



AirM2M

模块 Lua 程序设计、调试指南 V1.9

目 录

1.	Lua 简介.....	4
2.	AirM2M GPRS 模块简介.....	4
3.	模块程序和 Lua 程序之间的关系.....	4
4.	Lua 的安装使用以及 Lua 语法简介.....	6
4.1	Lua Windows 开发环境的安装.....	6
4.2	Lua Windows 开发环境的使用.....	6
4.3	Lua 语法简介.....	7
5.	Lua 模块应用程序开发架构介绍.....	15
5.1	main.lua 主程序.....	15
5.2	主架构 run()详解.....	16
5.3	lib 库中的各个模块.....	17
5.4	应用模块.....	18
5.5	模块之间的调用关系.....	18
6.	程序注册.....	18
6.1	regmsg()用来注册相应的消息处理程序.....	18
6.2	regapp()用来注册应用程序.....	21
6.3	regurc()用来注册某些 URC 相应的处理程序.....	23
6.4	regrsp()用来注册 AT 命令处理程序.....	24
6.5	reguart()用来注册 uart 口的数据处理程序.....	25
7.	各模块详细介绍.....	27
7.1	sys 模块详解.....	27
7.2	ril 模块详解.....	30
7.3	link 模块详解.....	31
7.4	sms 模块详解.....	34
7.5	cc 模块详解.....	35
7.6	net 模块详解.....	37
7.7	pm 模块详解.....	38
7.8	common 模块详解.....	39
7.9	pb 模块详解.....	40
7.10	audio 模块详解.....	41
7.11	misc 模块详解.....	45
7.12	gps 模块详解.....	47
8.	消息分发函数 dispatch 详细介绍.....	50
8.1	cc 模块中的 dispatch.....	51
8.2	net 模块中的 dispatch.....	52
8.3	sms 模块中的 dispatch.....	52
8.4	pb 模块中的 dispatch.....	53
8.5	audio 模块中的 dispatch.....	53
8.6	update 模块中的 dispatch.....	54
8.7	misc 模块中的 dispatch.....	54
9.	快速入手.....	54
10.	例程详解.....	56

10.1	定时器如何使用.....	56
10.2	如何映射键盘.....	56
10.3	如何加入一个按键处理程序.....	56
10.4	接收短信，并处理短信内容.....	57
10.5	如何建立一个 TCP 连接.....	57
10.6	如何加入 Lcd 驱动.....	58
10.7	如何在界面显示文字和图片.....	59
10.8	如何点亮各种灯.....	59
10.9	如何使马达震动.....	60
10.10	如何使用电池管理消息.....	60
10.11	如何使模块休眠和唤醒.....	60
10.12	如何使用 debug uart 串口与上位机通讯.....	61
10.13	如何使用物理串口与外部设备通讯.....	61
11.	程序调试环境.....	62
11.1	真实调试环境.....	63
11.2	PC 模拟调试环境.....	66

1. Lua 简介

Lua 是一个小巧的脚本语言。脚本语言不需要事先编译，直接可以运行（其实是在运行的时候进行解释）。该语言的设计目的是为了嵌入应用程序中，从而为应用程序提供灵活的扩展和定制功能。

Lua 由标准 C 编写而成，代码简洁优美，几乎在所有操作系统和平台上都可以解释，运行。

一个完整的 Lua 解释器不过 200k，在目前所有脚本引擎中，Lua 的速度是最快的。这一切都决定了 Lua 是作为嵌入式脚本的最佳选择。

Lua 使用者分为三大类：使用 Lua 嵌入到其他应用中的、独立使用 Lua 的、将 Lua 和 C 混合使用的。

第一：很多人使用 Lua 嵌入到其他应用程序。

我司模块中支持 Lua 程序开发和应用就属于这种情况。用户将自行开发的 lua 应用程序嵌入在我模块基础软件 Lod 之中。

第二：作为一种独立运行的语言，Lua 也是很有用的，主要用于文本处理或者只运行一次的小程序。这种应用 Lua 主要使用它的标准库来实现，标准库提供模式匹配和其它一些字符串处理的功能。我们可以这样认为：Lua 是文本处理领域的嵌入式语言。

第三：还有一些使用者使用其他语言开发，把 Lua 当作库使用。这些人大多使用 C 语言开发，但使用 Lua 建立简单灵活易于使用的接口。

2. AirM2M GPRS 模块简介

AirM2M GPRS 无线模块（后续简称 Air 模块），支持 4 频(900/1800MHz 850/1900MHz)，为相关无线接入应用提供无线数据传输承载业务服务，并支持语音，短信，电话本，STK，MMS，HTTP，文件操作，音频播放等功能。

AirM2M GPRS 无线模块，支持完整的 AT 指令接口，上述所有支持的功能均通过 AT 命令来实现。AT，即 ATtension，是以 AT 作首，以字符结束的字符串。每个指令执行成功与否都有相应的返回。

Air 模块提供的 AT 命令包含符合 GSM07.05、GSM07.07 和 ITU-T Recommendation V.25ter 的命令，以及开发的 Air 模块专有命令。

3. 模块程序和 Lua 程序之间的关系

Lua 脚本是内嵌在模块基础软件(简称 Lod)中运行的，lod 中有支持 Lua 运行的环境，脚本就在这个环境中运行。脚本实现功能是通过 AT 命令实现的。AT 命令的来源有 2 个：

1) 在 Air 模块内部, Lua 发出 AT 命令, 并通过和 Lod 之间虚拟的 uart.ATC 口进行 AT 命令的交互。即 Lua 发出 AT 命令, Lod 返回 AT 命令运行结果, 不需要上位机 (一般是单片机) 通过物理串口给模块发 AT 命令, 这样就节省了单片机的花费。

这样 Air 模块的物理串口就可以挪作他用, 比如连接 GPS 芯片, 或 PC 工具软件。Air 模块还有一个 DEBUG UART 口, 可以连接 PC 工具软件等。

2) 外设或 PC 工具等发出 AT 命令, Lua 脚本通过物理串口或 Debug 串口接收到这些 AT 命令后, 再通过 ATC 口转发给 Lod。

模块程序 Lod 和 Lua 程序的关系示意图如下:

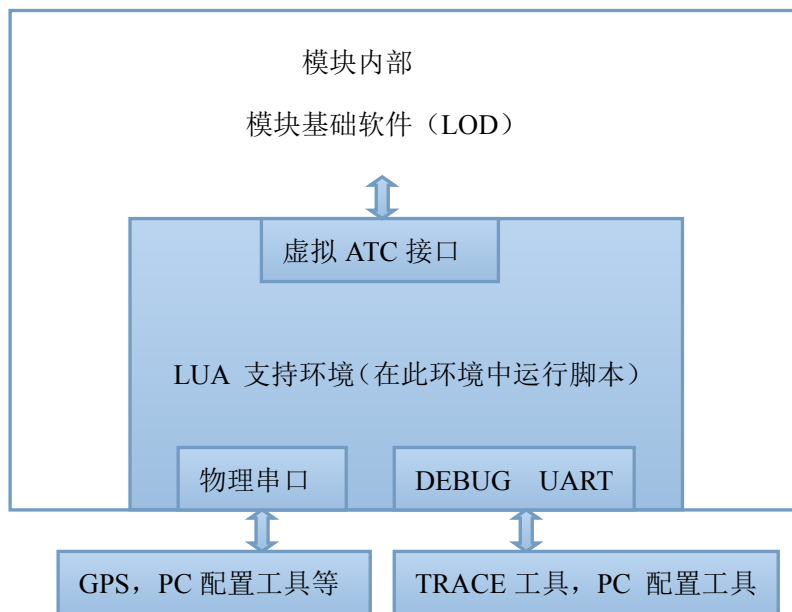


图 模块程序和 Lua 程序的关系示意图

4. Lua 的安装使用以及 Lua 语法简介

4.1 Lua Windows 开发环境的安装

LuaForWindows_v5.1.4-46.exe, 是 lua 在 windows 下的编辑和运行环境。
下载地址: <http://code.google.com/p/luaforwindows/downloads/list>

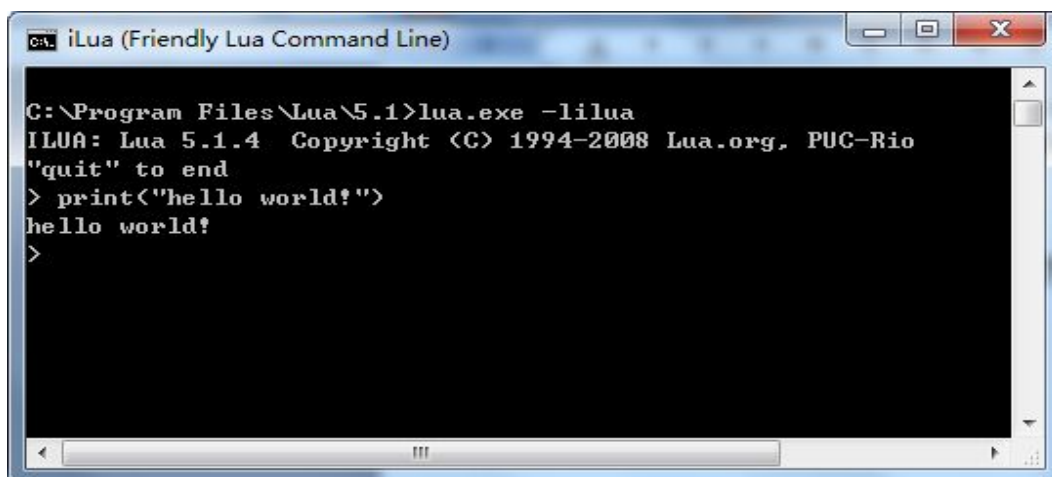
点击这个安装文件, 出现下图所示的 Setup Wizard:



使用缺省配置, 一路 Next 下去, 直到安装完成。

4.2 Lua Windows 开发环境的使用

Lua 的 Windows 开发环境安装好以后, 在 WINDOWS 开始菜单->程序->Lua 中可以找到交互式命令行窗口。如下图:



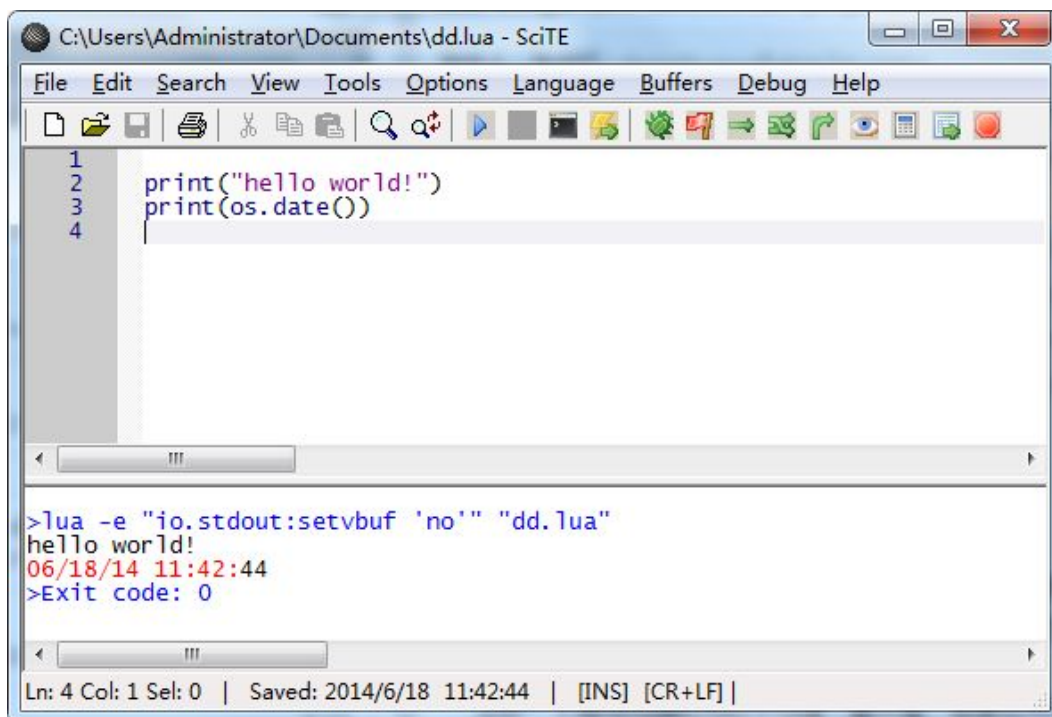
交互式命令行好处是马上就能得到结果，坏处是没法保存，下次还想运行的时候，还得再敲一遍。


