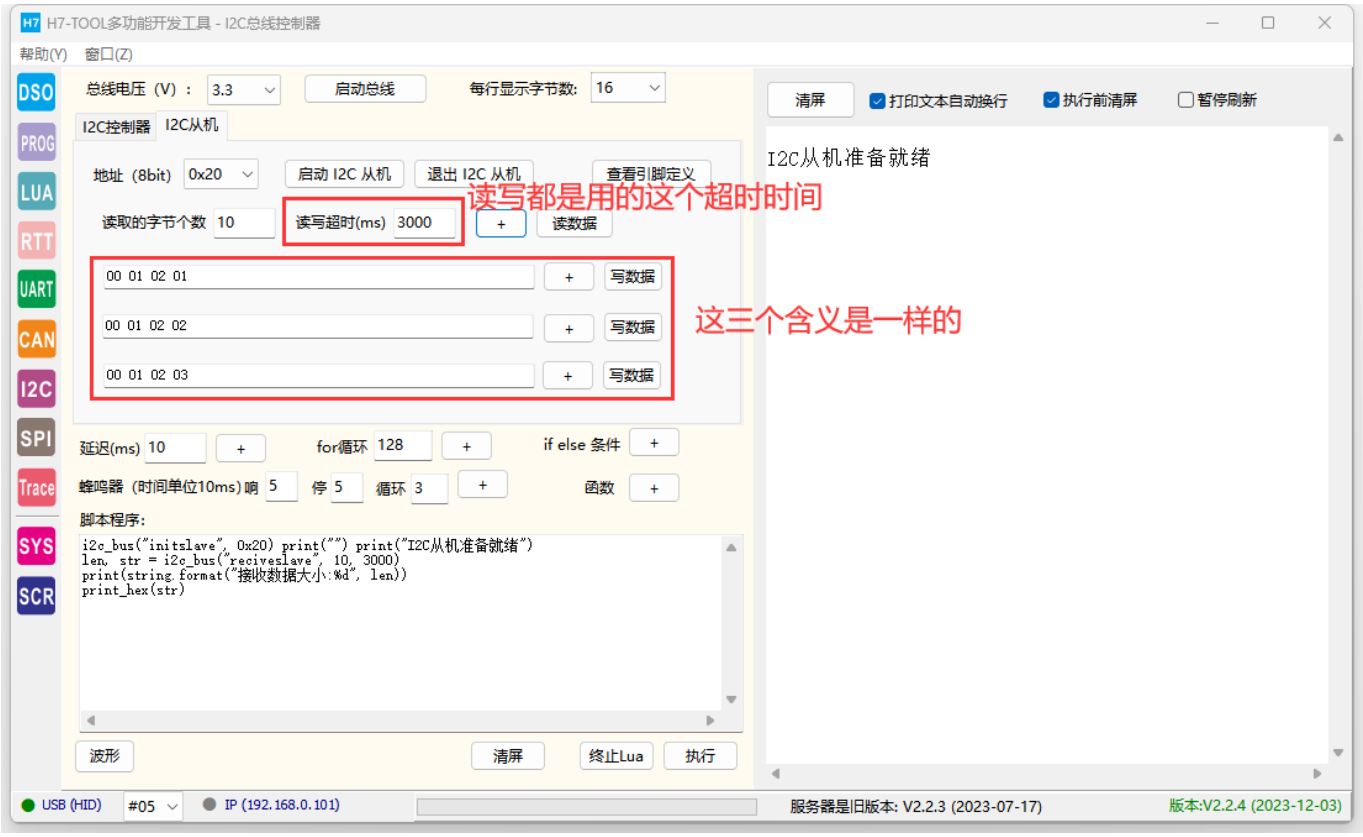
函数：

返回值 1 = i2c_bus("sendslave", 参数 1, 参数 2)

说明：

- 返回值 1 ：表示实际发送的字节数。
- 参数 1 ：等待发送的数据，可以发送字符串比如"12345678"，如果要发送实际数值，可以转义字符比如"\x00\x12\x34\x56"
- 参数 2 ：等待超时时间，单位ms，等待的时间内没有发送完成将立即返回

比如写 10 个数据，超时时间 2000ms，那么就可以这样调用：len = i2c_bus("sendslave", "0123456789", 2000)



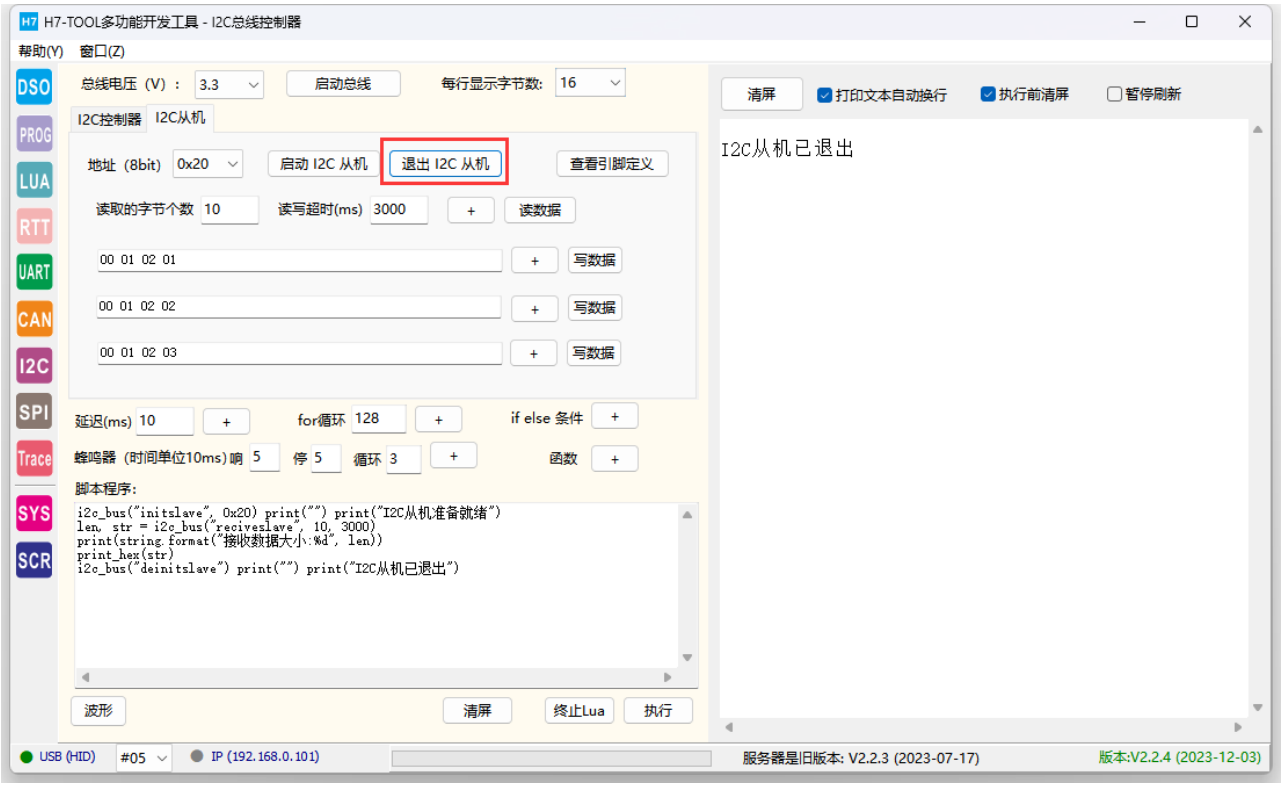
5.8.6. i2c_bus("deinitslave") 卸载函数

函数:

```
i2c_bus("deinitslave")
```

说明:

- 直接调用函数i2c_bus("deinitslave")即可，也可以上位机按钮退出



5.8.7. 范例

1、测试从机地址 0x20，主机发送给从机 10 个数据，从机等待超时时间设置为 5000ms，注意要主机发起操作，从机不能主动发起



2、测试从机地址 0x20，从机发送 10 个数据给主机，等待超时时间设置为 5000ms，注意要主机发起操作，从机不能主动发起操作



5.9. SPI 总线主机

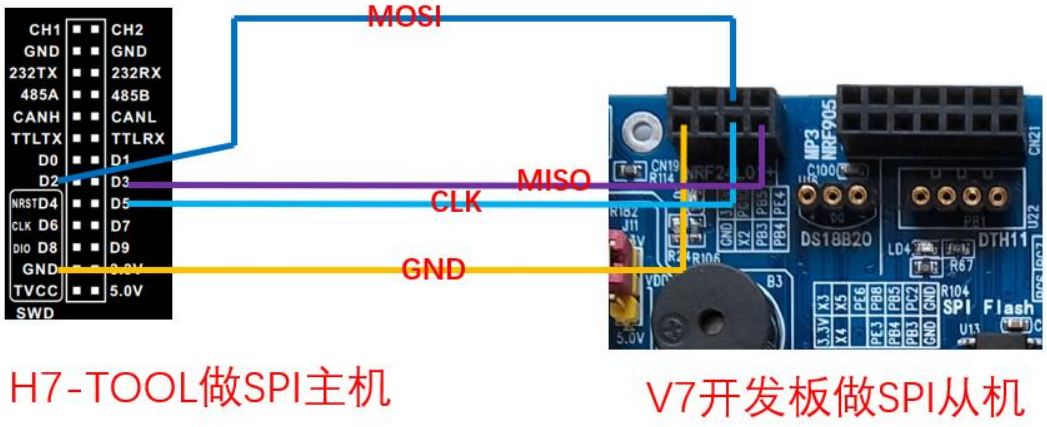
5.9.1. 功能特性

1、主机采用硬件 SPI 实现，最高速度 50MHz，TOOL 本身是支持 100MHz 的，但由于带双向 IO 电平控制，所以最高速设置为 50MHz
2、IO 电平范围支持 1.2V-5.0V。
3、支持波形展示。

5.9.2. 硬件接线

SPI 主机的引脚定义如下：
D0 ： CS0 片选 0
D1 ： CS1 片选 1
D5 ： SCK 时钟
D2 ： SDO MOSI
D3 ： SDI MISO

接目标板的话，接线方式如下，大家可以根据需要选择是否接片选，不接也可以使用。



5.9.3. spi_bus("init", 参数 1, 参数 2, 参数 3)初始化

函数：

```
spi_bus("initslave", 参数 1, 参数 2, 参数 3)
```

说明：

- 参数 1： 配置SPI速度
  0 表示速度 390.625K
  1 表示速度 781.25K
  2 表示速度 1.5625M
  3 表示速度 3.125M
  4 表示速度 6.25M
  5 表示速度 12.5M
  6 表示速度 25M
  7 表示速度 50M
- 参数 2：配置SPI时钟相位
  0 表示上升沿采集数据

1 表示下降沿采集数据

- 参数 3：时钟极性

  0 表示空闲时SCK时钟是低电平

  1 表示空闲时SCK时钟是高电平

比如配置时钟速度 50MHz，上升沿采集，SCK 空闲时低电平 ，可以单独 LUA 调用 spi_bus("init", 7, 0, 0)

也可以使用 TOOL 上位机操作：



上位机这里调用的函数是 SPI_Init(7, 0, 0)，这是因为上位机界面为了方便操作对函数 spi_bus("init", 参数 1, 参数 2, 参数 3)做了封装，原始定义位于上位机软件包路径 h7toolPC_release\USBBus 里面的 usb_bus.lua 文件中

5.9.4. spi_bus("recive", 参数) 接收函数

函数：

```
返回值 = spi_bus("recive", 参数)
```

说明：

- 返回值：读取数据
- 参数 ：要读取的数据个数

比如要读取 10 个数据，就可以 rd = spi_bus("recive", 10)，可以单独 LUA 调用也可以上位机操作



上位机这里调用的函数是 SPI_ReadData(0x00, 10, 16)，原始定义位于上位机软件包路径 h7toolPC_release\USBBus 里面的 usb_bus.lua 文件中，是对函数 spi_bus("recive", 参数)做了封装

5.9.5. spi_bus("send", 参数) 发送函数

函数：

```
spi_bus("send", 参数 1)
```

说明：

- 参数 1：发送的数据，可以发送字符串比如"12345678"，如果要发送实际数值，可以转义字符比如"\x00\x12\x34\x56"

比如要写 10 个数据，就可以 spi_bus("send", "1234567890")，可以单独 LUA 调用，也可以上位机操作



上位机这里调用的函数是SPI_WriteData(0x00, "\x00\x11\x22\x33\x44\x55\x66\x77\x88\x99")，原始定义位于上位机软件包路径h7toolPC_release\USBBus里面的usb_bus.lua文件中，是对函数spi_bus("send", 参数)做了封装

5.9.6. spi_bus("uninit") 卸载函数

函数：

```
spi_bus("uninit")
```

说明：

- 直接调用函数spi_bus("uninit")即可

5.9.7. 范例 1：SPI 控制器主从机操作说明

https://www.armbbs.cn/forum.php?mod=viewthread&tid=111864

5.9.8. 范例 2：SPI 接口方式脱机烧录操作方法

https://www.armbbs.cn/forum.php?mod=viewthread&tid=114301

5.10. SPI 总线从机

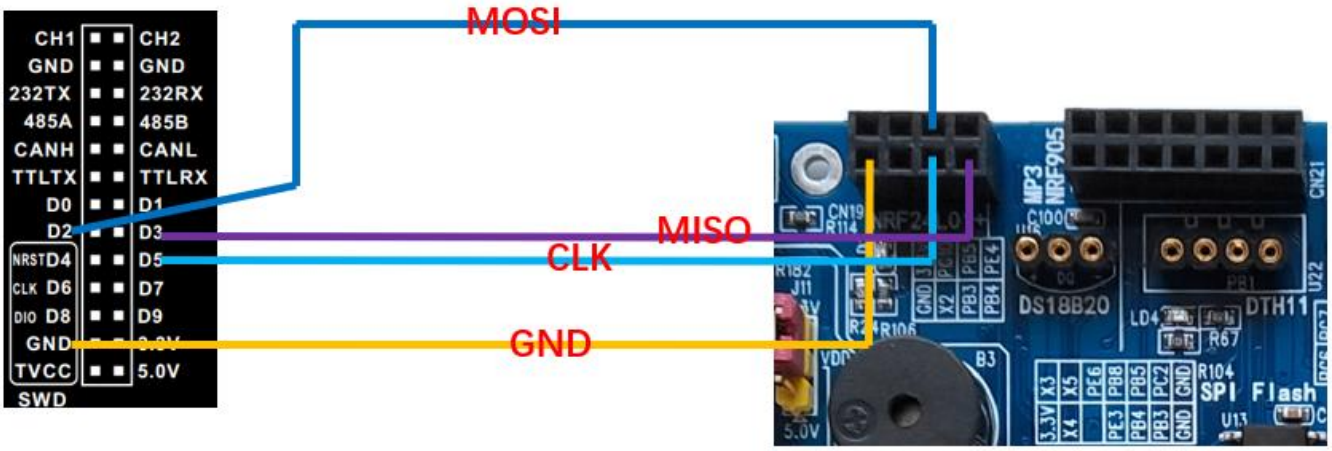
5.10.1. 功能特性

1、主机采用硬件 SPI 实现，最高速度 50MHz，TOOL 本身是支持 100MHz 的，但由于带双向 IO 电平控制，所以最高速设置为 50MHz
2、IO 电平范围支持 1.2V-5.0V。
3、支持波形展示。

5.10.2. 硬件接线

SPI 主机的引脚定义如下：
D0 ： CS0 片选 0
D1 ： CS1 片选 1
D5 ： SCK 时钟
D2 ： SDO MOSI
D3 ： SDI MISO

接目标板的话，接线方式如下，大家可以根据需要选择是否接片选，不接也可以使用。



5.10.3. spi_bus("init", 参数 1, 参数 2, 参数 3) 初始化

函数：

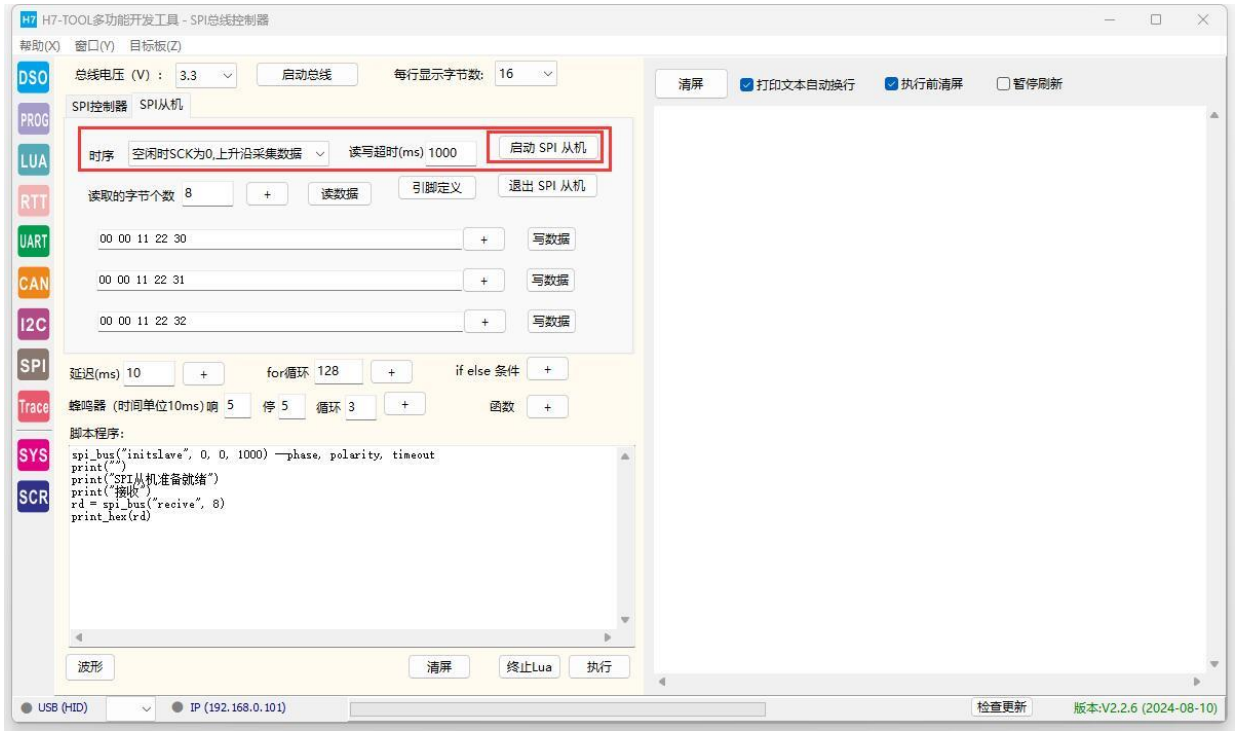
```
spi_bus("initslave", 参数 1, 参数 2, 参数 3)
```

说明：

- 参数 1：配置SPI时钟相位
  0 表示上升沿采集数据
  1 表示下降沿采集数据
- 参数 2：配置SPI时钟极性
  0 表示空闲时SCK时钟是低电平
  1 表示空闲时SCK时钟是高电平
- 参数 3：读写超时时间，单位ms，对从机的读函数和写函数起作用，表示在设置的时间内，SPI从机没有完成数据发送或者接收，将立即返回。

  比如配置上升沿采集，SCK空闲时低电平，读写超时时间 5000ms，可以单独LUA调用spi_bus("initslave", 0, 0, 5000)

也可以使用TOOL上位机操作：



5.10.4. spi_bus("recive", 参数 1) 接收函数

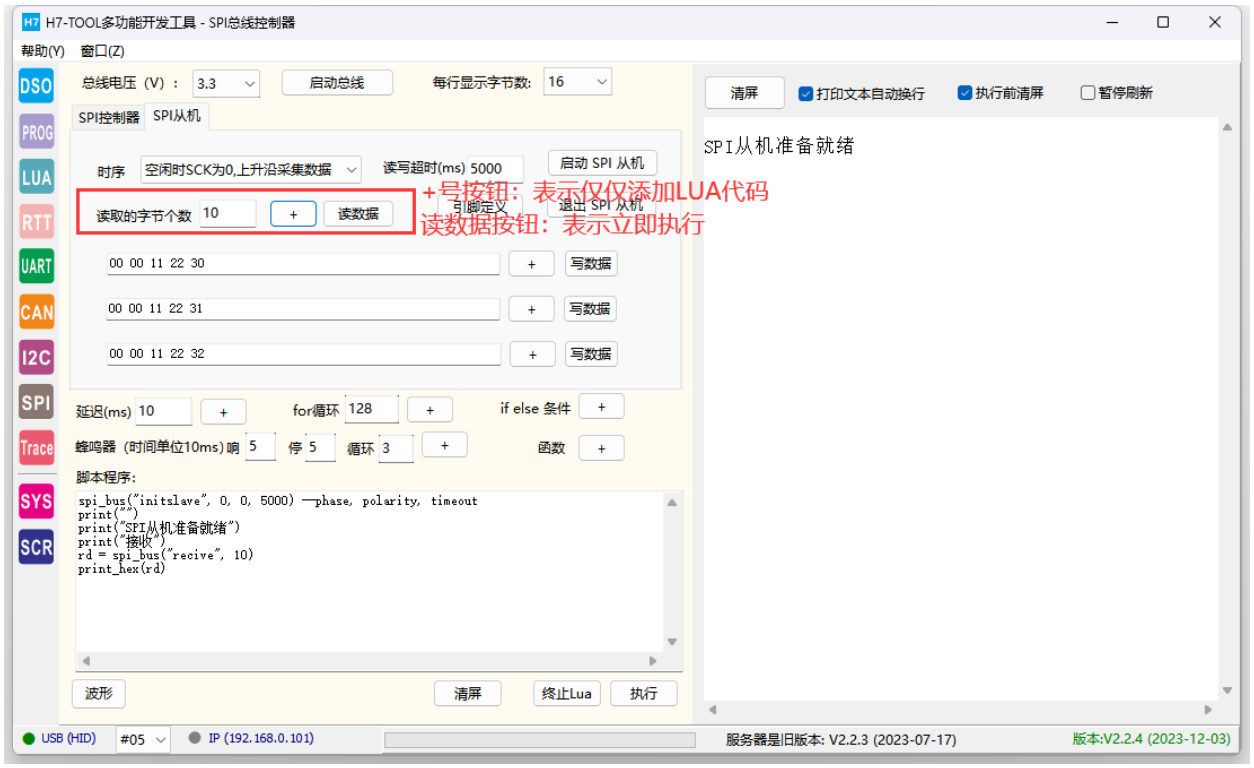
函数：

```
返回值 = spi_bus("recive", 参数 1)
```