所以，实际开发的时候，我们总是使用一个文本编辑器来写代码，写完了，保存为一个文件，这样，程序就可以反复运行了。

Lua Windows 开发环境也自带有一个编辑器 SciTE，安装完成后会将 SciTE 快捷方式图标放在桌面上。



双击打开后如下图：



编辑一段代码，保存为.lua 为后缀名的文件，然后点击运行按钮 ，在下面的窗口会打印运行结果。

4.3 Lua 语法简介

4.3.1 词法约定

变量是以字母(letter)或者下划线开头的字母、下划线、数字序列。请不要使用下划线加大写字母的标示符，因为 Lua 的保留字也是这样的。

以下字符为 Lua 的保留字，不能当作变量使用。

and	break	do	else	elseif
end	false	for	function	if
in	local	nil	not	or
repeat	return	then	true	until
while				

注意：Lua 是大小写敏感的。如 `and` 是保留字，但是 `And` 和 `AND` 则不是 Lua 的保留字。

在 SciTE 中输入保留字的时候，会自动变为蓝色粗黑体。

4.3.2 代码规范

◆ 书写规范

Lua 的多条语句之间并不要求任何分隔符，如 C 语言的分号(;)，其中换行符也同样不能起到语句分隔的作用。因此下面的写法均是合法的。如：

```
a = 1
b = a * 2

a = 1;
b = a * 2;

a = 1; b = a * 2;
a = 1 b = a * 2
```

◆ 注释

Lua 代码中的注释分为 2 种：

一种是单行注释，如：

```
-- This is a single line comment
```

另一种是多行注释，如：

```
--[[
This is a multi-lines comment
--]]
```

4.3.3 变量

Lua 变量分为**全局**变量和**局部**变量。

全局变量不需要声明，给一个变量赋值后即创建了这个全局变量，全局变量在整个文件中起作用。访问一个没有初始化的全局变量也不会出错，只不过得到的结果是：`nil`。


```

print(b)                --> nil
b = 10
print(b)                --> 10
如果你想删除一个全局变量，只需要将变量赋值为 nil
b = nil
print(b)                --> nil

```

局部变量通过 **local** 来声明。与全局变量不同，局部变量只在被声明的那个代码块内有效。代码块：指一个控制结构内，一个函数体，或者一个 **chunk**（变量被声明的那个文件或者文本串）。

例如：

```
local m = 9
```

```

if m <= 10 then
    local m = 5          -- 这个局部变量 m 的作用范围是 then 和 end 之间
    print(m)
end
运行结果是 5.

```

```
local m = 9
```

```

if m <= 10 then
    print(m)             -- 这个 m 是一开始声明的局部变量 m
end
运行结果是 9.

```

另外 **Lua** 是动态类型语言，声明变量的时候不要类型定义。变量的类型和当前所赋值的类型是一致的。我们可以通过 **type** 函数获得变量的或值的类型信息，该类型信息将以字符串的形式返回。

例如：

```

local k = 1              -- k 是局部变量
print(type(k))           --> number

```

```

k = "today is Friday"    -- 将一个字符串赋值给 k
print(type(k))           --> string

```

```

k = {1,2}                -- 将一个表赋值给 k
print(type(k))           --> table

```

```

local d                  -- d 是局部变量，声明的时候未赋值
print(type(d))           --> nil
d = 11                   -- d 后来被赋值
print(type(d))           --> number

```

```
h = 9                    -- h 是全局变量
```

```
print(type(h))          --> number
```

4.3.4 值和类型

Lua 中的值有 8 种类型：nil、boolean、number、string、userdata、function、thread 和 table。

◆ nil 型

Lua 中特殊的类型，它只有一个值：nil，它的主要功能是由于区别其他任何值。一个全局变量没有被赋值以前默认值为 nil；给全局变量赋 nil 可以删除该变量。Lua 将 nil 用于表示一种“无效值”的情况。

◆ boolean 型

该类型有两个可选值：false 和 true。但要注意 Lua 中所有的值都可以作为条件。在 Lua 中只有当值是 false 和 nil 时才视为“假”，其它值均视为真，如数字零和空字符串，这一点和 C 语言是不同的。

◆ number 型

表示实数。由于资源所限，我司提供的模块 Lua 开发平台不支持小数和浮点型的 number 类型，目前只支持整数。

◆ string 型

指字符的序列。lua 是 8 位字节，所以字符串可以包含任何数值字符，包括嵌入的 0。这意味着你可以存储任意的二进制数据在一个字符串里。

例如：

```
m = "abc"
n = "\97\98\99"          -- "a"的数值是 97(decimal), "b"是 98, "c"是 99
print(m,n)               --> abc  abc
print("\09798")          --> a98
```

string 和其他对象一样，Lua 自动进行内存分配和释放，一个 string 可以只包含一个字母也可以包含一本书，Lua 可以高效的处理长字符串，1M 的 string 在 Lua 中是很常见的。可以使用单引号或者双引号把一串字符括起来表示字符串：

```
a = "hello"
b = `world`
```

为了风格统一，最好使用一种。例如一直使用双引号来表示字符串。如果双引号之间有单引号，系统会自动识别此单引号；如果双引号中之间还有双引号，则里面的双引号需要用转义字符\来区别：

例如：

```
print("one line\nnext line\n\"in quotes\", 'in quotes')
```

运行结果是：

```
one line
next line
"in quotes", 'in quotes'
```

另外记住： Lua 中的字符串是恒定不变的。String.sub 函数以及 Lua 中其他的字符串操作函数都不会改变字符串的值，而是返回一个新的字符串。一个常见的错误是：

```
string.sub(s, 2, -2)
```

认为上面的这个函数会改变字符串 `s` 的值。其实不会。

如果你想修改一个字符串变量的值，你必须将变量赋给一个新的字符串：

```
s = string.sub(s, 2, -2)
```

◆ **table 型**

table 是 Lua 中唯一的数据结构，其他语言所提供的数据结构，如：数组、矩阵、链表、队列等，Lua 都是通过 **table** 来很好地实现。

Lua 中 **table** 的键(key)可以为任意类型(nil 除外)。当通过整数下标访问 **table** 中元素时，即是数组。

此外，**table** 没有固定的大小，可以动态的添加任意数量的元素到一个 **table** 中。例如：

```
local t = {32,"jeep", name = "cherry"}
local p = {"name" = "peach"}
print(t[1])           --> 32
print(t[2])           --> jeep
print(t["name"],t.name) --> cherry  cherry
print(p["name"],p.name) --> peach  peach
```

◆ **function 型**

函数是第一类值，意味着函数可以存储在变量中，可以作为函数的参数，也可以作为函数的返回值。这个特性给了语言很大的灵活性

◆ **userdata 型**

userdata 可以将 C 数据存放在 Lua 变量中，**userdata** 在 Lua 中除了赋值和相等比较外没有预定义的操作。**userdata** 用来描述应用程序或者使用 C 实现的库创建的新类型。

4.3.5 表达式

Lua 中的表达式包括数字、字符串、一元和二元操作符、函数调用。还可以是非传统的表构造。

◆ **算数表达式**

算数表达式是算数操作符及其操作对象所组成的表达式。Lua 中算数操作符的操作对象是实数。

Lua 中的算数操作符包括：

二元的算数操作符： + - * / ^(指数) %(取模)

一元的算数操作符： - (负号)

◆ **关系表达式**

由关系操作符及其操作对象所组成的表达式就是关系表达式。所有关系表达式的结果均为 **true** 或 **false**。

Lua 支持的关系操作符有：>、<、>=、<=、==、~=。

有几点需注意：

==和**~=**这两个操作符可以应用于两个任意类型的值。

如果两个值的类型不同，Lua 就认为他们不等。nil 值与其自身相等。例如：

```
print("0" == 0)           --> false
print(nil == false)       --> false
```

特别地，`tables`、`userdata`、`functions` 是通过引用进行比较的。也就是说，只有当他们引用同一个对象时，才视为相等。例如：

```
a = {x=1,y=2}
b = {x=1,y=2}
c = a
```

```
print(a==c)          --> true
print(b==c)          --> false
```

Lua 比较数字按传统的数字大小进行，比较字符串按字母的顺序进行，但是字母顺序依赖于本地环境。

例如：

```
print( 3 < 25)        --> true
print( "3" < "25")    --> false (alphabetical order!)
```

把字符串或数字用 `>`、`<`、`>=`、`<=` 符与不同类型的值比较时，会报错。

例如：

```
print( 3 < "25")      --> attempt to compare number with string
print( 5 >= false)    --> attempt to compare boolean with number
```

◆ 逻辑表达式

用逻辑运算符将关系表达式或逻辑量连接起来的有意义的式子称为逻辑表达式。

逻辑运算符有 3 个： `and` `or` `not`

逻辑运算符认为 `false` 和 `nil` 是假（`false`），其他为真，`0` 也是 `true`。

`and` 和 `or` 的运算结果不一定是 `true` 和 `false`，而是和它的两个操作数相关。

```
a and b              -- 如果 a 为 false，则返回 a，否则返回 b
a or b               -- 如果 a 为 true，则返回 a，否则返回 b
```

例如：

```
print(4 and 5)       --> 5
print(nil and 13)    --> nil
print(false and 13)  --> false
print(4 or 5)        --> 4
print(false or 5)    --> 5
print(true and false) --> false
print(true or false) --> true
```

◆ 字符串连接

字符串运算符是 `..`。

字符串连接，如果操作数为数字，**Lua** 将数字转成字符串。例如：

```
print ("hello" .. "everyone") --> hello everyone
print ( 2 .. " apples")      --> 2 apples
s = 2 .. 3
print(s , type(s))           --> 23 string
```

◆ 表的构造

表 (table) 构造器是用于创建和初始化表的表达式。其中最简单的构造器是空构造器 {}, 用于创建空表。

```
b = {x = 0, y = 1, "Monday", 109}
print(b[1],b[2],b.x,b.y)           --> Monday    109    0    1
```

4.3.6 函数

函数有两种用途:

1. 完成指定的任务, 这种情况下函数作为调用语句使用。例如:

```
print(8*9, "star")                --> 72    star
print(os.date())                   --> 07/30/14 15:17:36
                                   调用函数的时候, 如果参数列表为空, 必须使用()表明是函数调用
```

2. 计算并返回值, 这种情况下函数作为赋值语句的表达式使用。例如:

```
l = string.len("1234567")
print(l)                           --> 7
```

◆ **Lua 函数实参和形参的匹配与赋值语句类似, 多余部分被忽略, 缺少部分用 nil 补足**

```
function f(a, b)
    return a and b
end
c = f(3)  d = f(3,4)  e = f(3,4,5)
print(c, d, e)           --> nil  4  4
```

◆ **Lua 函数支持返回多个结果值**

例如: 有三个函数定义

```
function foo0 () end           -- returns no results
function foo1 () return 'a' end -- returns 1 result
function foo2 () return 'a','b' end -- returns 2 results
```

1. 当调用作为表达式最后一个参数或者仅有一个参数时, 根据变量个数函数尽可能多地返回多个值, 不足补 nil, 超出舍去。

```
x,y = foo2()                   -- x='a', y='b'
x = foo2()                     -- x='a', 'b' is discarded
x,y,z = 10,foo2()               -- x=10, y='a', z='b'
x,y = foo0()                   -- x=nil, y=nil
x,y = foo1()                   -- x='a', y=nil
x,y,z = foo2()                 -- x='a', y='b', z=nil
```

2. 其他情况下, 函数调用仅返回第一个值 (如果没有返回值为 nil)

```
x,y = foo2(), 20               -- x='a', y=20
x,y = foo0(), 20, 30           -- x=nil, y=20, 30 is discarded
print(foo2(), 1)               --> a 1
print(foo2() .. "x")           --> ax
```

```
a = {foo0(), foo2(), 4}           -- a[1] = nil, a[2] = 'a', a[3] = 4
```

◆ 函数的可变参数

Lua 函数可以接受可变数目的参数，在函数形参中用三点 (...) 表示函数有可变的参数。Lua 将函数的可变参数放在一个叫 **arg** 的表中，除了参数以外，**arg** 表中还有一个域 **n** 表示参数的个数。

在函数参数中，固定参数和可变参数可以一起声明，但是固定参数一定要在变长参数之前声明。

```
function test(arg1,arg2,...)
```

```
    ...
```

```
end
```