说明：

- 返回值：读取数据
- 参数 1：要读取的数据个数

比如要读取 10 个数据，就可以 rd = spi_bus("recive", 10)，可以单独 LUA 调用，也可以上位机操作



5.10.5. spi_bus("send", 参数 1) 发送函数

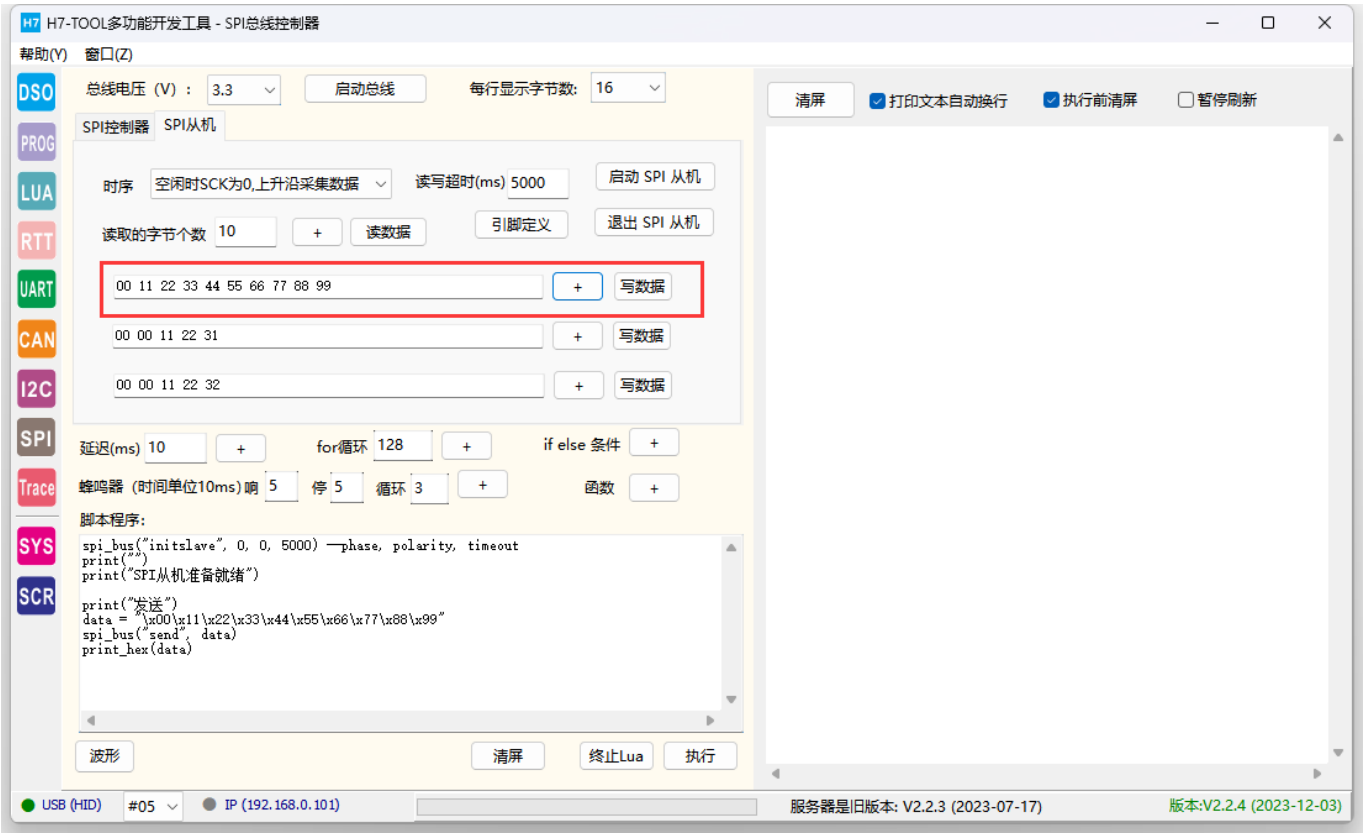
函数：

```
spi_bus("send", 参数 1)
```

说明：

- 参数 1：发送的数据，可以发送字符串比如"12345678"，如果要发送实际数值，可以转移字符比如"\x00\x12\x34\x56"

比如要写 10 个数据，就可以spi_bus("send", "1234567890")，可以单独LUA调用，也可以上位机操作



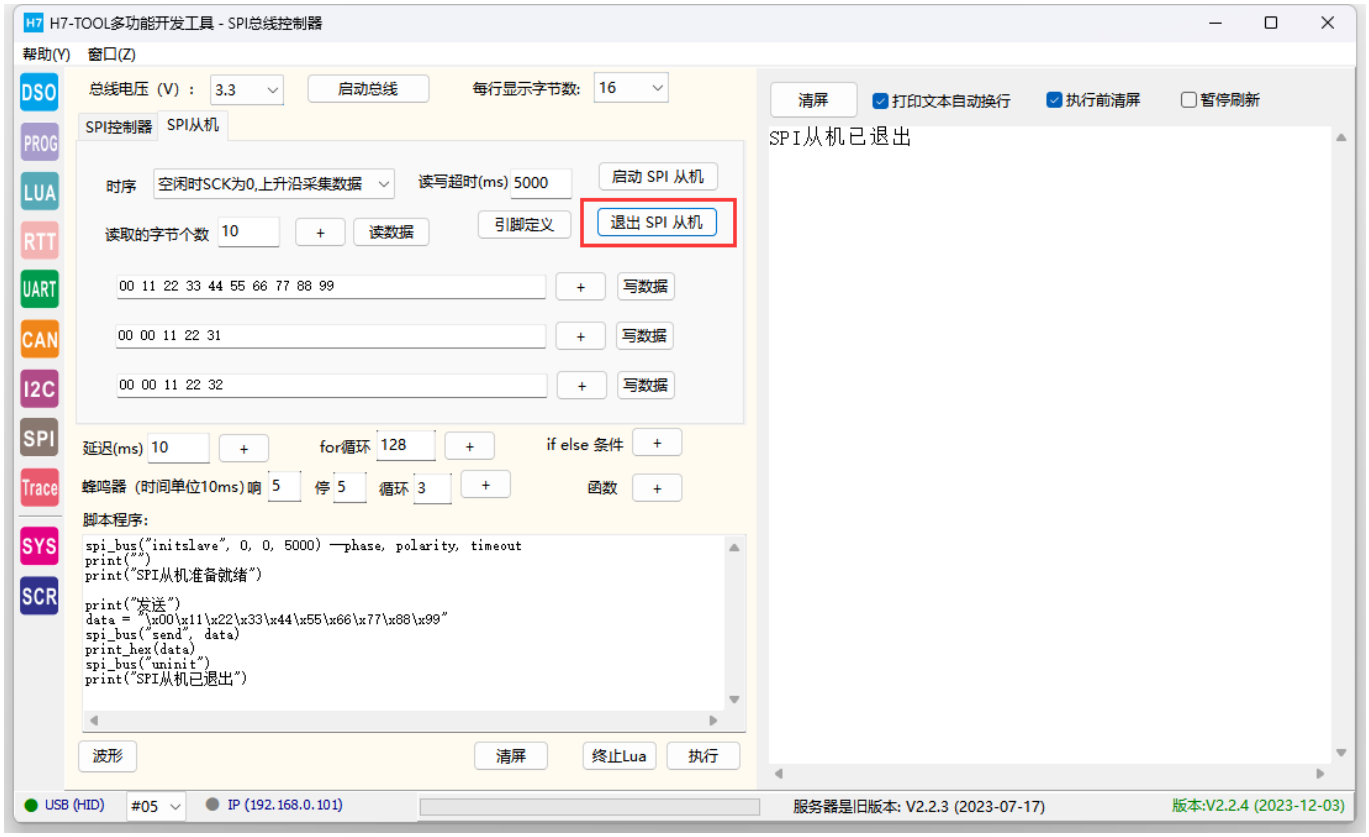
5.10.6. spi_bus("uninit") 卸载函数

函数：

```
spi_bus("uninit")
```

说明：

- 直接调用函数spi_bus("uninit")即可，也可以上位机操作：

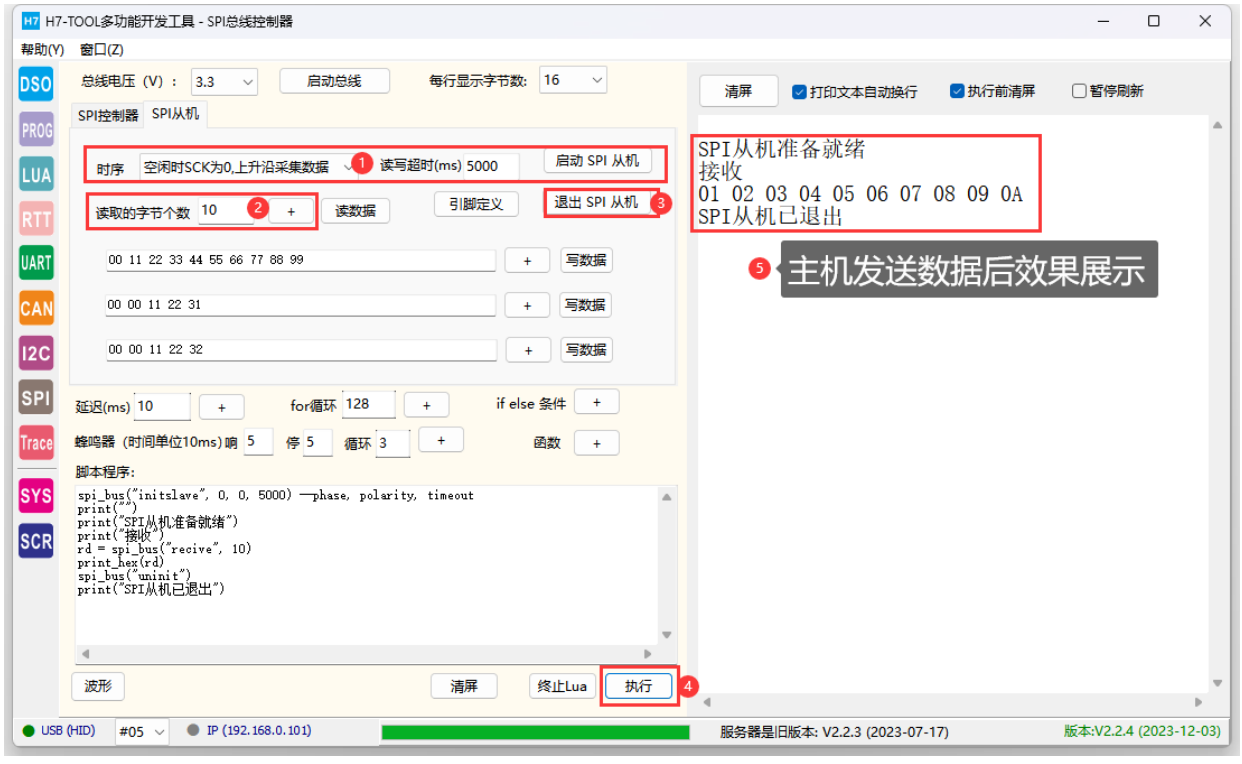


5.10.7. 范例 1：SPI 控制器做从机，开发板做主机的实际通信案例

https://www.armbbs.cn/forum.php?mod=viewthread&tid=126135

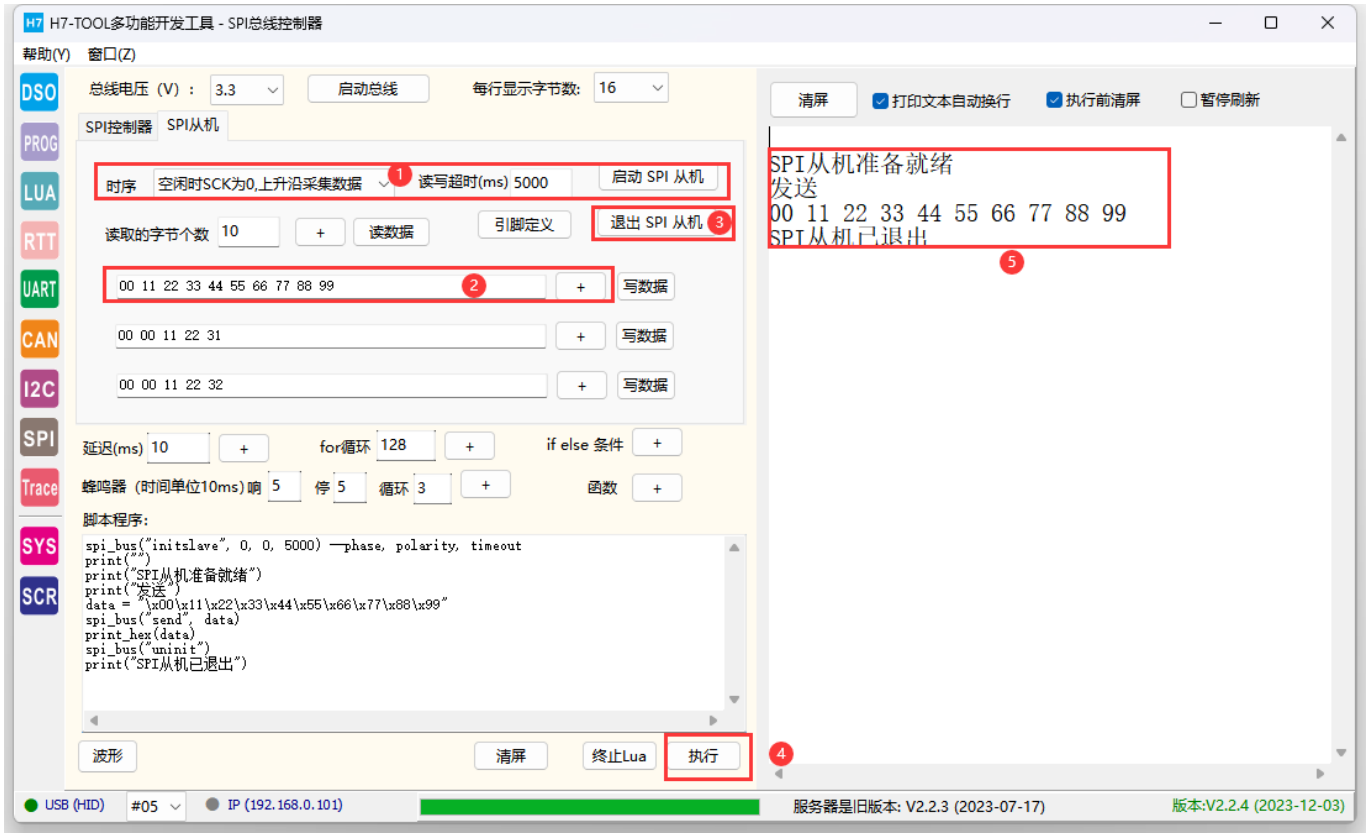
5.10.8. 范例 2：简易接收操作

配置上升沿采集数据，空闲时 SCK 是低电平，等待超时时间是 5000ms，接受主机发送 10 个字节数据，注意要主机发起操作，从机不能主动发起操作



5.10.9. 范例 3：简易发送操作

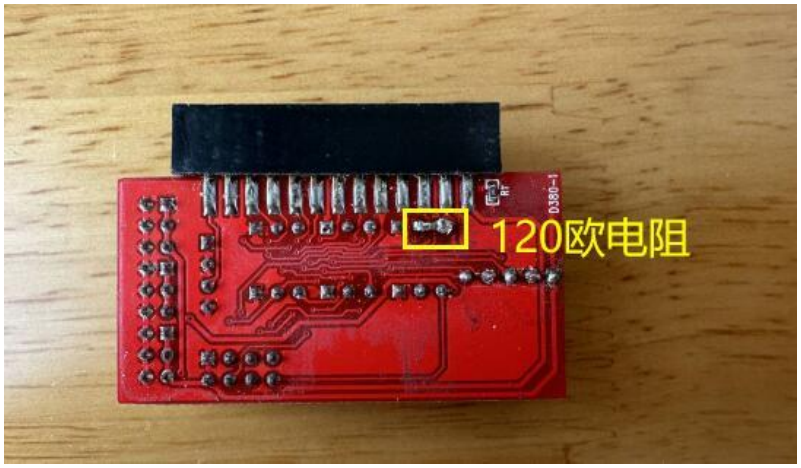
配置上升沿采集数据，空闲时 SCK 是低电平，等待超时时间是 5000ms，给主机发送 10 个字节数据，注意要主机发起操作，从机不能主动发起操作



5.11. CAN/CANFD 总线

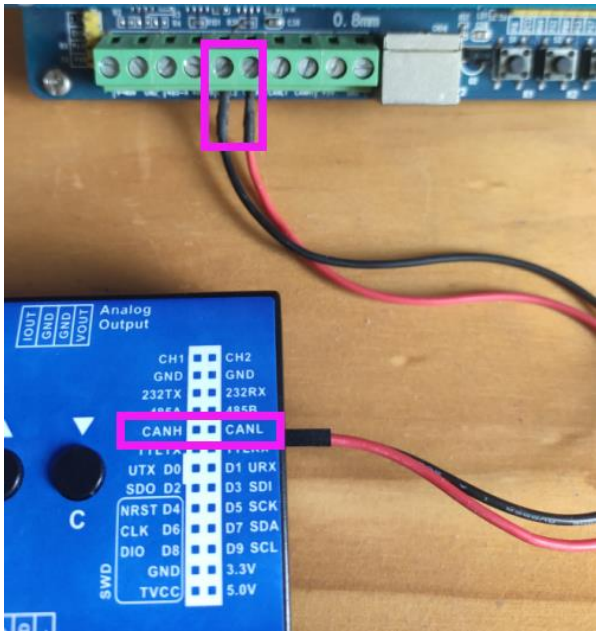
5.11.1. 功能特性

1、支持经典 CAN 和 CANFD。
2、支持 CAN 助手和 CAN LUA 同时使用，让 CAN 检测更灵活，更随心所欲。
3、可设置固定波特率 20K，25K，50K，80K，100K，125K，250K，500K，1M，2M，5M，8M。
4、可以自定义波特率，即 CAN 主频，SWJ，TS1，TS2 的参数设置。
5、支持波形展示。
6、TOOL 上电后，首界面的电压不要低于 4.7x.
7、支持 USB，以太网和 WiFi 方式以及局域网外网的上位机控制，支持离线控制运行。
8、对于经典 CAN 和低速 CANFD，至少一个节点要带终端电阻。高速 CANFD（5M，8M 测试），两端的终端电阻最好都加上。
对于 TOOL，本身是不带终端电阻的，不过可以方便在转接板背面的 CANL 和 CANH 之间焊接 120Ω 的终端电阻



5.11.2. 硬件接线

CANL 接 CANL，CANH 接 CANH



5.11.3. can_bus(”open“, 参数 2, 参数 3, 参数 4, 参数 5）初始化

函数：

```
can_bus(”open“, 参数 2, 参数 3, 参数 4,参数 5)
```

说明：

- 参数 1 ：open，close，send，recive
- 参数 2 ：0 表示FDCAN_FRAME_CLASSIC，即经典CAN
  1 表示FDCAN_FRAME_FD_NO_BRS，即CANFD，仲裁段波特率和数据段波特率一样
  2 表示FDCAN_FRAME_FD_BRS，即CANFD，仲裁段波特率和数据段波特率不一样
- 参数 3 ：支持的数据个数，可以设置 8，12，16，20，24，32，48，64
- 参数 4 ：仲裁阶段波特率
  可以设置 20K，25K，50K，80K，100K，125K，250K，500K，1M。
  注意填写的时候，填写实际数值大小，比如 500K就是 500000。
- 参数 5 ：数据阶段波特率
  可以设置 20K，25K，50K，80K，100K，125K，250K，500K，1M，2M，5M，8M。
  注意：（1）填写的时候，填写实际数值大小，比如 1M就是 1000000。
  （2）还有就是，如果设置了使用经典CAN，那么第 5 参数将不起作用。

例子：

比如设置使用经典CAN，数据个数 8，CAN速度 1M，那么配置就是

```
can_bus("open", 0, 8, 1000000, 1000000)      -- 第 5 个参数用不到，不起作用。
```

比如设置使用CANFD，数据个数 16，CAN仲裁阶段波特率 1M，通信阶段波特率 2M，那么配置就是

```
can_bus("open", 0, 8, 1000000, 2000000)
```

5.11.4. can_bus("send", 参数 2, 参数 3, 参数 4, 参数 5）发送函数

函数：

```
can_bus("send", 参数 2, 参数 3, 参数 4, 参数 5)
```

说明：

- 参数 1 ： open，close，send，recive
- 参数 2 ：ID类型，0 表示标准ID，1 表示扩展ID
- 参数 3 ：0 表示数据帧，1 表示遥控帧
- 参数 4 ：ID数值
- 参数 5 ：发送的数据，可以发送字符串比如"12345678"， 如果要发送实际数值 0，1，2，3，4，5，6，7，可以转义字符比如"\x00\x01\02\x03\x04\x05\x06\x07"

例子：

标志 ID，数据帧，ID 数值 0x222， 发送数据 0，1，2，3，4，5，6，7

```
can_bus("send", 0, 0, 0x222, "\x00\x11\x22\x33\x44\x55\x66\x77")
```

5.11.5. can_bus ("recive", 参数 2）接收函数

函数：

```
返回值 1, 返回值 2, 返回值 3, 返回值 4, 返回值 5 = can_bus ("recive", 参数 2)
```

说明：

- 返回值 1：返回的数字是字符串方式的，如果要数值方式可以使用LUA函数string.byte转换下，也可以直接调用print_hex打印出来
- 返回值 2：返回数据长度
- 返回值 3：ID数值
- 返回值 4：ID类型，0x00000000 表示数据帧，0x20000000 表示遥控帧
- 返回值 5：0 表示返回失败，1 表示返回成功。
- 参数 2：表示等待超时时间，单位ms，比如设置 5000，等待 5 秒内没有收到数据，将立即返回。

5.11.6. can_bus("close") 关闭函数

函数：

```
can_bus("close")
```

说明：

- 直接调用这个函数即可。

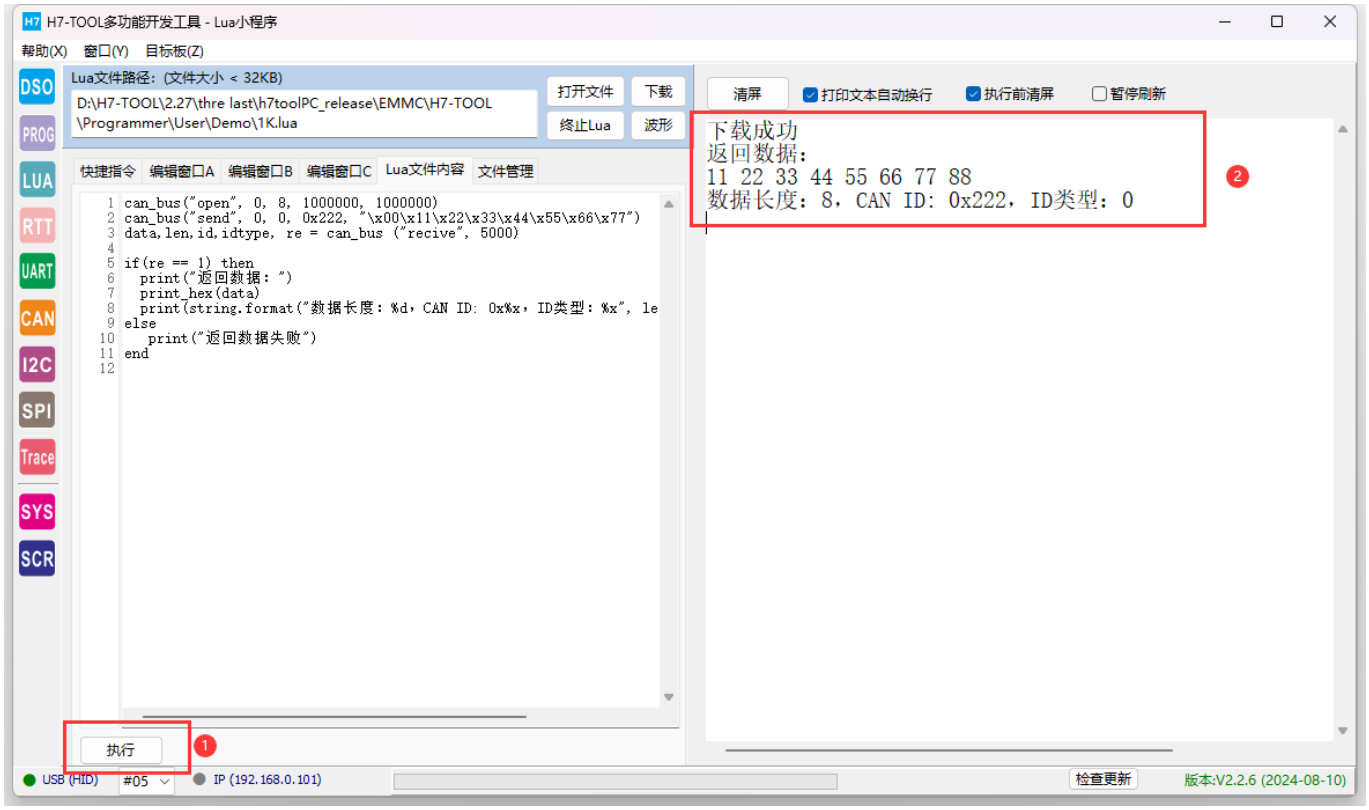
5.11.7. 范例 1：经典 CAN 通信

配置经典 CAN，1M，标准 ID 0x222，　向目标板发送数据 0x00，0x11，0x22，0x33，0x44，0x55，0x66，0x77，等待目标板返回数据超时时间 5000ms