4.3.7 基本语法

◆ 赋值语句

Lua 中的赋值语句和其它编程语言基本相同，唯一的差别是 Lua 支持“多重赋值”。

例如：

```
local x,y = "test", 12           -- "test" 赋值给 x, 12 赋值给 y
```

◆ 局部变量和块

可以用 **local** 来定义局部变量，例如：

```
local a = "china"
```

local 是保留字，表示该变量是局部变量。和全局变量不同的是，局部变量的作用范围仅限于其所在的程序块。Lua 中的程序可以为控制结构的执行体、函数执行体或者是一个程序块。举例：

```
local x =12
```

```
if x >10 then
```

```
    local x = 0
```

```
    print("x=",x)
```

```
    -- 打印结果是： x= 0 ，而不是 12
```

```
end
```

◆ 控制语句

1) if 语句

if 语句有 3 种结构：

```
if condition then
```

```
    statements
```

```
end
```

```
if condition then
```

```
    statements
```

```
else
```

```
    statements
```

```
end
```

```
if condition1 then
```

```
    statements
```

```
elseif condition2 then
```

```
    statements
```

```

...                                -- 很多个 elseif
    statements
end

```

2) while 语句

while 语句语法如下:

```

while condition do
    statements
end

```

3) repeat 语句

repeat 语句语法如下:

```

repeat
    statements
until condition

```

4) break 和 continue

break 跳出内层循环, continue 不会跳出循环, 但会结束本次判断。

5. Lua 模块应用程序开发架构介绍

我们现在已经搭了一个架构, 将各个功能放在各个 module 中处理。运行是从 main 开始, 在 sys.run () 中循环运行。所以, sys 模块的 run () 是主架构, 这个主架构是用消息机制实现的。客户拿到这个架构可以根据需求增加或修改对应的消息处理程序。

5.1 main.lua 主程序

```

require "lcd"
require "keypad"
require "idle"

sys.init()

sys.run()

```

Main.lua 是主程序模块, 里面一般包括:

Require 程序 -- 用来加载使用到的 module (以.lua 为后缀。包括 lib 中的 module 和自己写的 module)

sys.init() -- 初始化, ATC 口, 物理 UART 口, GPIO, I2C 口的初始化在这里进行, 只执行一次

sys.run() -- 主循环程序, 是个无限循环, 放在 sys.lua 这个 module 里

5.2 主架构 run()详解

1) 该架构是用消息机制实现的；消息类型目前分为：

- 定时器 timeout 消息 (msg.id=rtos.MSG_TIMER=1)
- 串口消息(msg.id = rtos.MSG_UART_RXDATA=2)
又分为虚拟 AT 口 (msg.uart_id = uart.ATC) 和串口 (msg.uart_id = 串口号)
- 键盘消息 (msg.id=rtos.MSG_KEYPAD=3)
- 中断消息 (msg.id=rtos.MSG_INT=4)
- 电源管理消息 (msg.id=rtos.MSG_PMD=5)。
当电池在位状态、电池电压百分比、电池电压、充电器在位状态、充电状态任何一个发生-- 变化，就会上报该消息。

msg.present (电池在位状态, boolean 型, true 或 false)

msg.level (百分比 0-100, number 型)

msg.voltage (电池电压, number 型)

msg.charger (充电器在位状态, boolean 型, true 或 false)

msg.state (充电状态, number 型, 0-不在充电 1-充电中 2-充电停止)

2) 消息处理程序使用前需要用 regmsg 程序注册到 handlers 表中（除了定时器 timeout 消息处理程序 timerfnc）。这样代码改动影响面小，当业务流程有修改的时候，只需要修改具体的消息处理程序的内容即可。

3) run() 流程详细介绍。

```
function run()
```

```
    local msg
```

```
    while true do
```

```
        runqmsg()
```

```
        -- 接收消息。如果有消息上报，流程就往下走：如果没有消息上来，就一直阻塞这这里，
```

```
        -- 不往下走。本程序具体信息请参考《lua 扩展库.chm》
```

```
        msg = rtos.receive(rtos.INF_TIMEOUT)
```

```
        -- 定时器到时消息
```

```
        if msg.id == rtos.MSG_TIMER then
```

```
            timerfnc(msg.timer_id)
```

```
        -- 串口消息：虚拟 AT 口
```

```
        elseif msg.id == rtos.MSG_UART_RXDATA and msg.uart_id == uart.ATC then
```

```
            handlers.atc()
```

```
        else
```

```
            -- 串口消息
```

```
            if msg.id == rtos.MSG_UART_RXDATA then
```



```

-- 串口消息：物理 uart/host uart
-- 使用物理串口/host uart 前，需要用 reguart()注册端口号 msg.uart_id
if uartprocs[msg.uart_id] ~= nil then
    uartprocs[msg.uart_id]()
else
    --注册的其他消息处理程序
    handlers[msg.id](msg)
end
else
    --注册的其他消息处理程序
    handlers[msg.id](msg)
end
end
end
end
end

```

在 ril.lua 中我们已经用 sys.regmsg("atc",atcreader)把 ATC 口的处理程序注册好了。

如果需要键盘，则可以单独写个键盘应用模块 keypad.lua（名字自己可以随便取），在里面注册键盘处理程序。例如：sys.regmsg(rtos.MSG_KEYPAD,keymsg)。

如果需要电源管理，可以单独写个应用模块，比如取名为 power.lua，并在里面注册 rtos.MSG_PMD 消息处理程序，并定义相应的处理程序。

如果需要 GPIO 中断，也可以单独写个应用模块，并在里面注册 rtos.MSG_INT 消息的处理程序，并定义相应的处理程序。

客户在 Lua 编程的时候请不要修改库文件里面的内容，如果想增加或修改需求，在相应的处理程序上做修改即可。比如，如果想修改键盘处理程序，在 keymsg（）中修改即可。

5.3 lib 库中的各个模块

另外我们按照功能划分了各个模块（module），里面有该功能常用的程序和处理，也是以.lua 做后缀，当做 lib 库来使用。目前 lib 有以下几个模块：

sys.lua	系统模块，核心模块，里面有常用的系统功能
ril.lua	处理 AT 命令的核心模块
sms.lua	处理短信的命令的模块
link.lua	处理 IP 链路和数据收发命令的模块
cc.lua	处理呼叫控制命令的模块
audio.lua	处理音频相关的命令的模块
pm.lua	处理休眠和唤醒相关的命令的模块
pb.lua	处理电话本相关的命令的模块
net.lua	处理网络相关的命令的模块
update.lua	处理脚本升级功能的模块
common.lua	一些经常用到的通用程序