实现代码如下：

```
can_bus("open",  0,  8,  1000000,  1000000)
can_bus("send",  0,  0,  0x222,  "\x00\x11\x22\x33\x44\x55\x66\x77")
data,len,id,idtype, re = can_bus ("recive",  5000)

if(re ==  1)  then
    print("返回数据：")
    print_hex(data)
    print(string.format("数据长度：%d，CAN ID: 0x%x，ID 类型：%x", data, len, id, idtype))
else
      print("返回数据失败")
end
```



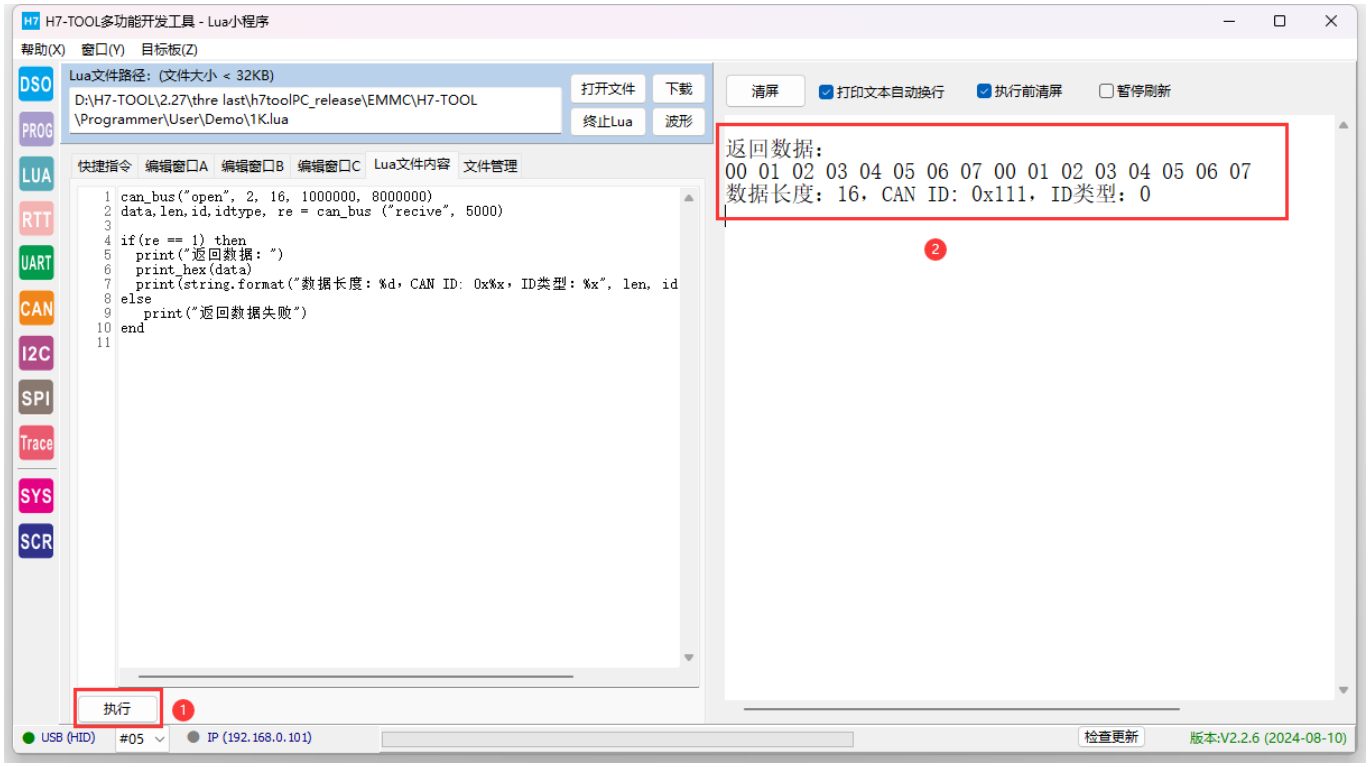
5.11.8. 范例 2：CANFD 通信

配置 CANFD，仲裁 1M+数据 2M，等待目标板返回数据超时时间 5000ms

实现代码如下：

```
can_bus("open", 2, 16, 1000000, 8000000)
data,len,id,idtype, re = can_bus ("recive", 5000)

if(re == 1) then
    print("返回数据: ")
    print_hex(data)
    print(string.format("数据长度: %d, CAN ID: 0x%x, ID 类型: %x", len, id, idtype))
else
      print("返回数据失败")
end
```



5.11.9. 范例 3：CAN Bootloader 脱机烧录

视频版：https://www.armbbs.cn/forum.php?mod=viewthread&tid=114491

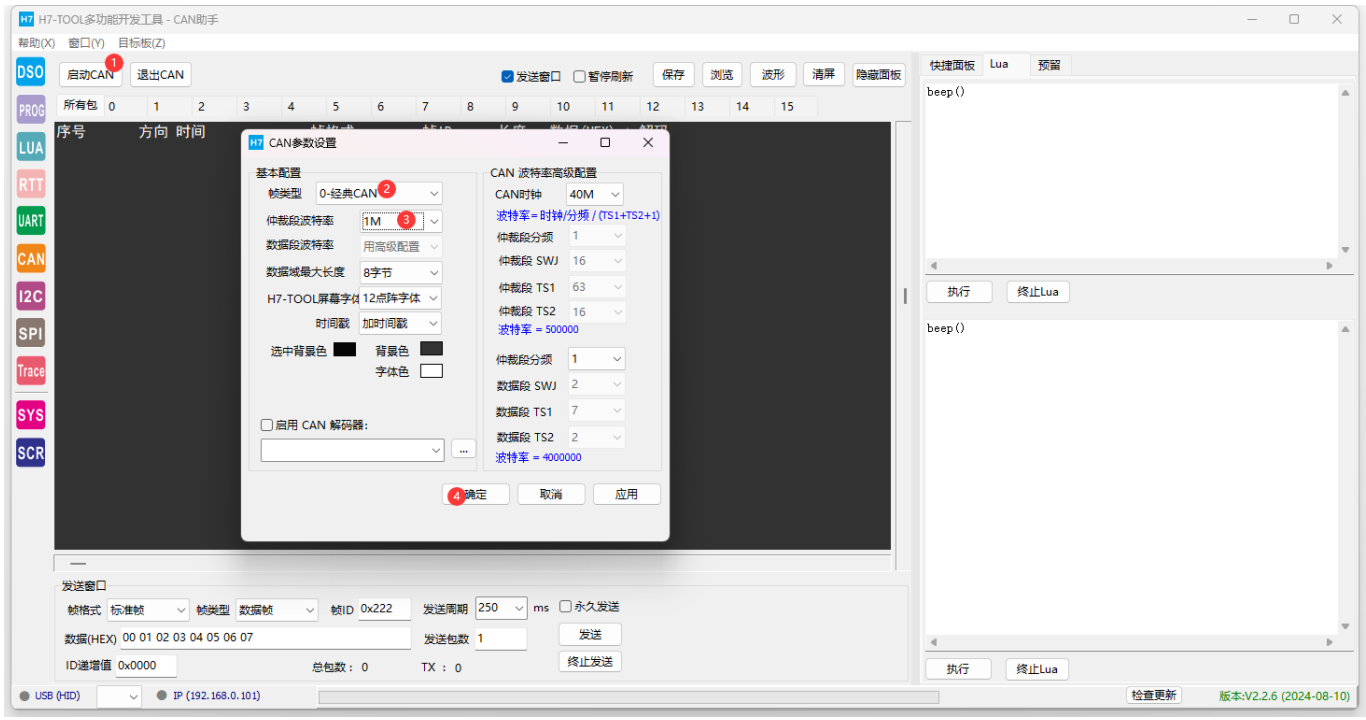
文档版：https://www.armbbs.cn/forum.php?mod=viewthread&tid=113770

5.11.10. 范例 4：CAN 助手和 LUA 同时使用

通过这种方式，更加灵活，更加随心所欲。

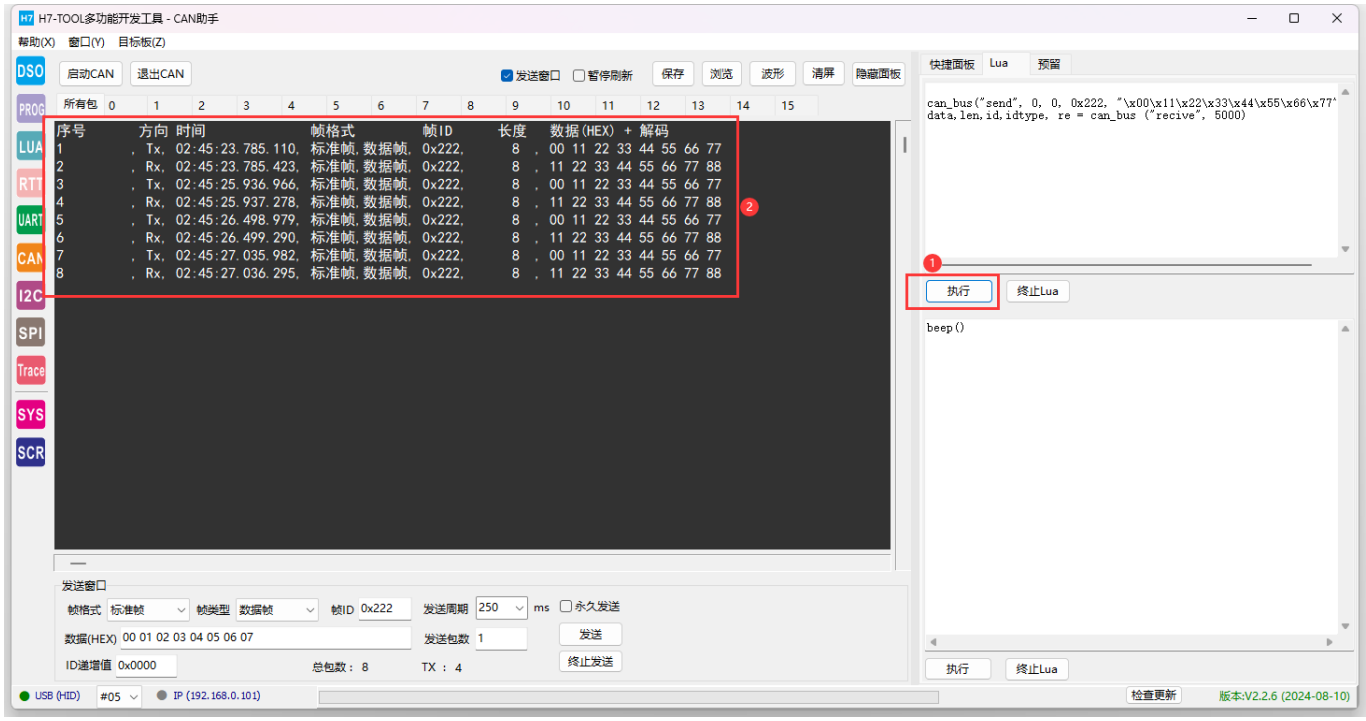
配置经典 CAN，1M，标准 ID 0x222,向目标板发送数据 0x00，0x11，0x22，0x33，0x44，0x55，0x66，0x77，等待目标板返回数据超时时间 5000ms

1、先配置：



2、添加LUA代码，仅需添加发送和接收处理即可，不需要再做初始化，因为第 1 步已经配置

```
can_bus("send", 0, 0, 0x222, "\x00\x11\x22\x33\x44\x55\x66\x77")
data,len,id,idtype, re = can_bus ("recive", 5000)
```



5.11.11. 范例 5：CANFD Trace

https://www.armbbs.cn/forum.php?mod=viewthread&tid=118999

5.11.12. 范例 6：CANopen 协议栈解析

https://www.armbbs.cn/forum.php?mod=viewthread&tid=120863

5.11.13. 范例 7：J1939 通用版解析

https://www.armbbs.cn/forum.php?mod=viewthread&tid=122456

5.11.14. 范例 8：UDS ISO14229 解析

https://www.armbbs.cn/forum.php?mod=viewthread&tid=120863

5.12. UART/485/232/Modbus

5.12.1. 注意事项

TOOL 配套了两个硬件串口，LUA 使用串口 1 没有什么要注意的，而串口 7 使用的 D0 D1, 缺省 GPIO 模式。 使用前需要配置 GPIO 模式为 UART。
gpio_cfg(1, 5) --配置模式 D1 = 5-UART
gpio_cfg(0, 5) --配置模式 D0 = 5-UART

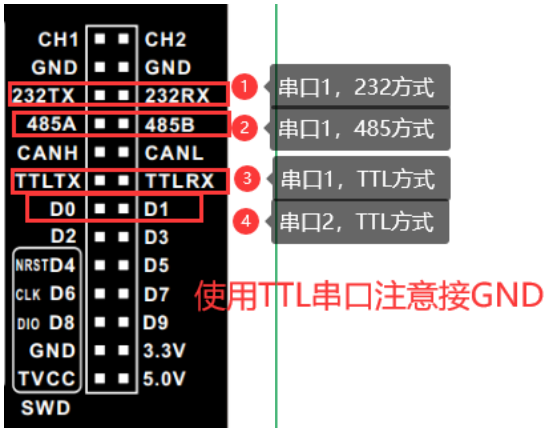
5.12.2. uart_cfg 配置串口

函数：

```
uart_cfg(port, BaudRate, Parity, WordLength, StopBits)
```

说明：

- port ： 串口号, 1 或 7
  串口号 1 对应的串口 1，串口号 7 对应的串口 2



- BaudRate ： 波特率
- Parity ： 0-无校验 1-奇校验 2-偶校验
- WordLength ： 字长 7 或者 8
- StopBits ： 停止位 1,2

例子：

比如设置串口 1，波特率 115200, 无校验，8bit，1 个停止位

```
uart_cfg(1, 115200, 0, 8, 1)
```

5.12.3. uart_send 发送数据

函数：

```
uart_send(port, bin)
```

说明：

- port： 串口号,1 或 7
- bin： 二进制字符串

例子：

```
uart_send(1, "12345678")
uart_send(1, "\x04\x00\0x12")

bin = "\x00\x12\x34\x56"
uart_send(1, bin)
```

5.12.4. uart_recive 接收数据

函数：

```
uart_recive(prot, maxlen, TotalTimeout, ByteTimeout)
```

说明：

- 接收串口数据，出条件满足任意 1 条
  (1)读到指定个数
  (2)没有收到字节，总超时到达 TotalTimeout
  (3)收到字节后，字符间超时到达 ByteTimeout
- port：串口号,1 或 7
- maxlen：最长长度
- TotalTimeout：总超时，ms
- ByteTimeout：字节间超时,ms。不提供该参数时，字节间超时等于总超时
- 返回: 长度, 读取到的二进制字符串

例子：

```
local str
local len

len, str = uart_recive(1, 200, 2000, 10) --读最大 200 字节, 总超时 2 秒, 字符间超时 10ms
len, str = uart_recive(1, 200, 2000) --读最大 200 字节, 总超时 2 秒，字符间超时 2 秒
```

5.12.5. uart_clear_rx 清空接收区

函数：

```
uart_clear_rx(prot)
```

说明：

- port ：串口号, 1 或 7

例子：

```
uart_clear_rx(1)
```

5.12.6. modbus_write_u16 写 16 位寄存器

函数：

```
int modbus_write_u16(port, timeout, addr485, regaddr, value0, value1, value2 ...)
```

说明：

- 发 0x06 功能码写 1 个寄存器或者发 0x10 功能码多个写寄存器，函数收到应答或等待超时才会返回
- port ： 串口号, 1 或 7
- timeout ： 超时
- addr485 ： 485 地址 0-255
- regaddr ： 寄存器地址
- value0 ： 寄存器值，可以罗列多个寄存器，函数自动识别寄存器个数
- 返回: err_code 错误代码

  0x00 OK

  0x01 不支持的功能码

  0x02 寄存器地址错误

  0x03 数据值域错误

  0x04 写入失败

  0x05 自己定义错误包应答

  0x06 超时 自定义的

例子：

```
local re
re = modbus_write_u16(1, 100, 1, 0x0000, 0x1234, 0x4567, 0x8888)

local port = 1
local timeout = 100
local addr485 = 1
local regaddr = 0
re = modbus_write_u16(port, timeout, addr485, regaddr, 0x1234, 0x4567, 0x8888)
print(re)
beep()
```

5.12.7. modbus_write_u32 写 32 位寄存器

函数：

```
int modbus_write_u32(port, timeout, addr485, regaddr, value0, value1, value2 ...)
```

说明：

- 发 0x10 功能码写 32 位整数寄存器, 2 个 16 位寄存器组合为 1 个 32 位寄存器，大端模式
- port ： 串口号, 1 或 7
- timeout ： 超时
- addr485 ： 485 地址 0-255
- regaddr ： 寄存器地址
- value0 ： 寄存器值，可以罗列多个寄存器，函数自动识别寄存器个数
- 返回: err_code 错误代码

  0x00 OK

  0x01 不支持的功能码

  0x02 寄存器地址错误

  0x03 数据值域错误

  0x04 写入失败

  0x05 自己定义错误包应答

0x06　超时 自定义的

例子：

```
local re
re = modbus_write_u32(1, 100, 1, 0x0000, 0x12345678, 0x45678888)

local port = 1
local timeout = 100
local addr485 = 1
local regaddr = 0
re = modbus_write_u16(port, timeout, addr485, regaddr, 0x12345678, 0x45678888)
print(re)
beep()
```

5.12.8. modbus_write_float 写单精度浮点寄存器

函数：

```
int modbus_write_float(port, timeout, addr485, regaddr, value0, value1, value2 ...)
```

说明：

- 发 0x10 功能码写单精度浮点寄存器，2 个 16 位寄存器组合为 1 个 32 位浮点寄存器，大端模式 float_ABCD
- port ： 串口号，1 或 7
- timeout ： 超时
- addr485 ： 485 地址 0-255
- regaddr ： 寄存器地址
- value0 ： 寄存器值，可以罗列多个寄存器，函数自动识别寄存器个数
- 返回: err_code 错误代码

  0x00　OK

  0x01　不支持的功能码

  0x02　寄存器地址错误

  0x03　数据值域错误

  0x04　写入失败

  0x05　自己定义错误包应答

  0x06　超时 自定义的

例子：

```
local re
re = modbus_write_float(1, 100, 1, 0x0000, -0.1234, 789.123)

local port = 1
local timeout = 100
local addr485 = 1
local regaddr = 0
re = modbus_write_u16(port, timeout, addr485, regaddr, -0.1234, 789.123)
print(re)
beep()
```

5.12.9. modbus_read_u16 读 16 位无符号整数寄存器

函数：

```
modbus_read_u16(port, timeout, addr485, regaddr, regnum)
```

说明：

- 发 0x03 功能码读 16 位无符号整数寄存器
- port ： 串口号, 1 或 7
- timeout ： 超时
- addr485 ： 485 地址 0-255
- regaddr ： 寄存器地址
- regnum: 寄存器个数
- 返回: err_code 错误代码

  0x00 OK

  0x01 不支持的功能码

  0x02 寄存器地址错误

  0x03 数据值域错误

  0x04 写入失败

  0x05 自己定义错误包应答

  0x06 超时 自定义的

例子：

```
local re
local data1
local data2
local port = 1
local timeout = 100
local addr485 = 1
local regaddr = 0
local value = {}

value = {modbus_read_u16(port, timeout, addr485, regaddr, 2)}
-- value[1] : 错误代码
-- value[2] : 第 1 个寄存器的值
-- value[3] : 第 3 个寄存器的值

re. data1, data2 = modbus_read_u16(port, timeout, addr485, regaddr, 2)
-- re : 错误代码
-- data1 : 第 1 个寄存器的值
-- data2 : 第 2 个寄存器的值
```

5.12.10. modbus_read_s16 读 16 位有符号整数寄存器

函数：

```
modbus_read_s16(port, timeout, addr485, regaddr, regnum)
```

说明：

- 读 16 位有符号整数寄存器，用法同 modbus_read_u16()

5.12.11. modbus_read_u32 读 32 位无符号整数寄存器

函数：

```
modbus_read_u32(port, timeout, addr485, regaddr, regnum)
```

说明：

- 读 32 位无符号整数寄存器或 2 个 16 位寄存器组合为 1 个 32 位寄存器，用法同 modbus_read_u16()

5.12.12. modbus_read_s32 读 32 位有符号整数寄存器

函数：

```
modbus_read_s32(port, timeout, addr485, regaddr, regnum)
```

说明：

- 读 32 位有符号整数寄存器或 2 个 16 位寄存器组合为 1 个 32 位寄存器，用法同 modbus_read_u16()

5.12.13. modbus_read_float 读浮点寄存器

函数：

```
modbus_read_float(port, timeout, addr485, regaddr, regnum)
```

说明：

- 读 32 位单精度浮点寄存器或 2 个 16 位寄存器组合为 1 个 32 位寄存器，用法同 modbus_read_u16()

5.12.14. modbus_write_do 写 DO 寄存器

函数：

```
modbus_write_do(port, timeout, addr485, regaddr, value0, value1 ...)
```

说明：

- 发 0x10 功能码写 1 个或多个 DO 寄存器(控制继电器)
- port ： 串口号, 1 或 7
- timeout ： 超时
- addr485 ： 485 地址 0-255
- regaddr ： 寄存器地址
- value0 ： 寄存器值，0 或 1. 可以罗列多个寄存器，函数自动识别寄存器个数
- 返回: err_code 错误代码

  0x00 OK

  0x01 不支持的功能码

  0x02 寄存器地址错误

  0x03 数据值域错误

  0x04 写入失败

0x05　　自己定义错误包应答
0x06　　超时 自定义的

例子：

```
local port = 1
local timeout = 100
local addr485 = 1
local regaddr = 0
re = modbus_write_do(port, timeout, addr485, regaddr, 1, 0, 1, 1, 0) --写 0 开始的 5 个寄存器
print(re)
```

5.12.15. modbus_read_do 读 DO 寄存器

函数：

```
modbus_read_do(port, timeout, addr485, regaddr, regnum)
```

说明：

- 发 0x01 功能码读 DO 寄存器
- port ： 串口号, 1 或 7
- timeout ： 超时
- addr485 ： 485 地址 0-255
- regaddr ： 寄存器地址
- regnum ： 寄存器个数
- 返回: err_code, value0, value1 ...
- err_code 错误代码
  0x00　　OK
  0x01　　不支持的功能码
  0x02　　寄存器地址错误
  0x03　　数据值域错误
  0x04　　写入失败
  0x05　　自己定义错误包应答
  0x06　　超时 自定义的

例子：

```
local re
local data1
local data2
local port = 1
local timeout = 100
local addr485 = 1
local regaddr = 0
local value = {}

value = {modbus_read_do(port, timeout, addr485, regaddr, 2)}
-- value[1] : 错误代码
-- value[2] : 第 1 个寄存器的值
```

```
-- value[3] : 第 3 个寄存器的值

re. data1, data2 = modbus_read_do(port, timeout, addr485, regaddr, 2)
-- re : 错误代码
-- data1 : 第 1 个寄存器的值
-- data2 : 第 2 个寄存器的值
```

5.12.16. modbus_read_di 读 DI 寄存器

函数：

```
modbus_read_do(port, timeout, addr485, regaddr, regnum)
```

说明：

- 发 0x02 功能码读 DI 寄存器
- port : 串口号, 1 或 7
- timeout : 超时
- addr485 : 485 地址 0-255
- regaddr : 寄存器地址
- regnum : 寄存器个数
- 返回: err_code, value0, value1 ...

  err_code 错误代码

  0x00 OK

  0x01 不支持的功能码

  0x02 寄存器地址错误

  0x03 数据值域错误

  0x04 写入失败

  0x05 自己定义错误包应答

  0x06 超时 自定义的

例子：

```
local re
local data1
local data2
local port = 1
local timeout = 100
local addr485 = 1
local regaddr = 0
local value = {}

value = {modbus_read_di(port, timeout, addr485, regaddr, 2)}
-- value[1] : 错误代码
-- value[2] : 第 1 个寄存器的值
-- value[3] : 第 3 个寄存器的值

re. data1, data2 = modbus_read_di(port, timeout, addr485, regaddr, 2)
-- re : 错误代码
```

```
-- data1 : 第 1 个寄存器的值
-- data2 : 第 2 个寄存器的值
```

5.12.17. 范例 1：简易回环测试

举例说明：

我们这里简单做一个回环测试，大家仅需将 TOOL 的 TTLTX 和 TTLRX 短接即可。

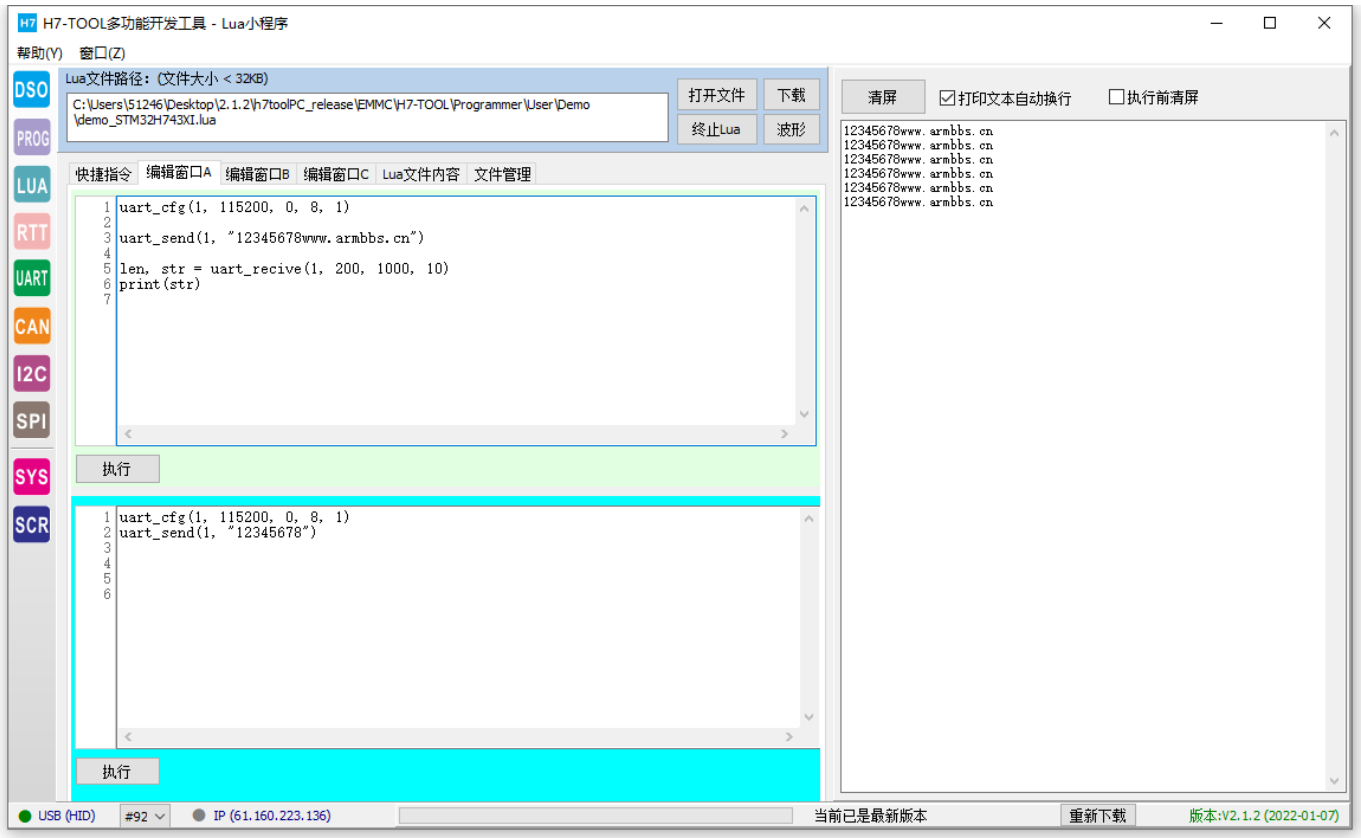
测试代码：

```
uart_cfg(1, 115200, 0, 8, 1)

uart_send(1, "12345678www.armbbs.cn")

len, str = uart_recive(1, 200, 1000, 10)
print(str)
```

测试效果：



5.12.18. 范例 2：串口 bootloader，脱机烧录

视频版：https://www.armbbs.cn/forum.php?mod=viewthread&tid=114491

文档版：https://www.armbbs.cn/forum.php?mod=viewthread&tid=113456

5.12.19. 范例 3：离线串口日志

存储到 TOOL 的 4GB 容量 eMMC 磁盘

https://www.armbbs.cn/forum.php?mod=viewthread&tid=122322

5.13. 波形打印

5.13.1. print_wave 支持 6 路波形打印

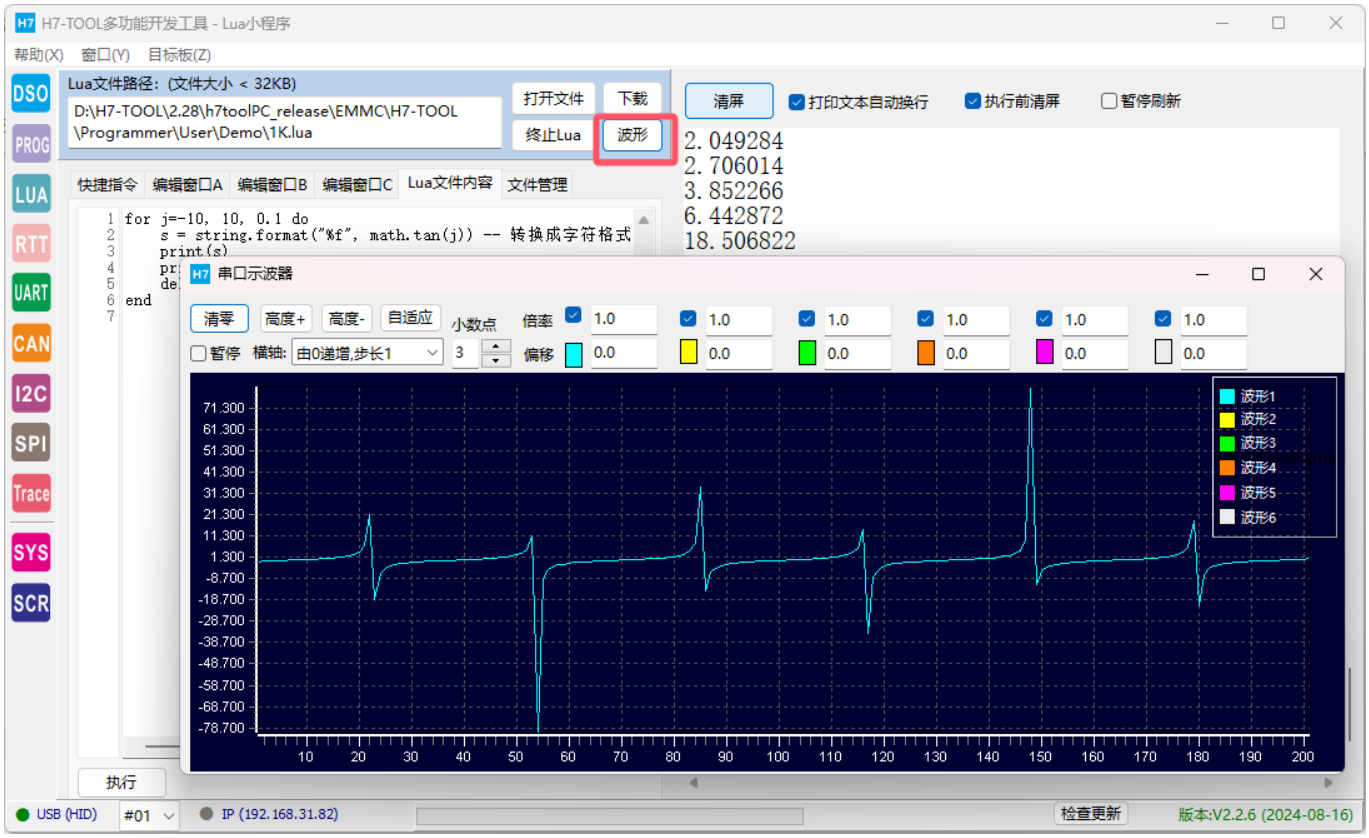
函数：

```
print_wave(s)
```

说明：

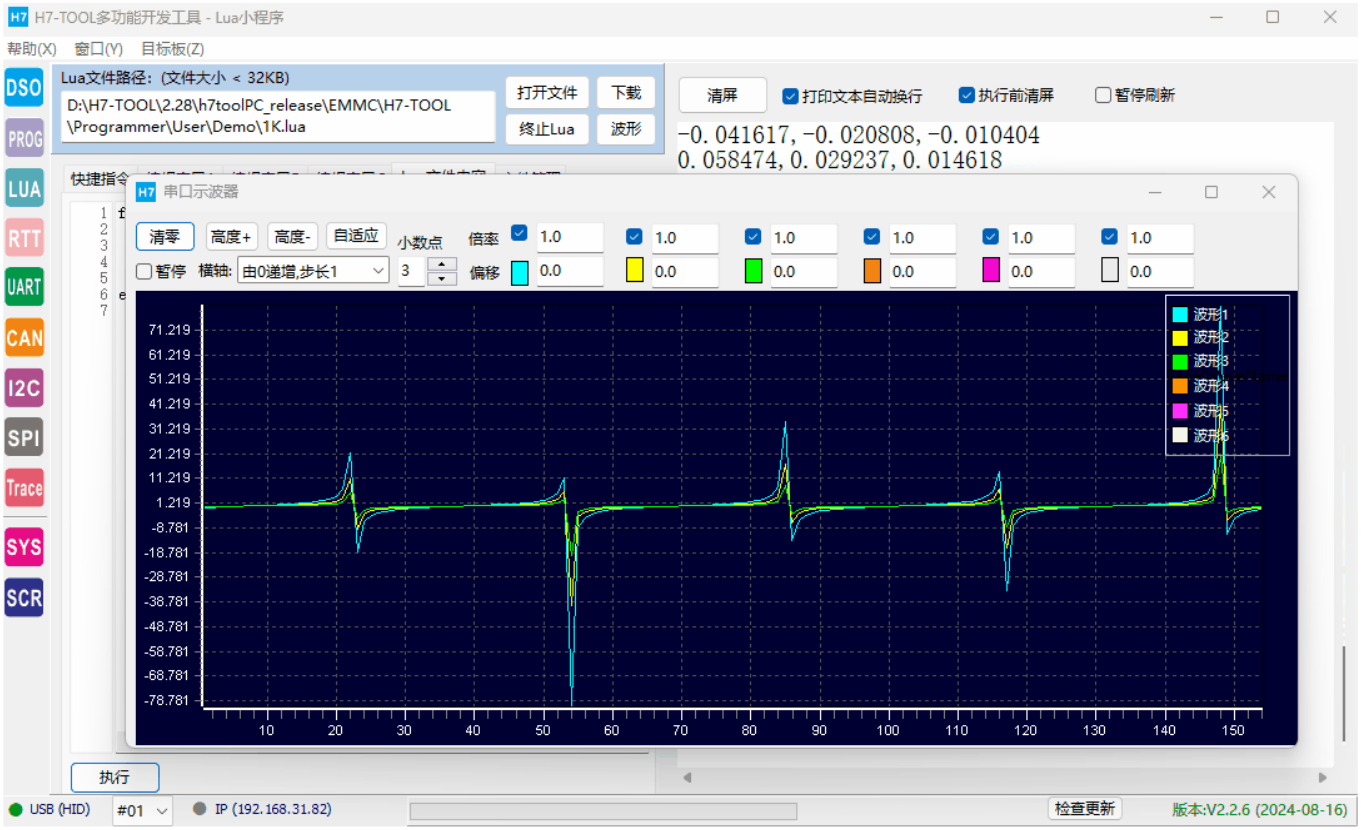
封装了一个 print_wave，方便大家将采集的数据以波形方式打印出来。使用比较简单，比如我们要测试 tan 函数：

```
for j=-10, 10, 0.1 do
        s = string.format("%f", math.tan(j)) -- 转换成字符格式
        print(s)
        print_wave(s)
        delayms(20)
end
```



又比如打印三路，就是%f, %f, %f

```
for j=-10, 10, 0.1 do
        s = string.format("%f,%f,%f", math.tan(j),math.tan(j)/2,math.tan(j)/4) -- 转换成字符格式
        print(s)
        print_wave(s)
        delayms(20)
end
```


H7-TOOL多功能开发工具 - Lua小程序
帮助(X) 窗口(Y) 目标板(Z)
DSO
PROG
LUA
RTT
UART
CAN
I2C
SPI
Trace
SYS
SCR
Lua文件路径：(文件大小 < 32KB)
D:\H7-TOOL\2.28\h7toolPC_release\EMMC\H7-TOOL\Programmer\User\Demo\1K.lua
打开文件
下载
终止Lua
波形
清屏
打印文本自动换行
执行前清屏
暂停刷新
-0. 041617, -0. 020808, -0. 010404
0. 058474, 0. 029237, 0. 014618
快捷指令
串口示波器
清零
高度+
高度-
自适应
小数点
倍率
暂停
横轴:
由0递增,步长1
偏移
波形1
波形2
波形3
波形4
波形5
波形6
71.219
61.219
51.219
41.219
31.219
21.219
11.219
1.219
-8.781
-18.781
-28.781
-38.781
-48.781
-58.781
-68.781
-78.781
10
20
30
40
50
60
70
80
90
100
110
120
130
140
150
执行
USB (HID)
#01
IP (192.168.31.82)
检查更新
版本:V2.2.6 (2024-08-16)

5.14. SWD 接口函数，读写各种数据类型，内存和寄存器

5.14.1. 注意事项

操作外设寄存器或者指定位宽的总线外设注意是 8bit，16bit 还是 32bit 操作，这样操作才能正常。支持 8bit，16bit，32bit，64bit，单精度浮点，双精度浮点，连续地址读写和字符串读取。

5.14.2. pg_init 初始化

函数：

```
pg_init()
```

说明：

- 用于初始化 SWD 接口，直接调用即可

5.14.3. pg_write8 原始 8bit 数值写

5.14.4. pg_write_u8 无符号 8bit 数值写

5.14.5. pg_write_s8 有符号 8bit 数值写

函数：

```
pg_write8 是写入原始 bit 值
pg_write_u8 表示写入无符号值
pg_write_s8 表示写入有符号值
```

说明：

- 用于目标板指定 RAM 空间地址或者寄存器地址的 8bit 数据写入

  比如写地址 0x2000 0000 地址的 8bit 数据，就是 re = pg_write(0x20000000, 0x55)，re = 1 表示 OK，0 表示失败

5.14.6. pg_read8 原始 8bit 数值读

5.14.7. pg_read_u8 无符号 8bit 数值读

5.14.8. pg_read_s8 有符号 8bit 数值读

函数：

```
pg_read8 读取是原始 bit 值
pg_read_u8 表示读取无符号值
pg_read_s8 表示读取有符号值
```

说明：

- 用于读取目标板指定 RAM 空间或者寄存器地址的 8bit 数据

  比如读取地址 0x2000 0000 地址的 8bit 数据，就是 data，re = pg_read8(0x20000000)，re = 1 表示 OK，0 表示失败， data 是返回的 8bit 数据

5.14.9. pg_write16 原始 16bit 数值写

5.14.10. pg_write_u16 无符号 16bit 数值写

5.14.11. pg_write_s16 有符号 16bit 数值写

函数：

```
pg_write16 是写入原始 bit 值
pg_write_u16 表示写入无符号值
pg_write_s16 表示写入有符号值
```

说明：

- 用于目标板指定 RAM 空间地址或者寄存器地址的 16bit 数据写入

  比如写地址 0x2000 0000 地址的 16bit 数据，就是 re = pg_write16(0x20000000, 0x5566)，re = 1 表示 OK，0 表示失败。

5.14.12. pg_read16 原始 16bit 数值读

5.14.13. pg_read_u16 无符号 16bit 数值读

5.14.14. pg_read_s16 有符号 16bit 数值读

函数：

```
pg_read16 读取是原始 bit 值
pg_read_u16 表示读取无符号值
pg_read_s16 表示读取有符号值
```

说明：

- 用于读取目标板指定 RAM 空间或者寄存器地址的 16bit 数据

  比如读取地址 0x2000 0000 地址的 16bit 数据，就是 data, re = pg_read16(0x20000000)，re = 1 表示 OK，0 表示失败， data 是返回的 16bit 数据

5.14.15. pg_write32 原始 32bit 数值写

5.14.16. pg_write_u32 无符号 32bit 数值写

5.14.17. pg_write_s32 有符号 32bit 数值写

函数：

```
pg_write32 是写入原始 bit 值
pg_write_u32 表示写入无符号值
pg_write_s32 表示写入有符号值
```

说明：

- 用于目标板指定 RAM 空间地址或者寄存器地址的 32bit 数据写入

  比如写地址 0x2000 0000 地址的 32bit 数据，就是 re = pg_write32(0x20000000, 0x55665566)，re = 1 表示 OK，0 表示失

败。

5.14.18. pg_read32 原始 32bit 数值读

5.14.19. pg_read_u32 无符号 32bit 数值读

5.14.20. pg_read_s32 有符号 32bit 数值读

函数：

```
pg_read32 读取是原始 bit 值
pg_read_u32 表示读取无符号值
pg_read_s32 表示读取有符号值
```

说明：

- 用于读取目标板指定 RAM 空间或者寄存器地址的 32bit 数据

  比如读取地址 0x2000 0000 地址的 32bit 数据，就是 data，re = pg_read32(0x20000000)，re = 1 表示 OK，0 表示失败， data 是返回的 32bit 数据

5.14.21. pg_write_float 单精度浮点数写

5.14.22. pg_read_float 单精度浮点数读

函数：

```
pg_write_float 是单精度浮点数据写入
pg_read_float 是单精度浮点数据读取
```

说明：

- 用于目标板指定 RAM 空间地址的单精度浮点数据读写

  比如读地址 0x2000 0000 地址的单精度数据，就是 data，re = pg_read_float (0x20000000)，re = 1 表示 OK，0 表示失败，data 是返回的单精度浮点数

  比如地址 0x2000 0000 写单精度数据，就是 re = pg_write_float (0x20000000)，re = 1 表示 OK，0 表示失败

5.14.23. pg_write_double 双精度浮点写

5.14.24. pg_read_double 双精度浮点读

函数：

```
pg_write_double 是双精度浮点数据写入
pg_read_double 是双精度浮点数据读取
```

说明：

- 用于目标板指定 RAM 空间地址的双精度浮点数据读写

  比如读地址 0x2000 0000 地址的双精度数据，就是 data，re = pg_read_double(0x20000000)，re = 1 表示 OK，0 表示失败，data 是返回的双精度浮点数

比如地址 0x2000 0000 写双精度数据，就是 re = pg_write_double(0x20000000)，re = 1 表示 OK，0 表示失败

5.14.25. pg_write_int64 有符号 64bit 写

5.14.26. pg_read_int64 有符号 64bit 读

函数：

```
pg_write_double 是双精度浮点数据写入
pg_read_double 是双精度浮点数据读取
```

说明：

- 用于目标板指定 RAM 空间地址的双精度浮点数据读写

  比如读地址 0x2000 0000 地址的双精度数据，就是 data，re = pg_read_double(0x20000000)，re = 1 表示 OK，0 表示失败，data 是返回的双精度浮点数

  比如地址 0x2000 0000 写双精度数据，就是 re = pg_write_double(0x20000000)，re = 1 表示 OK，0 表示失败

5.14.27. pg_write_mem 内存批量写

函数：

```
pg_write_mem
```

说明：

- 用于目标板指定 RAM 空间地址数据的写入，最大写 4096 字节
  比如给地址 0x20000000 写 5 个数据，就是
  data = "\x00\x00\x11\x22\x33"
  re = pg_write_mem(0x24000000, data)
  re = 1 表示 OK，0 表示失败。

5.14.28. pg_read_mem 内存批量读

函数：

```
pg_read_mem
```

说明：

- 用于目标板指定 RAM 空间地址数据的读取，最大读取 4096 字节
  比如读地址 0x20000000 地址的 10 个数据，就是
  re, data = pg_read_mem(0x20000000, 10)
  re = 1 表示 OK，0 表示失败。 data 是二进制字符串数组，只是以字符串形式表示，字符的原始值还是本身，读取出来就是类似这样 data = "\x00\x00\x11\x22\x33"

5.14.29. pg_write_mem 内存批量写

函数：

```
pg_write_mem
```

说明：

- 用于目标板指定 RAM 空间地址数据的写入，最大写 4096 字节
  比如给地址 0x20000000 写 5 个数据，就是
  data = "\x00\x00\x11\x22\x33"
  re = pg_write_mem(0x24000000, data)
  re = 1 表示 OK, 0 表示失败。