5.4 应用模块

模块脚本发布给客户的时候只有 lib 库，应用程序模块需要客户根据需求自己来写。

5.5 模块之间的调用关系

- 1) 需要使用这些模块的时候，只需要在调用模块中用 `require` 语句加载该模块即可。例如：在 `a.lua` 中想调用 `b.lua`，只需要在 `a.lua` 中 `require "b"`。这样，如果需要修改某处功能的时候，只需要修改相应的 `module`，从而保持代码的简洁和模块化。
- 2) 调用模块可以是 `main.lua`，也可以是各个分模块（包括 lib 中的模块和应用模块）。但是被调用模块不可以是 `main.lua`，只能是各个分模块。
- 3) 在被调模块加载后，可以在调用模块中使用被调模块的全局变量和全局程序，而不能使用被调模块的局部变量和局部程序。各个分模块内以 `local` 定义的变量和程序是局部程序；而未以 `local` 定义而直接赋值的变量是全局变量，直接用 `function` 定义的程序是全局程序。

例如，`idle.lua` 如果需要使用到 `keypad.lua` 中的一个全局程序 `prockey`，此时需要在 `idle.lua` 开头写上 `local keypad = require "keypad"`，并在需要使用的时候用 `keypad.prockey` 调用即可；如果想调用 `keypad.lua` 中的全局变量 `a`，调用方法是 `keypad.a`。

- 4) 不可以循环 `require`。
比如 A 模块 `require B` 模块，B 模块 `require C` 模块，C 模块不可以再 `require A` 模块。
- 5) 客户自己也能根据业务需求自己编制应用 `module`。但是请尽量不要改动 lib 库中的 `module`。

6. 程序注册

在使用相关的程序做各种处理的时候，需要先将程序注册下，否则无法使用。这样做的好处是可以将同类型程序集中放置一个表中，方便管理和修改。除了上面提及的 `regmsg()` 外，还有 `regurc()`，`regapp()`，`regnotify()`，`regrsp()`，`reguart()`。

6.1 regmsg()用来注册相应的消息处理程序

`regmsg()` 这个程序放在 `sys.lua` 这个库文件中。我司提供的 lua 主架构 `sys.run()` 目前能够处理 5 种类型的消息，除定时器消息外(定时器函数直接用 `sys.timer_start` 来启动)，其余四种均需要用 `regmsg` 或 `reguart` 注册相应的消息处理函数。

- 串口消息
又分为虚拟 AT 口 (`msg.uart_id = uart.ATC`) 和串口 (`msg.uart_id = 串口号`) 消息。
其中虚拟 AT 口的消息已经缺省注册了，不需要用户再注册；
串口消息处理函数使用前需要用 `reguart` 来注册。

- 键盘消息 (msg.id=rtos.MSG_KEYPAD=3)
需要用 regmsg()来注册处理函数
- 中断消息 (msg.id=rtos.MSG_INT=4)
需要用 regmsg()来注册处理函数
- 电源管理消息 (msg.id=rtos.MSG_PMD=5)
需要用 regmsg()来注册处理函数

```
function regmsg(id,handler)
    handlers[id] = handler
end
```

regmsg 处理流程见下面的图:

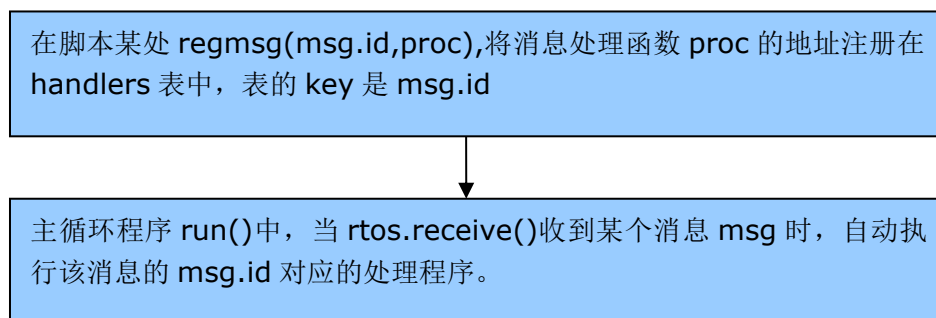


图 regmsg() 流程图

使用方法举例:

例 1:在应用模块中加入键盘处理程序 **keymsg (msg)**:

第一步: 编制键盘消息处理程序

```
local function keymsg(msg)
    if msg.pressed then
        键盘按下时的处理程序
    else
        键盘弹起时的处理程序
    end
end
```

第二步: 注册该处理程序

```
sys.regmsg(rtos.MSG_KEYPAD,keymsg)
```

这样，当 `rtos.receive()` 收到键盘消息 `msg` (`msg.id=rtos.MSG_KEYPAD=3`) 时，通过 `sys.run()` 的分发，自动进入 `keymsg(msg)` 这个 function 中进行处理。

例 2: 在应用模块中加入 **GPIO** 中断处理程序

第一步：编制 GPIO 中断处理程序

假设：

gsensor 连接的是 GPIO5

机械式震动传感器连接的是 GPIO3

GPS 连接的是 GPO1

```
IO.gsensor = pio.P0_5
IO.shake = pio.P0_3
IO.gps = pio.P1_1
local function gpio_int (msg)
    --产生下降沿脉冲中断
    if msg.int_id == cpu.INT_GPIO_NEGEDGE then
        --如果产生中断的 GPIO 是 gsensor 引脚
        if msg.int_resnum == IO.gsensor then
            --如果产生中断的 GPIO 是机械式振动传感器引脚

            elseif msg.int_resnum == IO.shake then

                --如果产生中断的 GPO 是 GPS 引脚
                elseif msg.int_resnum == IO.gps then

                    end
            --产生上升沿脉冲中断
        elseif id == cpu.INT_GPIO_POSEDGE then

            end
    end
end
```

第二步：注册该处理程序

```
sys.regmsg(rtos.MSG_INT,gpio_int)
```

例 3：在应用模块中加入 **PMD**（电源管理）消息处理程序

第一步：编制电源管理消息处理程序

```
local function BatManage(msg)
    print("msg.voltage,msg.charger,msg.state = ",msg.voltage,msg.charger,msg.state)
    DealCharger(msg)
    DealVolt(msg)
end
```

第二步：注册该处理程序

sys.regmsg(rtos.MSG_PMD,BatManage)