5.14.30. 范例 1：8，16，32，64bit，单精度和双精度浮点访问

测试程序：

```
-- 根据需要判断 re 返回值，不想判断，可以不加返回值
local re
local data
local value
local value1
local value2
local value3

function printf(...)
    print(string.format(...))
end

pg_init()

-- 8bit 测试
pg_write_u8(0x24000000, 4)
value = pg_read_u8(0x24000000)
printf("pg_read_u8 = %d", value)

-- 16bit 测试
pg_write_s16(0x24000000, -466)
value = pg_read_s16(0x24000000)
printf("pg_read_s16 = %d", value)

-- 32bit 测试
pg_write_s32(0x24000000, -466777)
value = pg_read_s32(0x24000000)
printf("pg_read_s32 = %d", value)

-- 单精度浮点
pg_write_float(0x24000000, 0.12)
value2 = pg_read_float(0x24000000)
printf("pg_read_float = %f", value2)

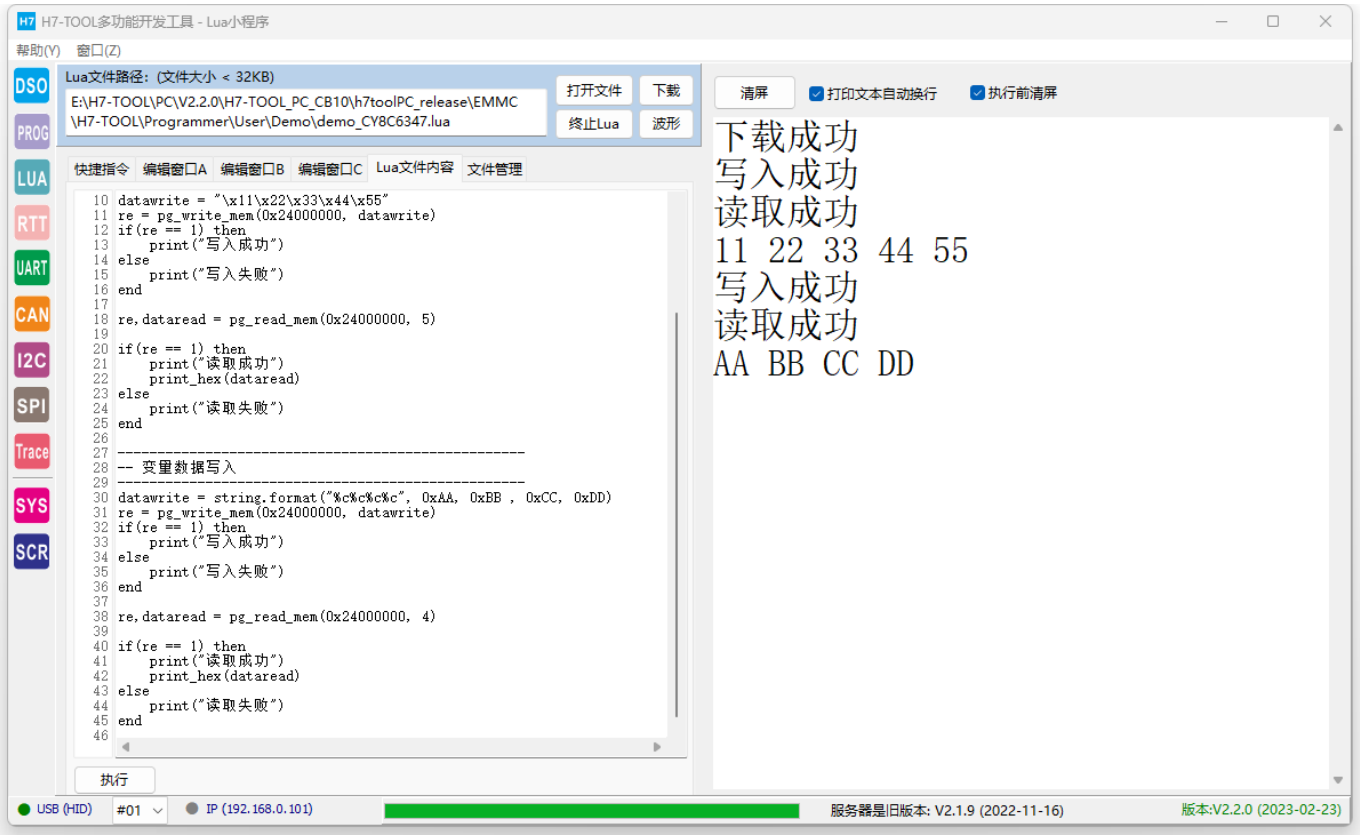
-- 64bit 测试
re = pg_write_int64(0x24000000, 0x1122334455667788)
if(re == 1) then
    print("64bit 数写入成功")
```

```
else
    print("64bit 数写入失败")
end

value1, re= pg_read_int64(0x24000000)
if(re == 1) then
    printf("pg_read_int64 = %16x", value1)
else
    print("64bit 数读取失败")
end

-- 双精度浮点
re = pg_write_double(0x24000000, 0.123)
if(re == 1) then
    print("双精度浮点数写入成功")
else
    print("双精度浮点数写入失败")
end

value3, re= pg_read_double(0x24000000)
if(re == 1) then
    printf("pg_read_double = %.15f", value3)
else
    print("双精度浮点数读取失败")
end
```



5.14.31. 范例 2：批量内存读写测试

测试程序：

```
local dataread
local datawrite
local re

pg_init()  -- 初始化 SWD 接口

-------------------------------------------------
-- 常数方式直接写入
-------------------------------------------------
datawrite = "\x11\x22\x33\x44\x55"
re = pg_write_mem(0x24000000, datawrite)
if(re == 1) then
    print("写入成功")
else
    print("写入失败")
end

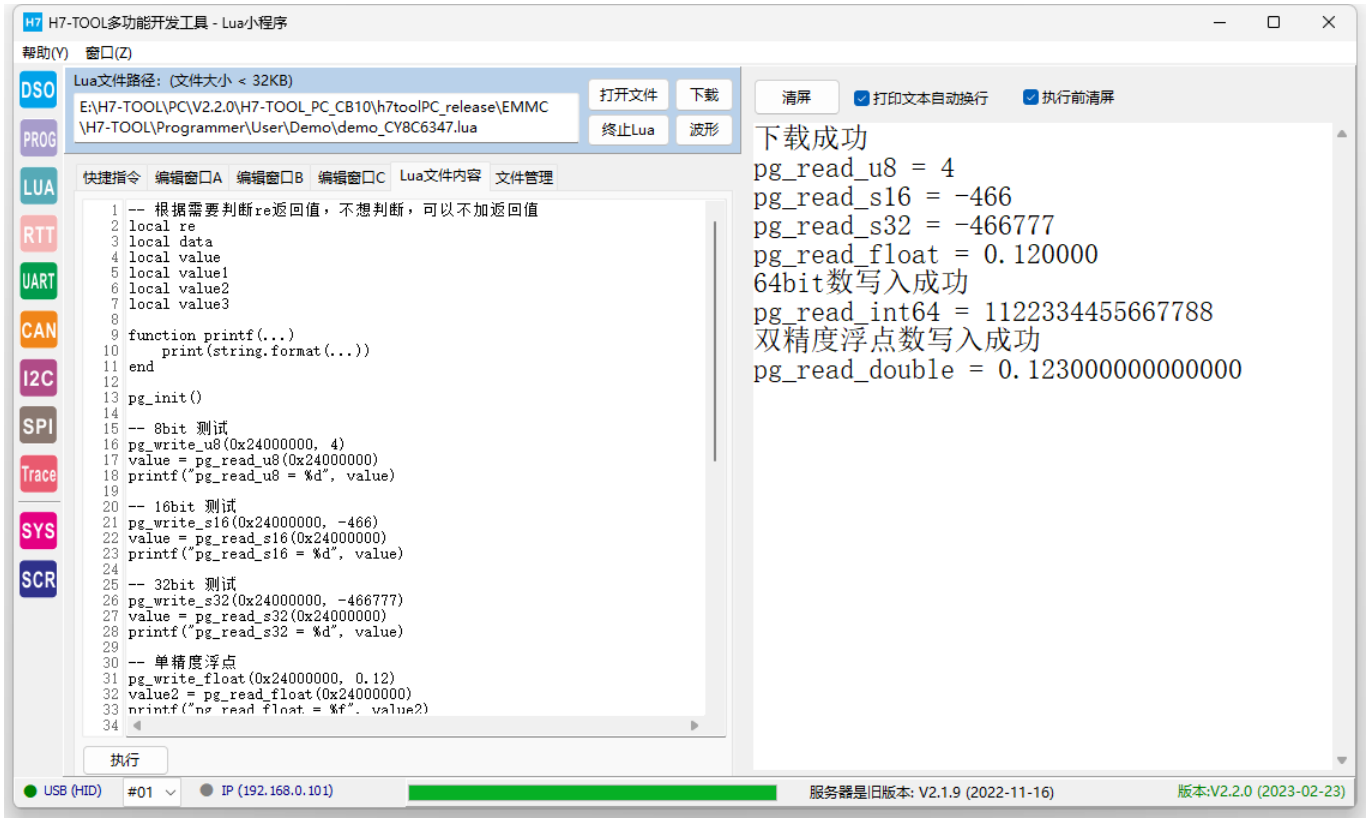
re,dataread = pg_read_mem(0x24000000, 5)

if(re == 1) then
    print("读取成功")
    print_hex(dataread)
else
    print("读取失败")
end

-------------------------------------------------
-- 变量数据写入
-------------------------------------------------
datawrite = string.format("%c%c%c%c", 0xAA, 0xBB , 0xCC, 0xDD)
re = pg_write_mem(0x24000000, datawrite)
if(re == 1) then
    print("写入成功")
else
    print("写入失败")
end

re,dataread = pg_read_mem(0x24000000, 4)

if(re == 1) then
    print("读取成功")
    print_hex(dataread)
else
    print("读取失败")
end
```



5.14.32. 范例 3：检索 stack 和 heap 使用情况

https://www.armbbs.cn/forum.php?mod=viewthread&tid=119024

5.14.33. 范例 4：检索 Flash 或者 RAM 空间数值方法

https://www.armbbs.cn/forum.php?mod=viewthread&tid=116766

5.14.34. 范例 5：访问 AP 和 DP 调试寄存器方法

https://www.armbbs.cn/forum.php?mod=viewthread&tid=120658

5.14.35. 范例 6：获取芯片 NVIC 优先级分组和芯片内核版本

https://www.armbbs.cn/forum.php?mod=viewthread&tid=120658

5.14.36. 范例 7：检索芯片硬件版本

https://www.armbbs.cn/forum.php?mod=viewthread&tid=119195

5.14.37. 范例 8：控制目标芯片 GPIO

https://www.armbbs.cn/forum.php?mod=viewthread&tid=113942

5.14.38. 范例 9：综合应用之 RTOS Trace

这个功能的实现基于 SWD 接口函数
RTOS Trace 功能介绍： https://www.armbbs.cn/forum.php?mod=viewthread&tid=114824
源文件实现位于上位机软件路径：h7toolPC_release\Trace

5.14.39. 范例 10：综合应用之目标板截图

这个功能的实现基于 SWD 接口函数
截图功能介绍： https://www.armbbs.cn/forum.php?mod=viewthread&tid=118116
源文件实现位于上位机软件路径：h7toolPC_release\Trace\Screeshot

5.14.40. 范例 11：综合应用之硬件异常黑盒子

这个功能的实现基于 SWD 接口函数
功能介绍： https://www.armbbs.cn/forum.php?mod=viewthread&tid=114792
源文件实现位于上位机软件路径：h7toolPC_release\Trace\Hardfault

5.15. ADC 模拟量采集

5.15.1. 简介

电压，电流，NTC 热敏电阻以及 4-20mA 输入，可以在上位机端设置，也可以显示屏端设置
详细使用说明可以看在线或者离线操作说明手册：
https://www.armbbs.cn/forum.php?mod=viewthread&tid=95373
建议优先熟悉下，特别是这几个功能对应使用的引脚。

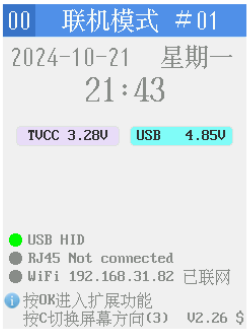












5.15.2. 启动模拟量采集

启动模拟量采集仅需用到两个大类配置，一个负载电流测量，还有一个低速多通道。所以启动模拟信号采集封装了两种配置
（1）负载电流测量，配置代码固定如下：

```
function start_dso(void)
	write_reg16(0x01FF, 1) -- 测量模式 0:示波器 1:负载电流 2:多路低速扫描
	write_reg16(0x0200, 1) -- CH1 耦合,0:AC 1:DC
	write_reg16(0x0201, 1) -- CH2 耦合,0:AC 1:DC

	--量程取值 0:±13.8V 1:±6.4V 2:±3.2V 3:±1.6V 4:±800mV 5:±400mV 6:±200mV 7:±100mV
	write_reg16(0x0202, 0) -- CH1 量程
	write_reg16(0x0203, 0) -- CH2 量程
	write_reg16(0x0204, 0) -- CH1 通道直流偏值，未用
	write_reg16(0x0205, 0) -- CH2 通道直流偏值，未用
	write_reg16(0x0206, 12) --采样频率 0:100 1:200 2:500 3:1K 4:2K 5:5K 6:10K 7:20K
                                --8:50K 9:100K 10:200K 11:500K 12:1M 13:2M 14:5M
	write_reg16(0x0207, 0) --采样深度 0:1K 1:2K 3:4K 4:8K 5:16K 6:32K
	write_reg16(0x0208, 32768) --触发电平 ADC 0-65535
	write_reg16(0x0209, 50) --触发位置百分比 0-100
	write_reg16(0x020A, 0) --触发模式 0:自动 1:普通 2:单次
	write_reg16(0x020B, 0) --触发通道 0:CH1 1:CH2
```

```
      write_reg16(0x020C, 0) --触发边沿 0:下降沿 1:上升沿
      write_reg16(0x020D, 0x03) --通道使能控制 bit0 = CH1   bit1 = CH2
      write_reg16(0x020E, 1) --采集控制 0:停止 1:启动
end
```

（2）低速多通道测量，配置代码固定如下：

```
--启动模拟量电路
function start_dso(void)
      write_reg16(0x01FF, 2) -- 测量模式 0:示波器 1:负载电流 2:多路低速扫描
      write_reg16(0x0200, 1) -- CH1 耦合,0:AC 1:DC
      write_reg16(0x0201, 1) -- CH2 耦合,0:AC 1:DC

      --量程取值 0:±13.8V 1:±6.4V 2:±3.2V 3:±1.6V 4:±800mV 5:±400mV 6:±200mV 7:±100mV
      write_reg16(0x0202, 0) -- CH1 量程
      write_reg16(0x0203, 0) -- CH2 量程
      write_reg16(0x0204, 0) -- CH1 通道直流偏值，未用
      write_reg16(0x0205, 0) -- CH2 通道直流偏值，未用
      write_reg16(0x0206, 12) --采样频率 0:100 1:200 2:500 3:1K 4:2K 5:5K 6:10K 7:20K
                              --8:50K 9:100K 10:200K 11:500K 12:1M 13:2M 14:5M
      write_reg16(0x0207, 0) --采样深度 0:1K 1:2K 3:4K 4:8K 5:16K 6:32K
      write_reg16(0x0208, 32768) --触发电平 ADC 0-65535
      write_reg16(0x0209, 50) --触发位置百分比 0-100
      write_reg16(0x020A, 0) --触发模式 0:自动 1:普通 2:单次
      write_reg16(0x020B, 0) --触发通道 0:CH1 1:CH2
      write_reg16(0x020C, 0) --触发边沿 0:下降沿 1:上升沿
      write_reg16(0x020D, 0x03) --通道使能控制 bit0 = CH1   bit1 = CH2
      write_reg16(0x020E, 1) --采集控制 0:停止 1:启动
end
```

5.15.3. read_analog 注意事项

测量函数比较简单，周期调用即可，建议 100ms 以上读取一次，因为所有数据 100ms 更新一轮

5.15.4. read_analog 电压读取

```
read_analog(0) -- 0 - CH1 电压
read_analog(1) -- 1 - CH2 电压
```

举例：每 500ms 读取一次 CH1 和 CH2 通道电压

实现代码如下：

```
--启动模拟量电路
function start_dso(void)
        write_reg16(0x01FF, 2) -- 测量模式 0:示波器 1:负载电流 2:多路低速扫描
        write_reg16(0x0200, 1) -- CH1 耦合,0:AC 1:DC
        write_reg16(0x0201, 1) -- CH2 耦合,0:AC 1:DC

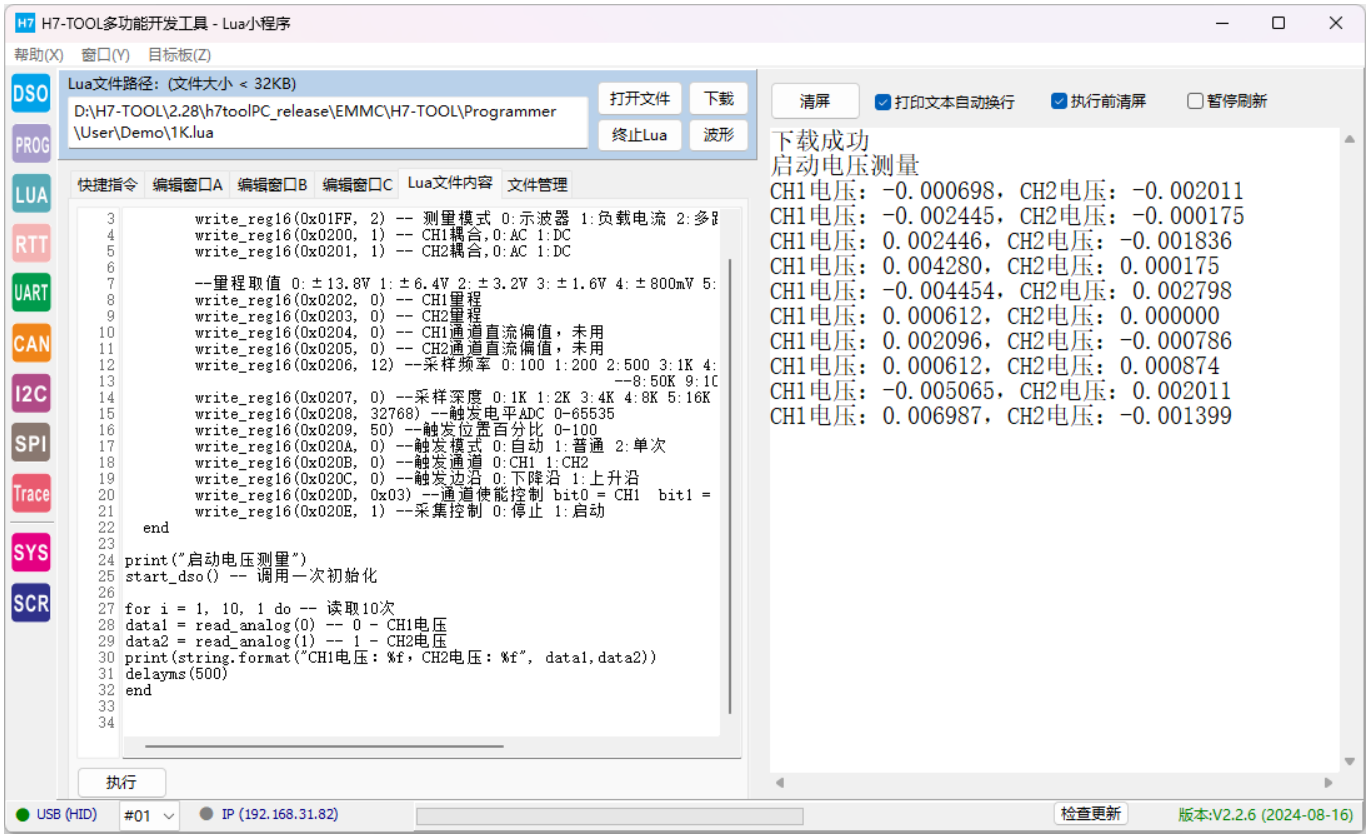
        --量程取值 0:±13.8V 1:±6.4V 2:±3.2V 3:±1.6V 4:±800mV 5:±400mV 6:±200mV 7:±100mV
```

```
        write_reg16(0x0202, 0) -- CH1 量程
        write_reg16(0x0203, 0) -- CH2 量程
        write_reg16(0x0204, 0) -- CH1 通道直流偏值，未用
        write_reg16(0x0205, 0) -- CH2 通道直流偏值，未用
        write_reg16(0x0206, 12) --采样频率 0:100 1:200 2:500 3:1K 4:2K 5:5K 6:10K 7:20K
                                                    --8:50K 9:100K 10:200K 11:500K 12:1M 13:2M 14:5M
        write_reg16(0x0207, 0) --采样深度 0:1K 1:2K 3:4K 4:8K 5:16K 6:32K
        write_reg16(0x0208, 32768) --触发电平 ADC 0-65535
        write_reg16(0x0209, 50) --触发位置百分比 0-100
        write_reg16(0x020A, 0) --触发模式 0:自动 1:普通 2:单次
        write_reg16(0x020B, 0) --触发通道 0:CH1 1:CH2
        write_reg16(0x020C, 0) --触发边沿 0:下降沿 1:上升沿
        write_reg16(0x020D, 0x03) --通道使能控制 bit0 = CH1  bit1 = CH2
        write_reg16(0x020E, 1) --采集控制 0:停止 1:启动
  end

print("启动电压测量")
start_dso() -- 调用一次初始化

for i = 1, 10, 1 do -- 读取 10 次
data1 = read_analog(0) -- 0 - CH1 电压
data2 = read_analog(1) -- 1 - CH2 电压
print(string.format("CH1 电压：%f，CH2 电压：%f", data1,data2))
delayms(500)
end
```

实际效果：



5.15.5. read_analog 高侧负载测量

read_analog(2) --2 - 高侧负载电压
read_analog(3) --3 - 高端负载电流

举例：每 500ms 读取一次

实现代码如下（特别注意，这个启动代码采用的负载电流启动模式）：

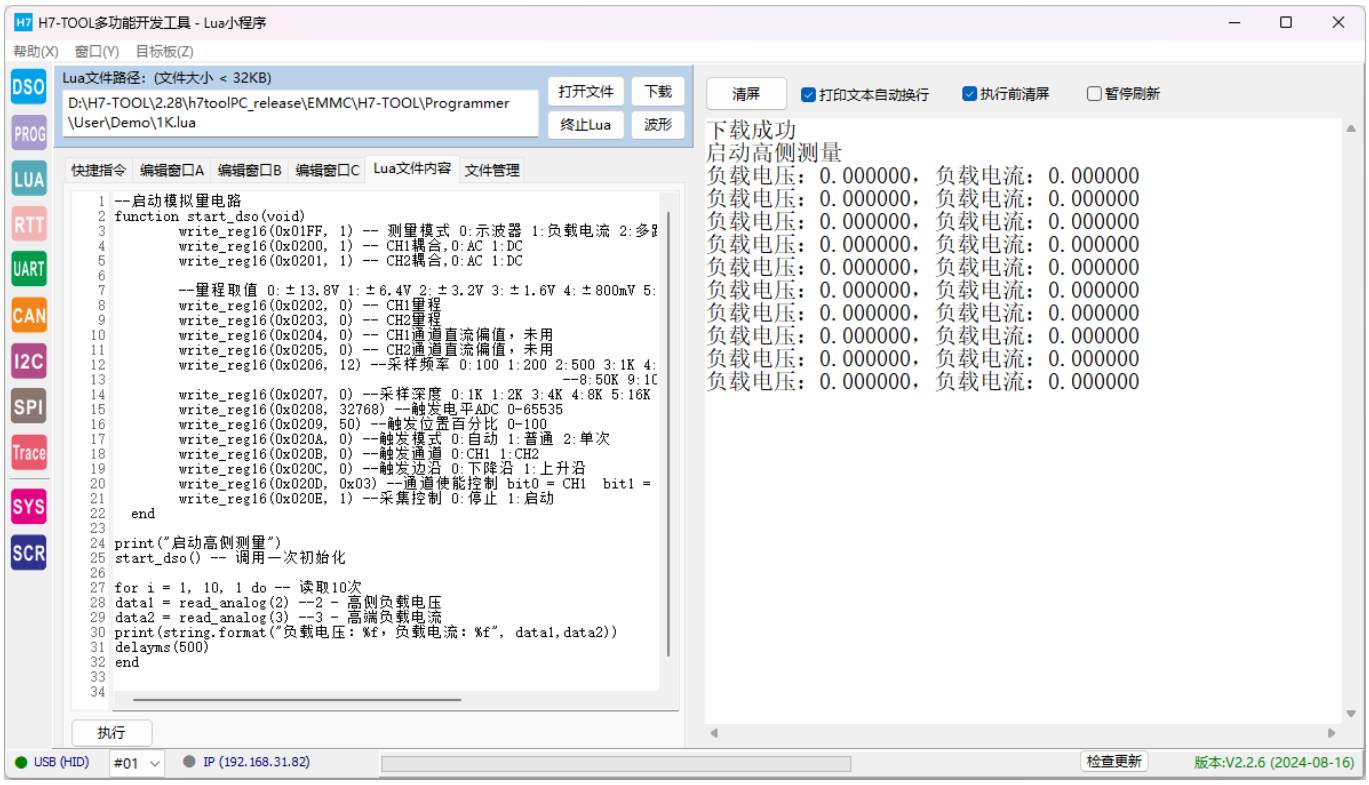
```
--启动模拟量电路
function start_dso(void)
        write_reg16(0x01FF, 1) -- 测量模式 0:示波器 1:负载电流 2:多路低速扫描
        write_reg16(0x0200, 1) -- CH1 耦合,0:AC 1:DC
        write_reg16(0x0201, 1) -- CH2 耦合,0:AC 1:DC

        --量程取值 0:±13.8V 1:±6.4V 2:±3.2V 3:±1.6V 4:±800mV 5:±400mV 6:±200mV 7:±100mV
        write_reg16(0x0202, 0) -- CH1 量程
        write_reg16(0x0203, 0) -- CH2 量程
        write_reg16(0x0204, 0) -- CH1 通道直流偏值，未用
        write_reg16(0x0205, 0) -- CH2 通道直流偏值，未用
        write_reg16(0x0206, 12) --采样频率 0:100 1:200 2:500 3:1K 4:2K 5:5K 6:10K 7:20K
                                                        --8:50K 9:100K 10:200K 11:500K 12:1M 13:2M 14:5M
        write_reg16(0x0207, 0) --采样深度 0:1K 1:2K 3:4K 4:8K 5:16K 6:32K
        write_reg16(0x0208, 32768) --触发电平 ADC 0-65535
        write_reg16(0x0209, 50) --触发位置百分比 0-100
        write_reg16(0x020A, 0) --触发模式 0:自动 1:普通 2:单次
        write_reg16(0x020B, 0) --触发通道 0:CH1 1:CH2
        write_reg16(0x020C, 0) --触发边沿 0:下降沿 1:上升沿
        write_reg16(0x020D, 0x03) --通道使能控制 bit0 = CH1  bit1 = CH2
        write_reg16(0x020E, 1) --采集控制 0:停止 1:启动
  end

print("启动高侧测量")
start_dso() -- 调用一次初始化

for i = 1, 10, 1 do -- 读取 10 次
data1 = read_analog(2) --2 - 高侧负载电压
data2 = read_analog(3) --3 - 高端负载电流
print(string.format("负载电压：%f, 负载电流：%f", data1,data2))
delayms(500)
end
```