6.2 regapp()用来注册应用程序

对 AT 命令的查询结果或某些事件的处理结果，会以 **dispatch**（消息 id, 参数 1,参数 2,...）的形式来通知，如果想在 **app** 中对该消息 id 及对应的参数（参数 1,参数 2,...）进行进一步处理，需要事先以 **regapp** 方式注册该 **app** 程序，注册后，每次 **dispatch** 相应的消息 id，就自动会进入相应的 **app** 程序对 **dispatch** 出来的消息 id 和参数进行处理。

***注：dispatch 接口会在第 7 章详细介绍**

regapp 的详细流程详见下图：

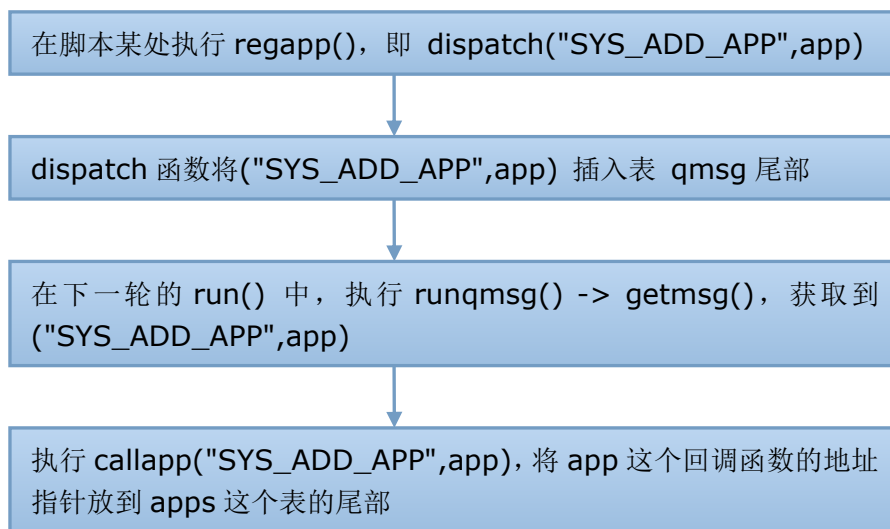


图 regapp() 流程图

regapp 有 2 种注册方式：

1) 以表的形式来注册

```
local table = { 消息 id1 = app1, 消息 id2 = app2, ..... , 消息 idn = appn }  
sys.regapp (table)
```

这种以表的方式注册 **app** 程序，一次可以注册多个 **app**。

使用方法举例：

例：注册多个 dispatch 消息处理程序

第一步：填写注册表

在本例中：

长按键消息（MMI_KEYPAD_LPRESS）和短按键消息（MMI_KEYPAD_SPRESS）由客户编写用程序 **dispatch** 出来；

通话相关的消息（CALL_CONNECTED，CALL_DISCONNECTED，CALL_INCOMING）由库文

件 cc.lua dispatch 出来

```
local idleapp = {  
    MMI_KEYPAD_LPRESS = LongPresskey,    --长按键处理  
    MMI_KEYPAD_SPRESS = ShortPresskey,   --短按键处理  
    CALL_CONNECTED = connect,            --通话接通  
    CALL_DISCONNECTED = disconnect,       --通话挂断  
    CALL_INCOMING = incall                --来电话  
}
```

第二步：注册该表

```
sys.regapp(idleapp)
```

2) 以 app 的形式注册

```
sys.regapp (app,消息 id)
```

以 app 形式注册的程序，一次只可以注册一个 app，但是消息 id 可以不止一个。

使用方法举例：

例：当网络注册状态改变时，闪灯随之改变

第一步：编制闪灯变化程序

```
local function light(id,data)  
    -- 当 GSM 网络注册状态发生变化时闪灯发生变化  
    if id == "NET_STATE_CHANGED" then  
        if data == "REGISTERED" then  
            GPIO_ontime(SLOW,ioChgNet)    --蓝灯慢闪  
        else  
            GPIO_ontime(SLOW,ioChgingNoNet) --红灯慢闪  
        end  
        -- 当充电状态发生变化时的闪灯也相应变化  
    elseif id == "CHARGE_STATE_CHANGED" then  
        if data == "CHARGING" then  
            GPIO_ontime(FAST,ioChgingNoNet) --红灯快闪  
        else  
            GPIO_on(ioChgNet)                --蓝灯常亮  
        end  
    end  
end  
end
```

第二步：注册 light 程序

```
sys.regapp(light,"NET_STATE_CHANGED"," CHARGE_STATE_CHANGED")
```

6.3 regurc()用来注册某些 URC 相应的处理程序

URC=unsolicited result code ，即非请求式上报命令，也就是主动上报的命令。

如果用户需要在应用模块中自己处理感兴趣的 URC，需要用 regurc() 程序来注册 URC 处理程序来达到目的。regurc() 程序放在 ril.lua 中。

```
function regurc(prefix,handler)
    urctable[prefix] = handler
end
```

regurc 处理流程见下面的图：

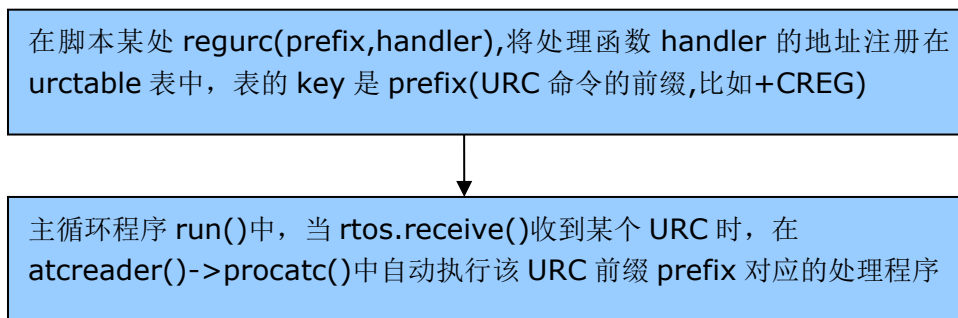


图 regurc() 流程图

使用方法举例：

例：自己处理 URC

第一步：根据需求，自己编制 URC 的处理程序

```
local function IdleUrc(data,prefix)
    if prefix == "+CPIN" then
        local p = smatch(data,"NOT%s*INSERTED",string.len(prefix)+1)
        if p then
            pbapp.SetSim(false)
        end
        local p = smatch(data,"READY",string.len(prefix)+1)
        if p then
            pbapp.SetSim(true)
        end
    end
end
```