实际效果：



5.15.6. read_analog TVCC 测量

read_analog(4) --4 - TVCC 电压

read_analog(5) --5 - TVCC 电流

举例：每 500ms 读取一次

实现代码如下：

```
--启动模拟量电路
function start_dso(void)
        write_reg16(0x01FF, 2) -- 测量模式 0:示波器 1:负载电流 2:多路低速扫描
        write_reg16(0x0200, 1) -- CH1 耦合,0:AC 1:DC
        write_reg16(0x0201, 1) -- CH2 耦合,0:AC 1:DC

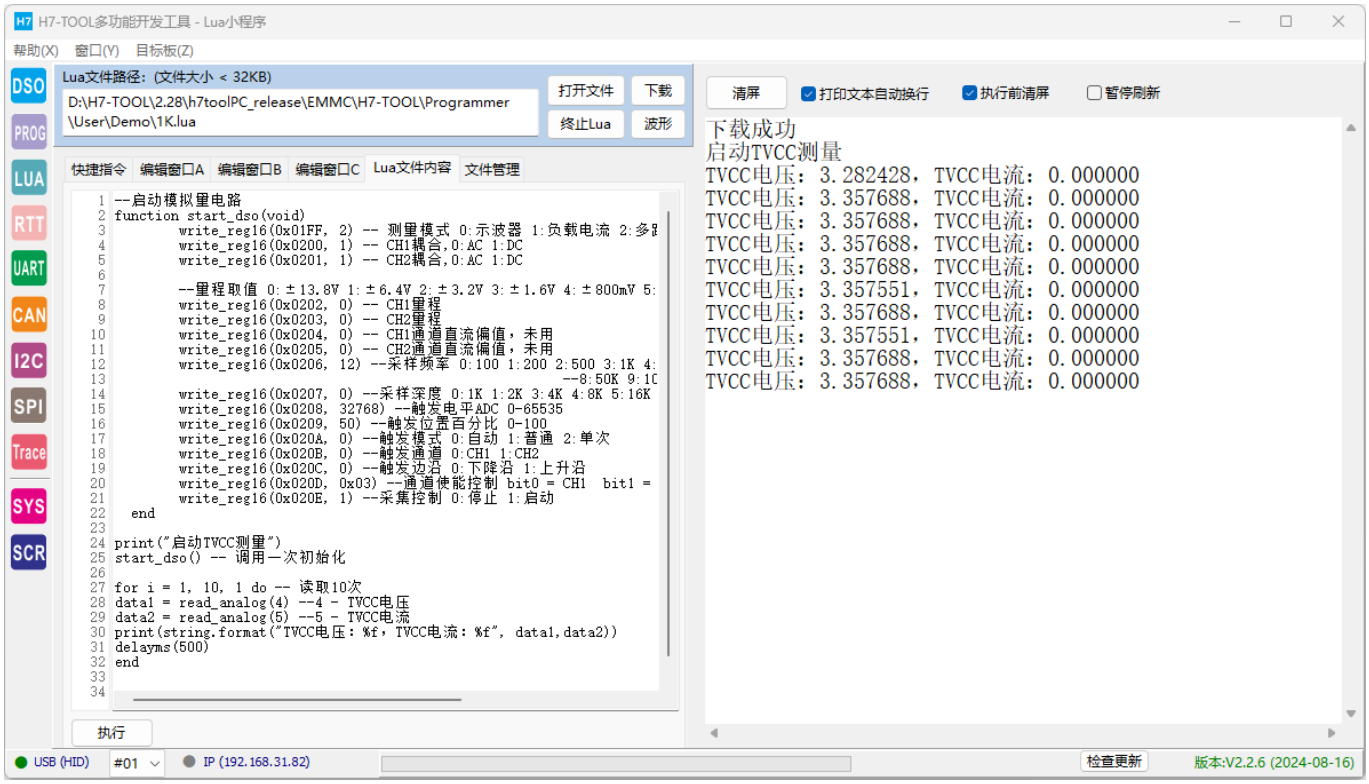
        --量程取值 0:±13.8V 1:±6.4V 2:±3.2V 3:±1.6V 4:±800mV 5:±400mV 6:±200mV 7:±100mV
        write_reg16(0x0202, 0) -- CH1 量程
        write_reg16(0x0203, 0) -- CH2 量程
        write_reg16(0x0204, 0) -- CH1 通道直流偏值，未用
        write_reg16(0x0205, 0) -- CH2 通道直流偏值，未用
        write_reg16(0x0206, 12) --采样频率 0:100 1:200 2:500 3:1K 4:2K 5:5K 6:10K 7:20K
                                                --8:50K 9:100K 10:200K 11:500K 12:1M 13:2M 14:5M
        write_reg16(0x0207, 0) --采样深度 0:1K 1:2K 3:4K 4:8K 5:16K 6:32K
        write_reg16(0x0208, 32768) --触发电平 ADC 0-65535
        write_reg16(0x0209, 50) --触发位置百分比 0-100
        write_reg16(0x020A, 0) --触发模式 0:自动 1:普通 2:单次
        write_reg16(0x020B, 0) --触发通道 0:CH1 1:CH2
        write_reg16(0x020C, 0) --触发边沿 0:下降沿 1:上升沿
        write_reg16(0x020D, 0x03) --通道使能控制 bit0 = CH1  bit1 = CH2
```

```
        write_reg16(0x020E, 1) --采集控制 0:停止 1:启动
  end

print("启动 TVCC 测量")
start_dso() -- 调用一次初始化

for i = 1, 10, 1 do -- 读取 10 次
data1 = read_analog(4) --4 - TVCC 电压
data2 = read_analog(5) --5 - TVCC 电流
print(string.format("TVCC 电压：%f，TVCC 电流：%f", data1,data2))
delayms(500)
end
```

实际效果：



5.15.7. read_analog NTC 热敏电阻测量

read_analog(6) --6 - NTC 热敏电阻阻值

举例：每 500ms 读取一次

```
--启动模拟量电路
function start_dso(void)
        write_reg16(0x01FF, 2) -- 测量模式 0:示波器 1:负载电流 2:多路低速扫描
        write_reg16(0x0200, 1) -- CH1 耦合,0:AC 1:DC
        write_reg16(0x0201, 1) -- CH2 耦合,0:AC 1:DC

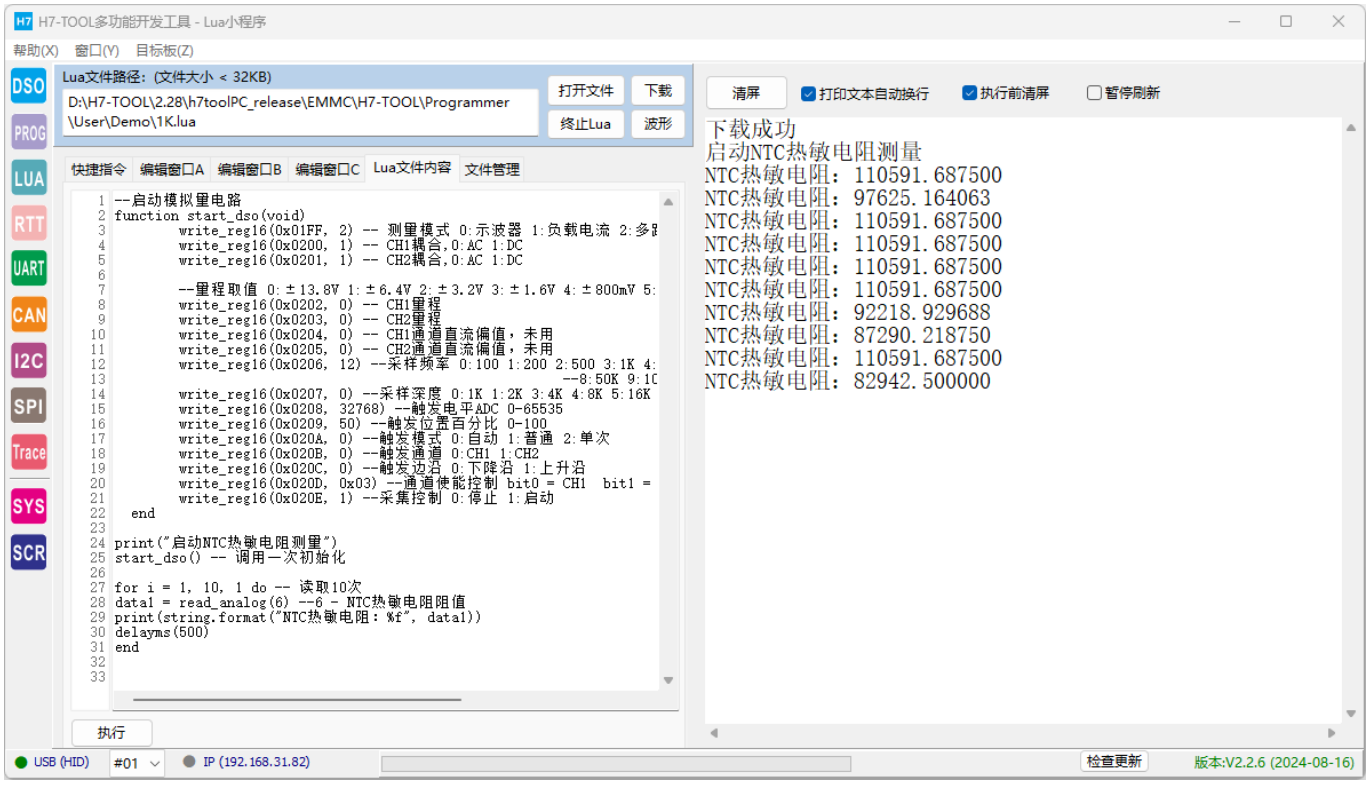
        --量程取值 0:±13.8V 1:±6.4V 2:±3.2V 3:±1.6V 4:±800mV 5:±400mV 6:±200mV 7:±100mV
        write_reg16(0x0202, 0) -- CH1 量程
        write_reg16(0x0203, 0) -- CH2 量程
```

```
        write_reg16(0x0204, 0) -- CH1 通道直流偏值，未用
        write_reg16(0x0205, 0) -- CH2 通道直流偏值，未用
        write_reg16(0x0206, 12) --采样频率 0:100 1:200 2:500 3:1K 4:2K 5:5K 6:10K 7:20K
                                                        --8:50K 9:100K 10:200K 11:500K 12:1M 13:2M 14:5M
        write_reg16(0x0207, 0) --采样深度 0:1K 1:2K 3:4K 4:8K 5:16K 6:32K
        write_reg16(0x0208, 32768) --触发电平 ADC 0-65535
        write_reg16(0x0209, 50) --触发位置百分比 0-100
        write_reg16(0x020A, 0) --触发模式 0:自动 1:普通 2:单次
        write_reg16(0x020B, 0) --触发通道 0:CH1 1:CH2
        write_reg16(0x020C, 0) --触发边沿 0:下降沿 1:上升沿
        write_reg16(0x020D, 0x03) --通道使能控制 bit0 = CH1  bit1 = CH2
        write_reg16(0x020E, 1) --采集控制 0:停止 1:启动
  end

print("启动 NTC 热敏电阻测量")
start_dso() -- 调用一次初始化

for i = 1, 10, 1 do -- 读取 10 次
data1 = read_analog(6) --6 - NTC 热敏电阻阻值
print(string.format("NTC 热敏电阻：%f", data1))
delayms(500)
end
```

实际效果：



5.15.8. read_analog 外置电源和 USB 供电电压测量

read_adc(7) --7 - 外部供电电压

read_analog(8) --8 - USB 供电电压

举例：每 500ms 读取一次

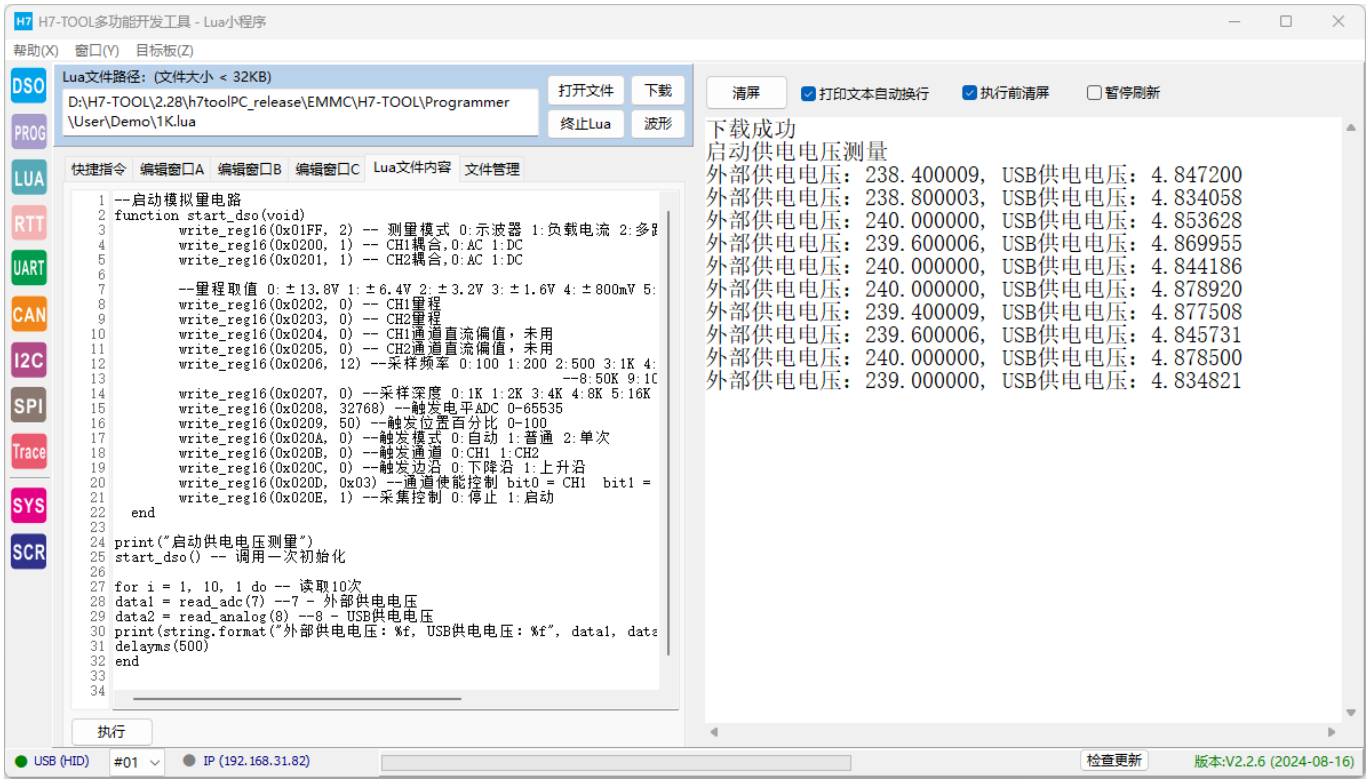
```
--启动模拟量电路
function start_dso(void)
        write_reg16(0x01FF, 2) -- 测量模式 0:示波器 1:负载电流 2:多路低速扫描
        write_reg16(0x0200, 1) -- CH1 耦合,0:AC 1:DC
        write_reg16(0x0201, 1) -- CH2 耦合,0:AC 1:DC

        --量程取值 0:±13.8V 1:±6.4V 2:±3.2V 3:±1.6V 4:±800mV 5:±400mV 6:±200mV 7:±100mV
        write_reg16(0x0202, 0) -- CH1 量程
        write_reg16(0x0203, 0) -- CH2 量程
        write_reg16(0x0204, 0) -- CH1 通道直流偏值，未用
        write_reg16(0x0205, 0) -- CH2 通道直流偏值，未用
        write_reg16(0x0206, 12) --采样频率 0:100 1:200 2:500 3:1K 4:2K 5:5K 6:10K 7:20K
                                                        --8:50K 9:100K 10:200K 11:500K 12:1M 13:2M 14:5M
        write_reg16(0x0207, 0) --采样深度 0:1K 1:2K 3:4K 4:8K 5:16K 6:32K
        write_reg16(0x0208, 32768) --触发电平 ADC 0-65535
        write_reg16(0x0209, 50) --触发位置百分比 0-100
        write_reg16(0x020A, 0) --触发模式 0:自动 1:普通 2:单次
        write_reg16(0x020B, 0) --触发通道 0:CH1 1:CH2
        write_reg16(0x020C, 0) --触发边沿 0:下降沿 1:上升沿
        write_reg16(0x020D, 0x03) --通道使能控制 bit0 = CH1  bit1 = CH2
        write_reg16(0x020E, 1) --采集控制 0:停止 1:启动
  end

print("启动供电电压测量")
start_dso() -- 调用一次初始化

for i = 1, 10, 1 do -- 读取 10 次
data1 = read_adc(7) --7 - 外部供电电压
data2 = read_analog(8) --8 - USB 供电电压
print(string.format("外部供电电压：%f, USB 供电电压：%f", data1, data2))
delayms(500)
end
```

实际效果：



5.15.9. read_analog 4-20mA 输入测量

read_analog(9) -- 4-20mA 测量

举例：每 500ms 读取一次

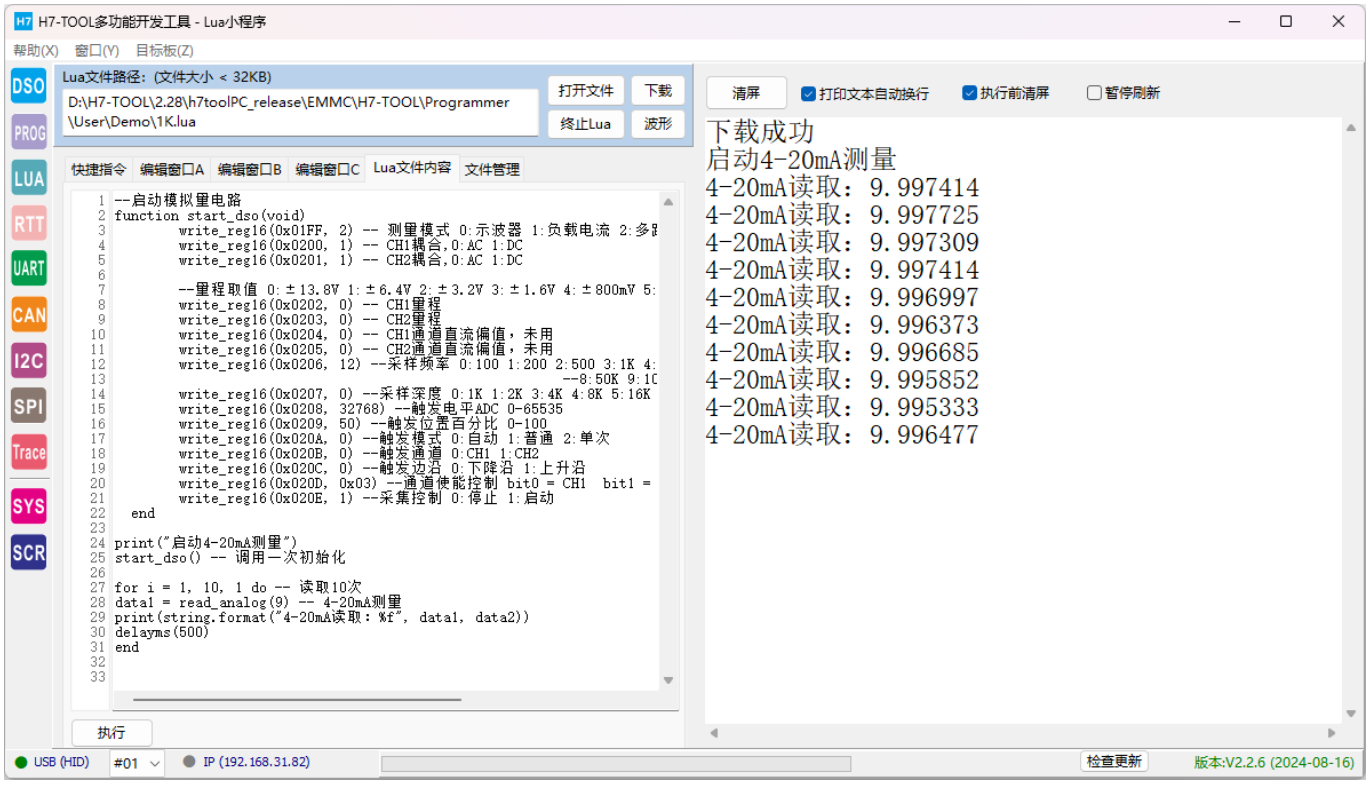
```
--启动模拟量电路
function start_dso(void)
        write_reg16(0x01FF, 2) -- 测量模式 0:示波器 1:负载电流 2:多路低速扫描
        write_reg16(0x0200, 1) -- CH1 耦合,0:AC 1:DC
        write_reg16(0x0201, 1) -- CH2 耦合,0:AC 1:DC

        --量程取值 0:±13.8V 1:±6.4V 2:±3.2V 3:±1.6V 4:±800mV 5:±400mV 6:±200mV 7:±100mV
        write_reg16(0x0202, 0) -- CH1 量程
        write_reg16(0x0203, 0) -- CH2 量程
        write_reg16(0x0204, 0) -- CH1 通道直流偏值，未用
        write_reg16(0x0205, 0) -- CH2 通道直流偏值，未用
        write_reg16(0x0206, 12) --采样频率 0:100 1:200 2:500 3:1K 4:2K 5:5K 6:10K 7:20K
                                                --8:50K 9:100K 10:200K 11:500K 12:1M 13:2M 14:5M
        write_reg16(0x0207, 0) --采样深度 0:1K 1:2K 3:4K 4:8K 5:16K 6:32K
        write_reg16(0x0208, 32768) --触发电平 ADC 0-65535
        write_reg16(0x0209, 50) --触发位置百分比 0-100
        write_reg16(0x020A, 0) --触发模式 0:自动 1:普通 2:单次
        write_reg16(0x020B, 0) --触发通道 0:CH1 1:CH2
        write_reg16(0x020C, 0) --触发边沿 0:下降沿 1:上升沿
        write_reg16(0x020D, 0x03) --通道使能控制 bit0 = CH1  bit1 = CH2
        write_reg16(0x020E, 1) --采集控制 0:停止 1:启动
  end
```

```
print("启动 4-20mA 测量")
start_dso() -- 调用一次初始化

for i = 1, 10, 1 do -- 读取 10 次
data1 = read_analog(9) -- 4-20mA 测量
print(string.format("4-20mA 读取：%f", data1, data2))
delayms(500)
end
```

实际效果：测量的 10mA，精度还是非常不错的



5.15.10. read_adc 直接读取 ADC 的值

这个用法和前面 read_analog 读取模拟值是完全一样的。只是这里获取的是 ADC 支持。

read_adc(0) --0 - CH1 电压

read_adc(1) --1 - CH2 电压

read_adc(2) --2 - 高侧负载电压

read_adc(3) --3 - 高端负载电流

read_adc(4) --4 - TVCC 电压

read_adc(5) --5 - TVCC 电流

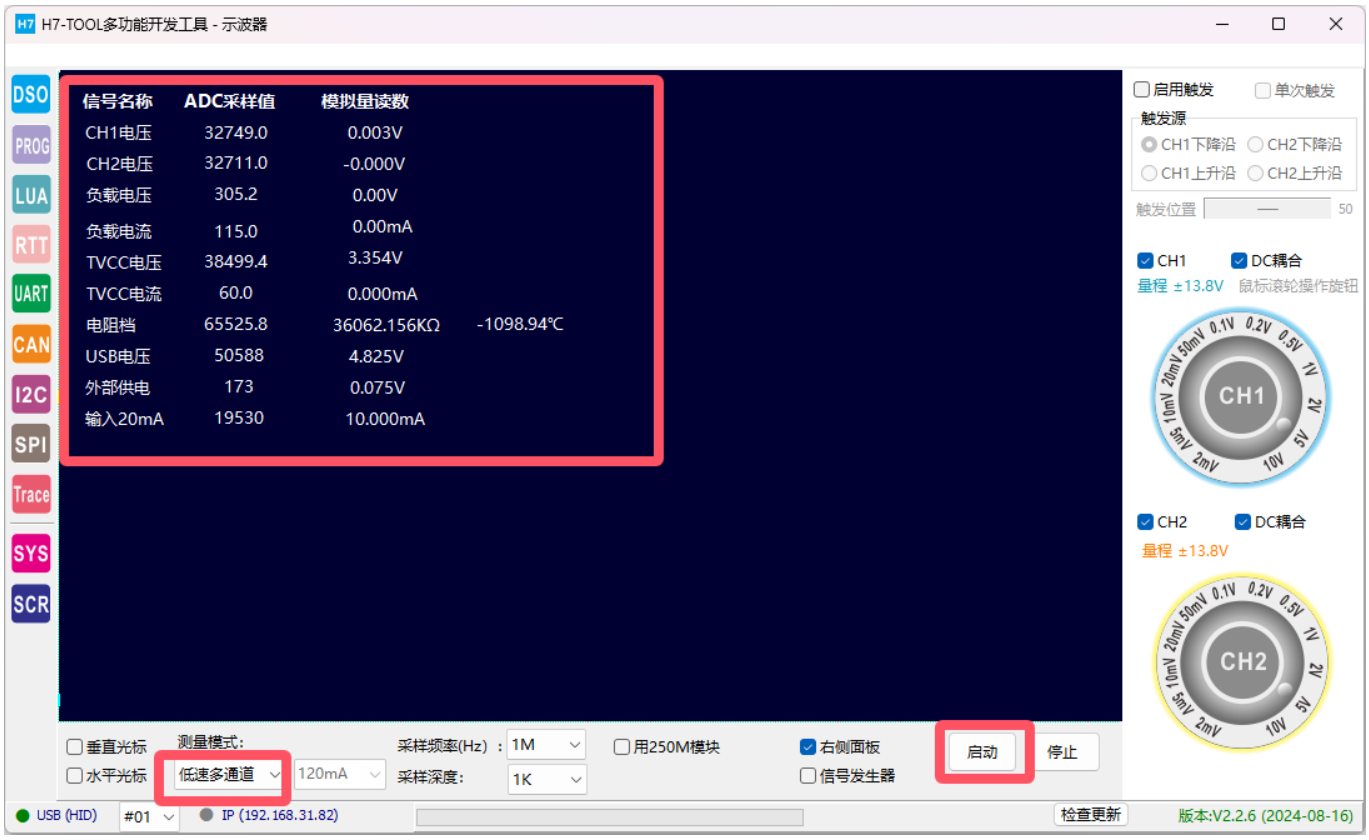
read_adc(6) --6 - NTC 热敏电阻阻值

read_adc(7) --7 - 外部供电电压

read_adc(8) --8 - USB 供电电压

read_adc(9) -- 9 - 4-20mA 输入

5.15.11. 使用上位机同时展示 read_adc 和 read_analog



5.16. 扩展模块

5.16.1. 简介

扩展模板支持了脉冲测量，ADS1256，MCP3421，8 路继电器和 5 路 DS18B20

5.16.2. m_init 初始化模组

5.16.3. m_uninit 复位模组

5.16.4. m_read 读数据

5.16.5. m_write 写数据

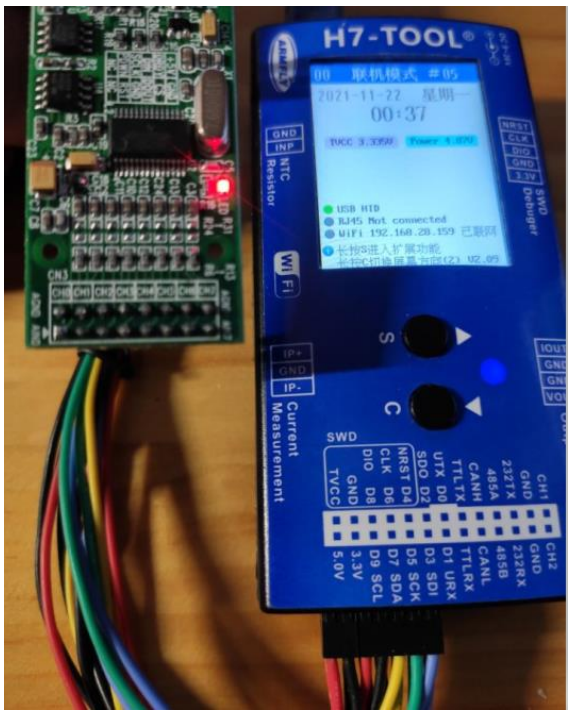
针对每个扩展模块，这几个函数使用都不同，所以我们这里分模块进行说明

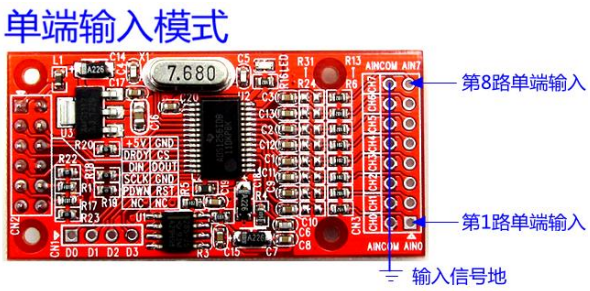
5.16.6. 驱动 8 通道 24bit 分辨率 ADC ADS1256

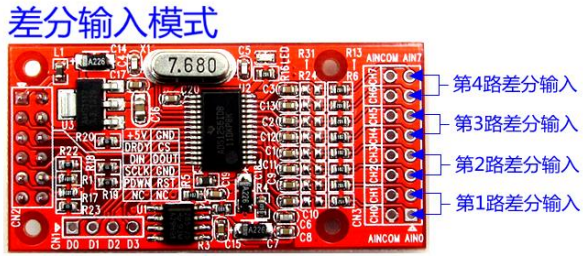
需要深入了解 ADS1256 可以看教程第 93 章节：
http://www.armbbs.cn/forum.php?mod=viewthread&tid=86980

1、 ADS1256 接线说明：

ADS1256 模块　　H7-TOOL　　(TVCC 输出 3.3V)
+5V　　<------　　5.0V　　5V 供电
GND　　-------　　GND　　地
CS　　<------　　D0　　SPI_CS　　PI0
DRDY　　------>　　D1　　准备就绪（用到外部中断）PH10　　--所有的 IO 电平固定是 3.3V
DIN　　<------　　D2　　SPI_MOSI　PE6
DOUT　　------>　　D3　　SPI_MISO　PE5
SCLK　　<------　　D5　　SPI 时钟　PE2
GND　　-------　　GND　　地
PDWN　　（模块有上拉，可不接线）
RST　　（模块有上拉，可不接线）
NC　　空脚
NC　　空脚







2、ADS1256 测试代码

使用比较简单，注释很详细，使用也比较简单。

```
local gain = 0     -- 增益 （0=X1,  1=X2,  2=X4,  3=X8, 4=X16, 5=X32, 6=X64）
local drate = 7     --数据率 (0=30000,1=15000,2=7500,3=3750,4=2000,5=1000,6=500,7=100, 8=60, 9=50, 10=30, 11=25, 12=15,
13=10, 14=5,  15=2.5sps)
local buffen = 1  -- 1 表示使能输入 buffer 使能，0 表示禁止
local diffmode = 0  -- 0 表示单端模式，1 表示差分模式
local scantable = "01234567"  --扫描次序, 可只扫描需要的通道

m_init("ADS1256", gain, drate, buffen, diffmode, scantable)  -- 挂载 ADS1256

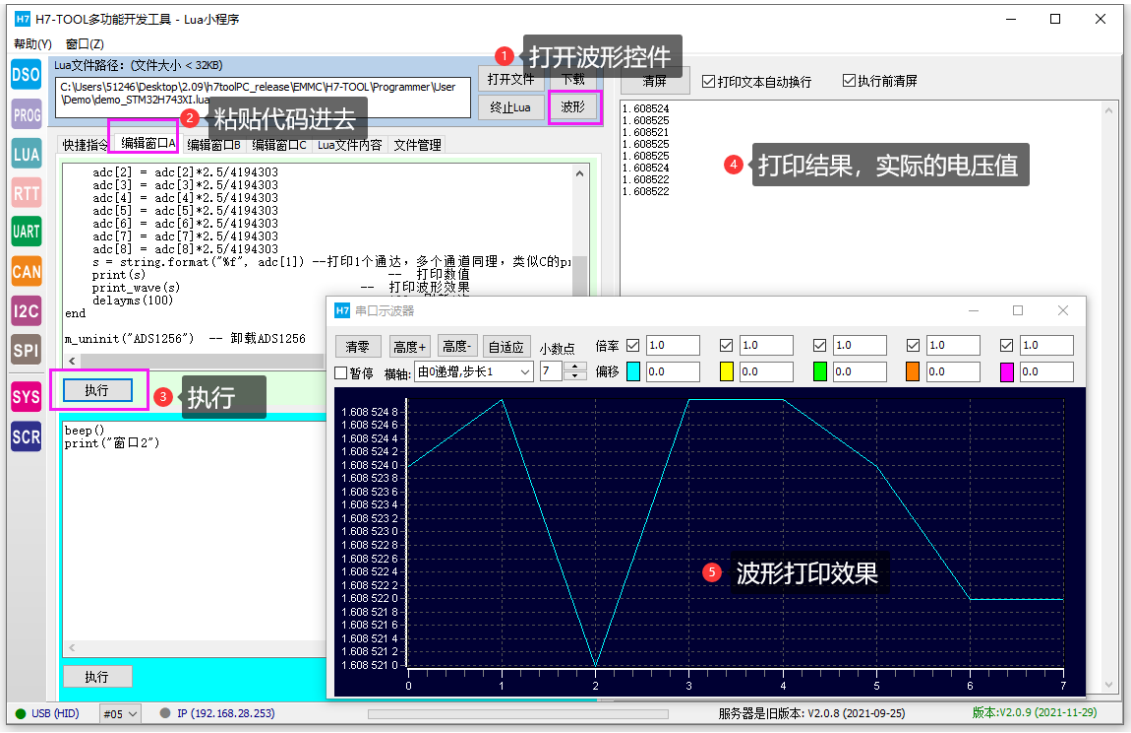
delayms(500)

for j=1, 8, 1 do                                  --  循环执行了 8 次
    adc = {m_read("ADS1256")}     --  读取数据
```

```
    adc[1] = adc[1]*2.5/4194303    --  8 通道的数据全部读取，直接计算实际电压值。
    adc[2] = adc[2]*2.5/4194303
    adc[3] = adc[3]*2.5/4194303
    adc[4] = adc[4]*2.5/4194303
    adc[5] = adc[5]*2.5/4194303
    adc[6] = adc[6]*2.5/4194303
    adc[7] = adc[7]*2.5/4194303
    adc[8] = adc[8]*2.5/4194303
    s = string.format("%f", adc[1]) --打印 1 个通达，多个通道同理，类似 C 的 printf
    print(s)                                    --  打印数值
    print_wave(s)                           --  打印波形效果
    delayms(100)                            --  100ms 刷新 1 次
end

m_uninit("ADS1256")  -- 卸载 ADS1256
```

将代码贴到 Lua 小程序窗口里面执行即可：



5.16.7. 驱动 18bit 分辨率 ADC MCP3421

这个帖子有个早期使用 I2C 控制驱动说明，可以优先看下：
https://www.armbbs.cn/forum.php?mod=viewthread&tid=109658

使用专用的 API 驱动实现如下：

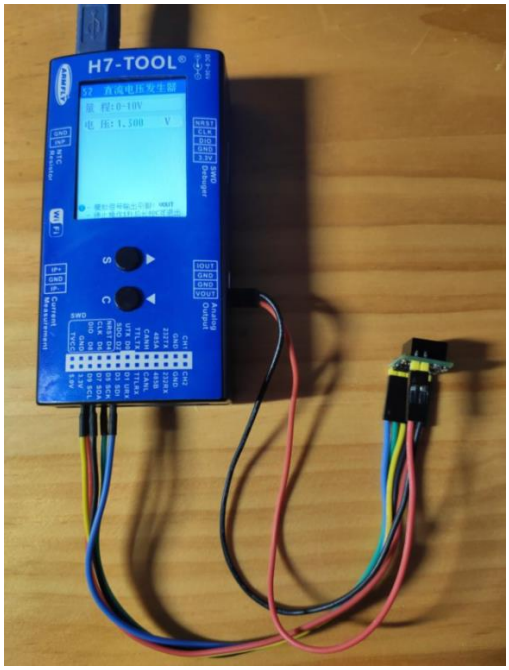
```
local gain = 0   --0=X1 1=X2 2=X4 3=X8
local drate = 3  --0=240sp(12bit)  1=60sps(14bit) 2=15sps(16bit) 3=3.75sps(18bit)
m_init("MCP3421", gain, drate)

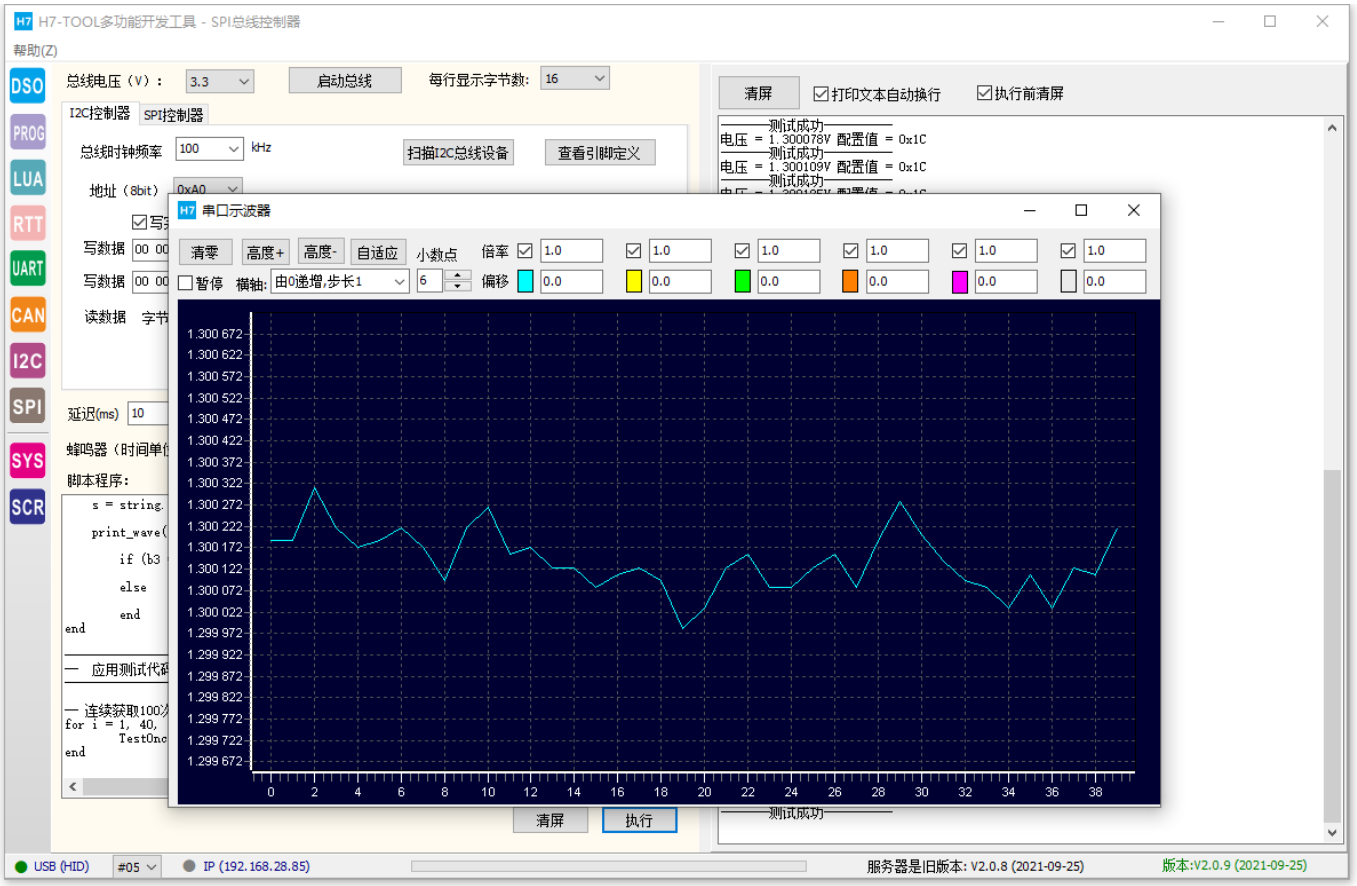
delayms(200)
```

```
adc = m_read("MCP3421")  print(adc) --读 ADC 数据
```



同时将 TOOL 的信号发生器输出 1.4V 电压让 MCP3421 模块测试：

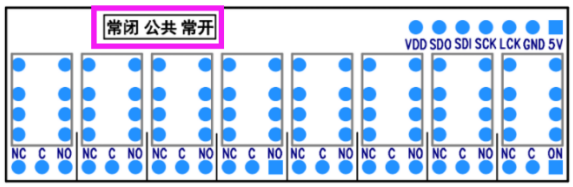




5.16.8. 驱动 8 路继电器

1、驱动模块说明：

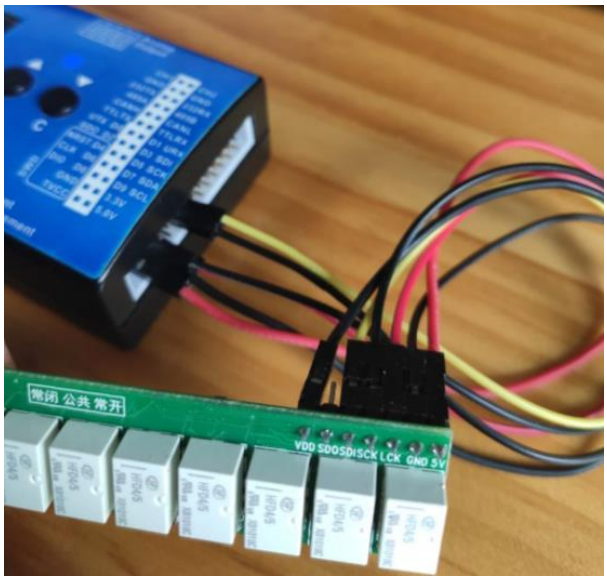
Lua 驱动 74HC595，可配置任意引脚、595 芯片个数。发布 74HC595_8 路继电器模块
https://www.armbbs.cn/forum.php?mod=viewthread&tid=110038



2、TOOL 控制继电器模块的接线：

H7-TOOL	继电器
5.0V	5.0
3.3V	VDD
GND	GND
D4	SCK
D2	SDI
D9	LCK

实际接线效果：

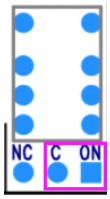


3、控制说明：

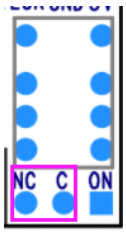
```
m_init("74HC595", 4, 2, 9, 1)    --    配置 D4 为 SCK 时钟,    D2 为 SDI 数据, D9 为 LCK 锁存,    1 个 74HC595 芯片
m_write("74HC595", 2,    1)    --    设置 74HC595 的 Q2 = 1    (第 2 个形参是起始 bit, 第 3 个形参值。 支持多个形参)
m_write("74HC595", 2,    1,0) --设置 74HC595 的 Q2 = 1    Q3 = 0
m_write("74HC595", 0,    1,0,1,0,    0,0,0,0)    --    设置 74HC595 的 Q0-7 分别为 1,0,1,0,    0,0,0,0
```

注意一点就行，第 2 个形参是设置的起始继电器，范围从 0 到 7。后面的参数根据大家的设置起始继电器，设置继电器的开关。

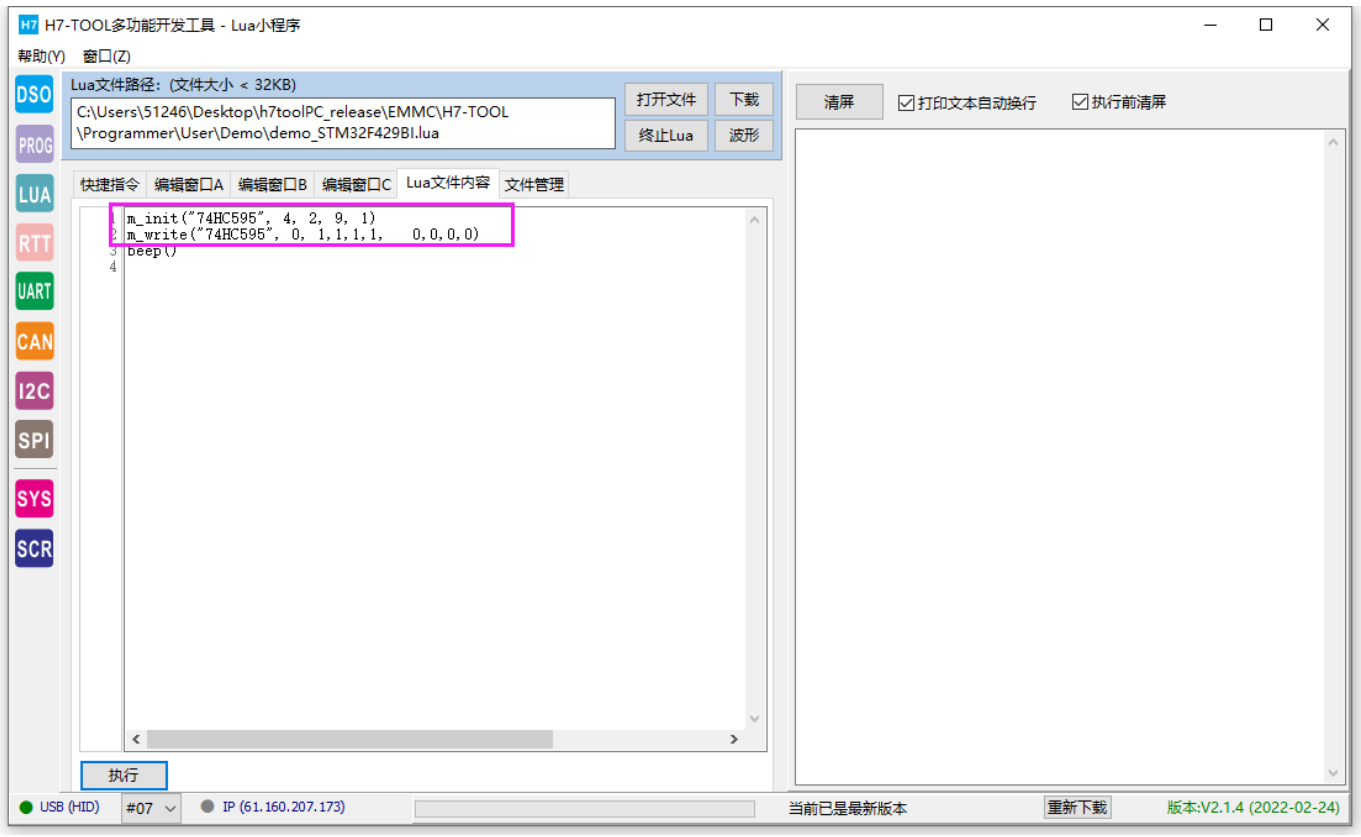
1 表示这两个导通：



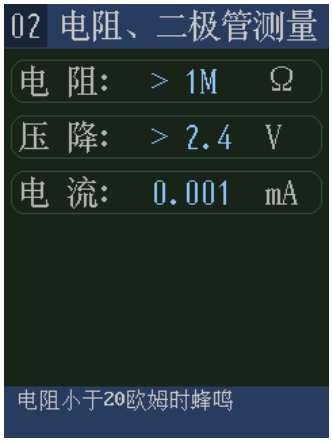
0 表示这两个导通：



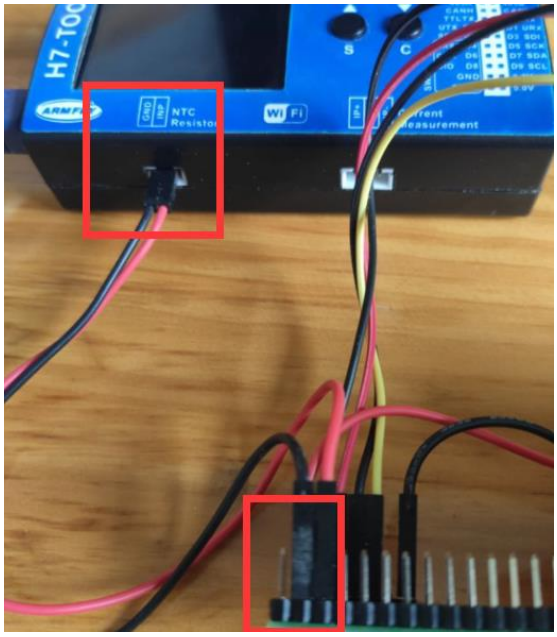
比如我们控制前 4 个为 1，后 4 个为 0，就是如下效果：



验证是否设置正常，进入 TOOL 的电阻，二极管测量界面即可：



接线如下：



可以方便的测量继电器的闭合状态。

5.16.9. 驱动 5 路 DS18B20

使用这个功能，有必要优先看下 TOOL 端展示和使用方法：
https://www.armbbs.cn/forum.php?mod=viewthread&tid=106958

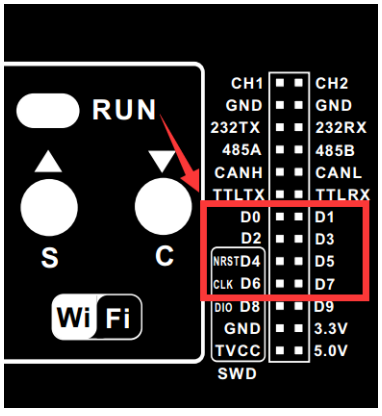
使用专用的 API 驱动实现如下：

```
m_init("DS18B20")

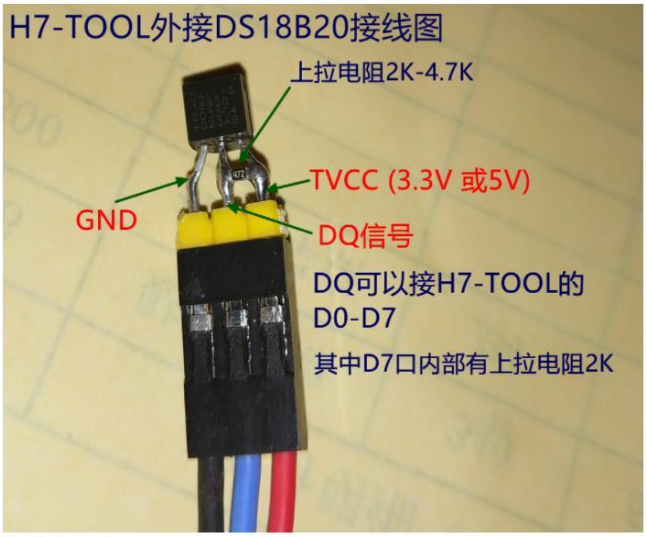
re,temp = m_read("DS18B20", 0)  --读 D0 口 DS18B20 的温度值，  re = 1 表示 OK
re,temp = m_read("DS18B20", 2)  --读 D2 口 DS18B20 的温度值，  re = 1 表示 OK

m_uninit("DS18B20") --退出程序
```

1、H7-TOOL 支持外接 5 路 DS18B20，通过 D1，D3，D5，D7 和 D9 外接 DS18B20 数据引脚，而 18B20 的 GND 和 VCC 接到 TOOL 的 GND 和 TVCC 供电上即可。



2、其中 D7 和 D9 是内置了上拉电阻的，大家可以直接外接 DS18B20 测试，如果使用 D1，D3 和 D5，采用下面方式即可。



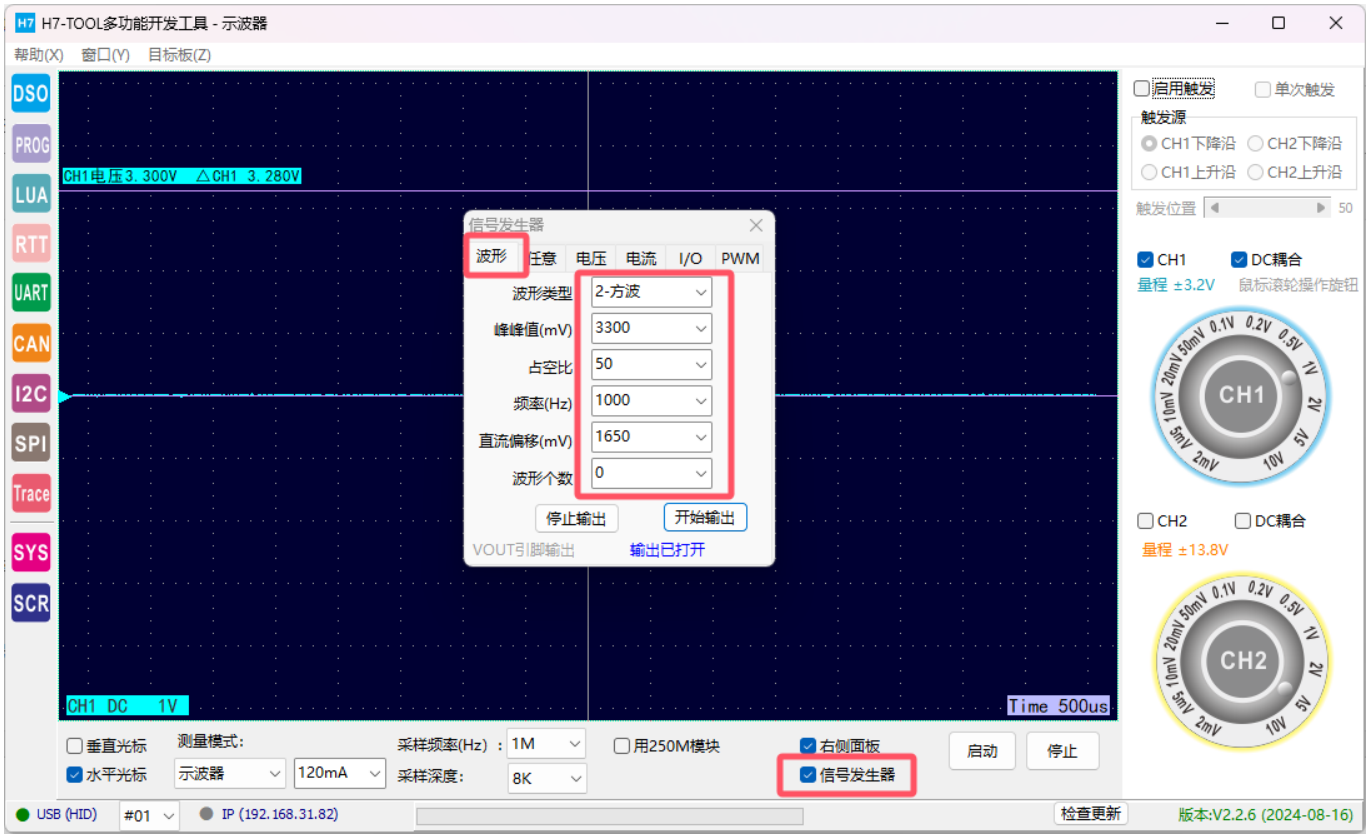
5.16.10. 脉冲测量

H7-TOOL 的 D1 接口，可以测量输入脉冲的频率、占空比、个数。

下面是 lua 程序用法:

```
m_init("DETPULSE")  --进入 D1 测量脉冲模式
count, freq, duty = m_read("DETPULSE") --返回脉冲个数(整数)、频率(Hz 浮点数)、占空比(% 浮点数)
m_write("DETPULSE", 0,0,0)  --清零脉冲个数、频率、占空比
m_uninit("DETPULSE")  --退出 D1 测量脉冲模式
```

比如我们这里使用 D1 端口测量 1KHz 的脉冲信号。直接使用 TOOL 的信号发生器功能输出即可，峰峰值 3.3V，直流偏置 1.65，频率 1KHz



测试代码：

```
m_init("DETPULSE")  --进入 D1 测量脉冲模式
```

```
set_tvcc(3.30)  --设置电平为3.3V

print("测试D2输出高电平")

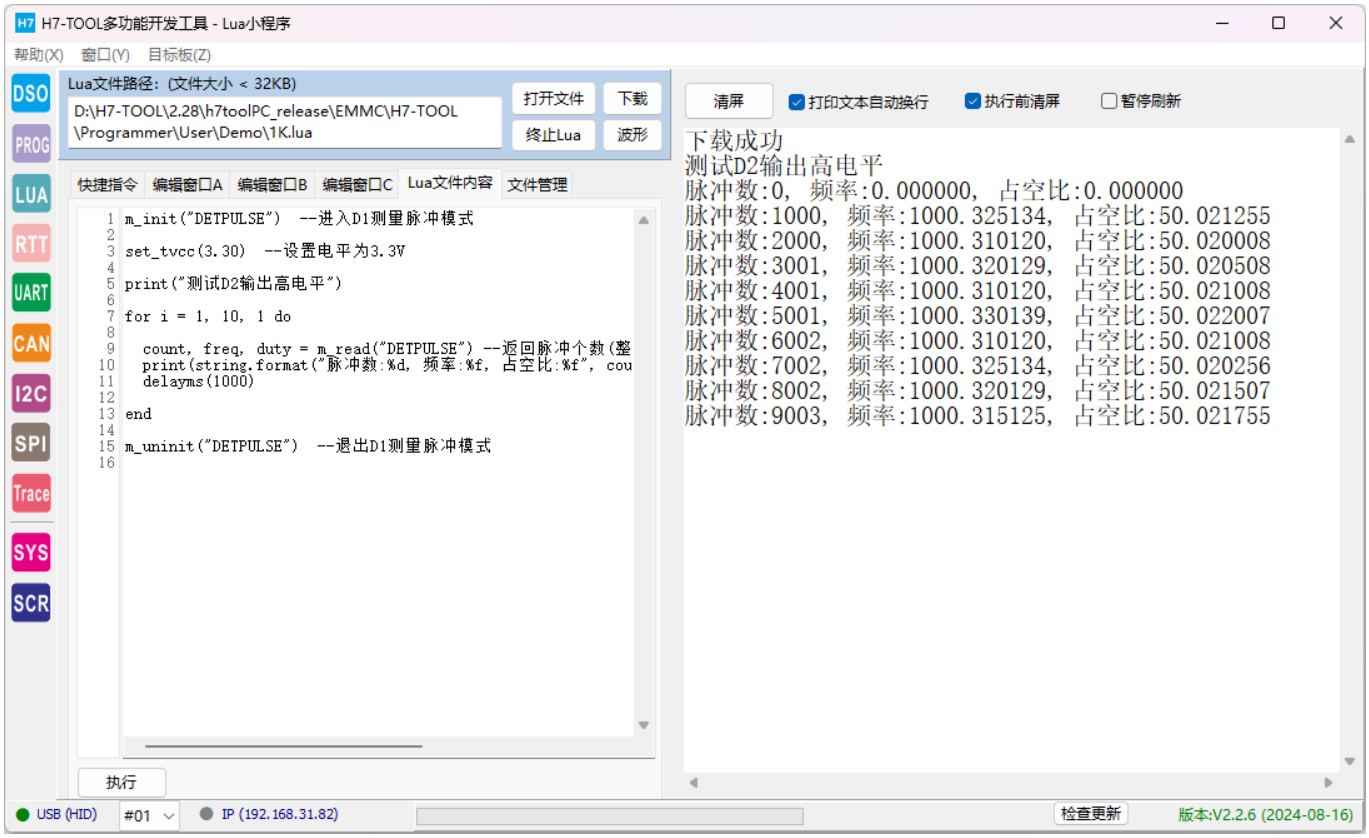
for i = 1, 10, 1 do

  count, freq, duty = m_read("DETPULSE") --返回脉冲个数(整数)、频率(Hz 浮点数)、占空比(% 浮点数)
  print(string.format("脉冲数:%d, 频率:%f, 占空比:%f", count, freq, duty))
  delayms(1000)

end

m_uninit("DETPULSE")  --退出D1测量脉冲模式
```

实际效果：



5.17. IO、ADC、DAC 扩展

5.17.1. 简介

当前这个驱动方式仅做了 AD7606 的支持

5.17.2. ex_start 启动函数

5.17.3. ex_stop 停止函数

5.17.4. ex_dout 输出函数

5.17.5. ex_din 输入函数

5.17.6. ex_dac DAC 配置

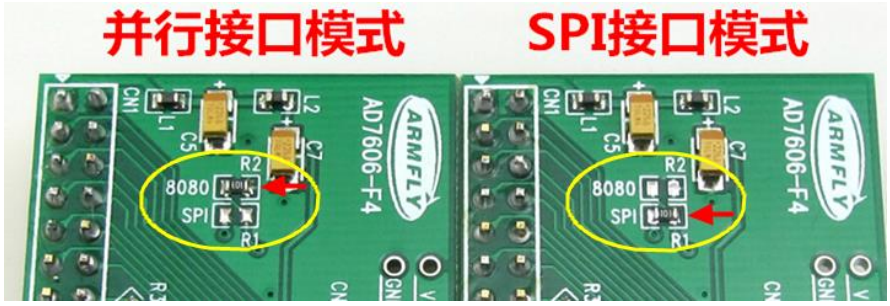
5.17.7. ex_adc ADC 配置

针对每个扩展模块，这几个函数使用都不同，所以我们这里分模块进行说明

5.17.8. 驱动 8 通道 16bit 分辨率 ADC AD7606

需要深入了解 AD7606 可以看教程第 76 或 77 章节：
http://www.armbbs.cn/forum.php?mod=viewthread&tid=86980

1、设置 AD7606 模块是 SPI 接口：



2、AD7606 SPI 接线说明：

AD7606 模块　　H7-TOOL
+5V　　<------　　5.0V　　5V 供电
GND　　-------　　GND　　地
CS　　<------　　D7　　片选控制
RD/SCLK　　<------　　D5　　时钟线
DB7/DOUT　　------>　　D3　　数据输出
BUSY　　------>　　D6　　BUSY 信号
CVA CVB　　<------　　D0，　　启动转换

RAGE <------ 可以接 3.3V 或者 GND 控制，3.3 表示±10V 测量范围，GND 表示±5V 测量范围。

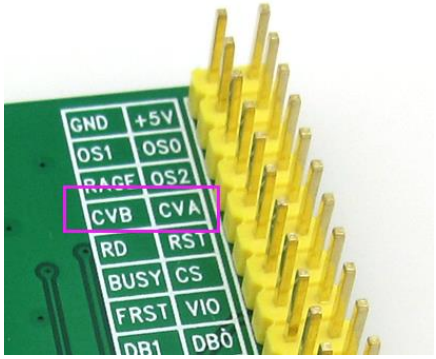
RST <------ 不接

OS0 <------ 可以接 3.3V 或者 GND 控制， 过采样引脚 0(默认接地)，最好接上

OS1 <------ 可以接 3.3V 或者 GND 控制，过采样引脚 1(默认接地)，最好接上

OS2 <------ 可以接 3.3V 或者 GND 控制， 过采样引脚 2(默认接地)，最好接上

其中 CVA 和 CVB 可以考虑短接



最终接线效果：



3、AD7606 测试代码：

```
adc = {} --定义个数组

ex_start()    -- 启动 AD7606

delayms(100)
```

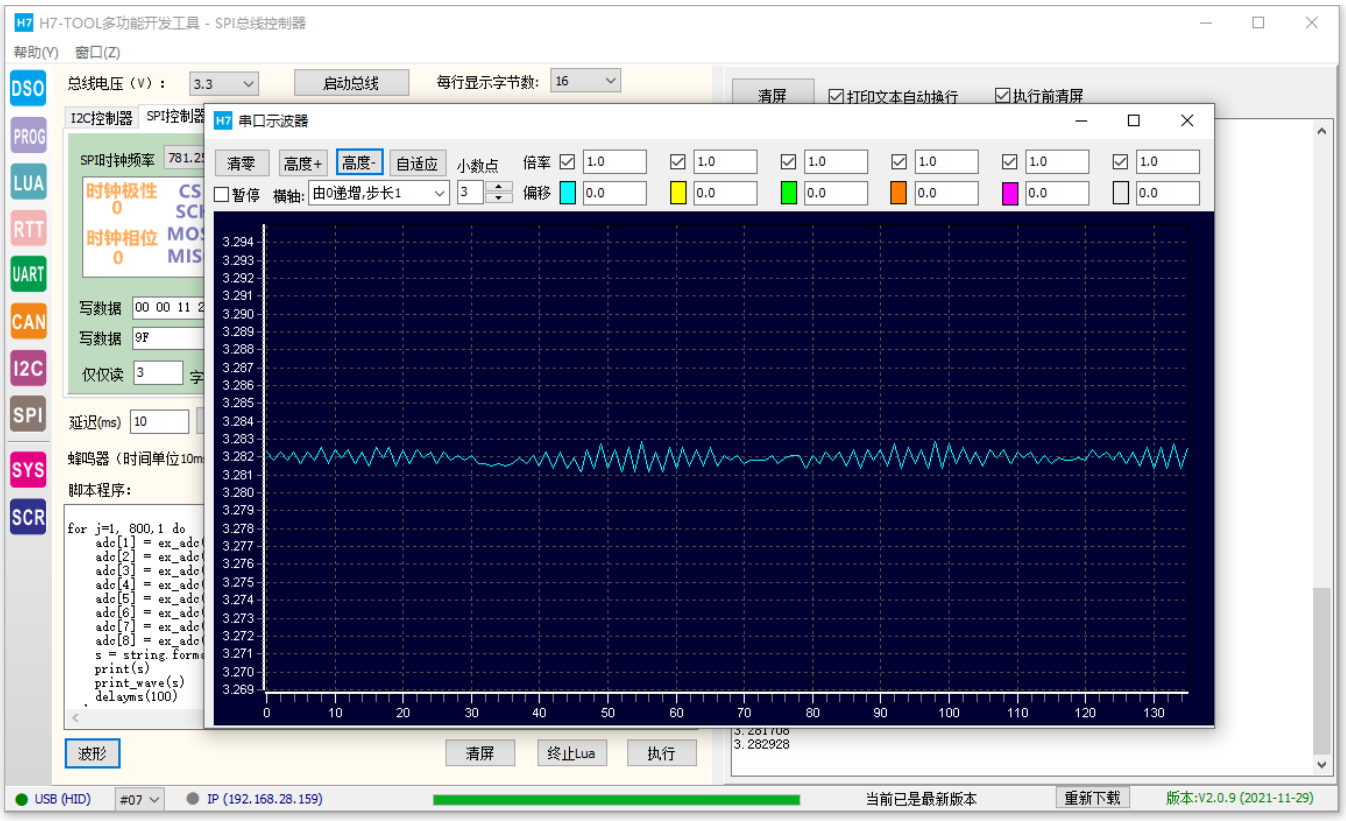
```
for j=1, 800,1 do                    -- 循环 800 次
    adc[1] = ex_adc(0)*5/32768
    adc[2] = ex_adc(1)*5/32768
    adc[3] = ex_adc(2)*5/32768
    adc[4] = ex_adc(3)*5/32768
    adc[5] = ex_adc(4)*5/32768
    adc[6] = ex_adc(5)*5/32768
    adc[7] = ex_adc(6)*5/32768
    adc[8] = ex_adc(7)*5/32768
    s = string.format("%f", adc[8]) --打印 1 个通达，多个通道同理，类似 C 的 printf
    print(s)                         --  打印数值
    print_wave(s)                    --  打印波形效果
    delayms(100)                     --   100ms 刷新 1 次
end

ex_stop() -- 卸载 AD7606
```

将代码贴到 Lua 小程序窗口里面执行即可：



测试 3.3V 的效果：


H7-TOOL多功能开发工具 - SPI总线控制器
帮助(Y)
窗口(Z)
DSO
PROG
LUA
RTT
UART
CAN
I2C
SPI
SYS
SCR
总线电压（V）：
3.3
启动总线
每行显示字节数：
16
清屏
打印文本自动换行
执行前清屏
I2C控制器
SPI控制器
SPI时钟频率
781.25
时钟极性
0
时钟相位
0
写数据
00 00 11 2
写数据
9F
仅仅读
3
延迟(ms)
10
蜂鸣器（时间单位10m
脚本程序：
for j=1, 800,1 do
adc[1] = ex_adc(
adc[2] = ex_adc(
adc[3] = ex_adc(
adc[4] = ex_adc(
adc[5] = ex_adc(
adc[6] = ex_adc(
adc[7] = ex_adc(
adc[8] = ex_adc(
s = string.forma
print(s)
print_wave(s)
delayms(100)
波形
清屏
终止Lua
执行
3.282928
USB (HID)
#07
IP (192.168.28.159)
当前已是最新版本
重新下载
版本:V2.0.9 (2021-11-29)
串口示波器
清零
高度+
高度-
自适应
小数点
倍率
暂停
横轴:
由0递增,步长1
偏移
3.294
3.293
3.292
3.291
3.290
3.289
3.288
3.287
3.286
3.285
3.284
3.283
3.282
3.281
3.280
3.279
3.278
3.277
3.276
3.275
3.274
3.273
3.272
3.271
3.270
3.269
0
10
20
30
40
50
60
70
80
90
100
110
120
130

5.18. LUA 小程序进行产品批量测试实战

5.18.1. 出厂批量测试 RS485 数码管产品

https://www.armbbs.cn/forum.php?mod=viewthread&tid=98463



5.18.2. 出厂批量测试 10 几款 IO 控制器和脱机烧录一条龙

https://www.armbbs.cn/forum.php?mod=viewthread&tid=97285

5.18.3. 出厂批量测试隔离型 TTL 转 RS485/RS232 模组

https://www.armbbs.cn/forum.php?mod=viewthread&tid=97306

5.18.4. LUA 小程序长时间运行稳定性研究

https://www.armbbs.cn/forum.php?mod=viewthread&tid=117869