第二步：注册该程序

```
ril.regurc("+CPIN",IdleUrc)
```

6.4 regrp()用来注册 AT 命令处理程序

当用户发送 AT 命令时，一般是需要这些命令的返回结果的，AT 命令返回结果的处理程序 app 是通过 regrp()函数注册在 rsptable 表中的。这样当发送 AT 命令时，会自动执行该处理程序处理 AT 命令返回结果。

function regrp(head,fnc,typ) 放在 ril.lua 这个库文件中。
head 是 AT 命令的头，fnc 是处理函数，typ 是该命令的类型（cmd type: 0:no result 1:number 2:sline 3:mline 4:string 10:spec）。

例如：AT+CHFA?这个获取音频通道命令的 head 是+CHFA?，fnc 是 AT 命令返回结果处理程序，由客户自己来编写，typ 是 2。

对 AT 命令的返回结果的处理，一般有两种情况：

1) 一些常用的 AT 命令的处理程序，库文件已经缺省用 regrp 注册了。此时处理程序会把 AT 命令结果 dispatch 出来，后续再用 regapp 再做进一步处理（适用于异步处理或前后有因果关系的情况）

2) 如果 AT 命令在 lib 库文件中没有被 regrp 注册过，则需要用 regrp()注册相应的处理程序。此时用户可以在这些处理程序中直接处理 AT 命令返回结果（适用于同步处理），也可以用 1) 的异步处理方式。

regrp 处理流程见下面的图：

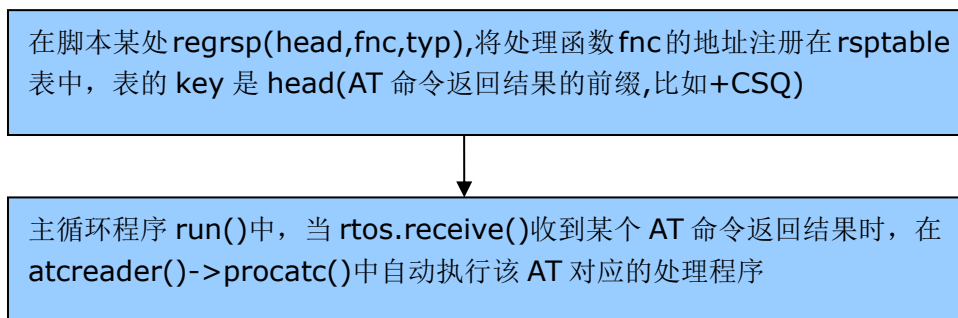


图 regrp() 流程图

使用方法举例：

例 1：使用 AT+CPAS?命令获取手机状态

第一步：编制 rsp 程序来处理 AT+CPAS?命令的返回结果

```
local sta
```

```
local function rsp(cmd,succes,response,intermediate)
```



```

    if cmd == "AT+CPAS?" then
        sta = string.match(intermediate,"+CPAS:%s*(%d)")
    end
end

```

第二步：注册该处理程序

```
ril.regrsp("+CPAS?",rsp,2)
```

6.5 reguart()用来注册 uart 口的数据处理程序

串口(包括物理串口和 debug uart 口)使用之前需要先用 reguart () 程序来注册处理程序。

reguart() 放在 sys.lua 这个库文件中。

```

function reguart(id,fnc)
    uartprocs[id] = fnc
end

```

现在的端口 id 分配如下：

物理端口 1: id = 1

物理端口 2: id = 2

debug 口: id = 3

reguart 处理流程见下面的图：

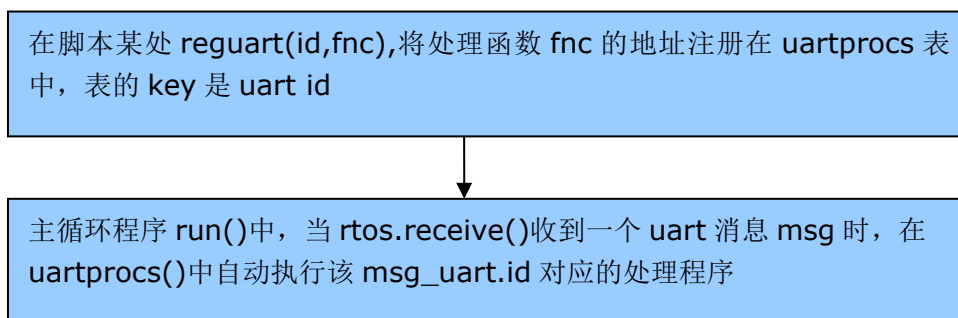


图 reguart() 流程图

举例说明：如何使用 debug uart 口与上位机通讯

第一步：

第二步：设置 debug uart 口的通讯特性

```
uart.setup(3,921600,8,uart.PAR_NONE,uart.STOP_1,2) （debug uart 口波特率必须是 921600 ， mode=2，使用 ID=0xA2 数据透传）
```

第二步：编写 debug uart 的输入输出处理程序

```
local function huartreader()
    local s

    while true do
        s = uart.read(3,"*l",0)

        if string.len(s) ~= 0 then
            cmddealer(s)
        else
            break
        end
    end
end
```

第三步：注册该处理程序

```
sys.reguart(3,huartreader)
```

7. 各模块详细介绍

全局函数是可以被外部模块调用的，而局部函数不可以被别的模块调用。所以本章将对各个分模块（.lib 文件）中的全局函数接口做详细介绍。

全局函数的调用方法，前面已经讲过：先 **require** 相关的 lib，然后以库名.函数名的方法进行调用。

7.1 sys 模块详解

sys.lua 这个 module 中除了主框架程序 run（）外，还有些常用的程序：

● 开启定时器

程序名	timer_start (fnc,ms,...)	
功能	开启一个定时器	
输入参数	fnc	function 型。定时器时间 ms 到时后，自动调用 fnc 处理程序
	ms	number 型。该参数设置定时器的时间，以 ms（毫秒）为单位
	...	不定参数，fnc 的参数
返回值	number 型。本次开启定时器分配的 uniquetid。uniquetid 具有独一性，每次调用开启定时器程序 timer_start 的时候分配。	

● 关闭定时器

程序名	timer_stop (val,...)	
功能	结束一个定时器	
输入参数	val	可以为 timer_id，对应上面的 uniquetid，number 型。也可以为 function 类型，对应上面的 fnc
	...	不定参数，fnc 的参数
返回值	无	
举例	例 1：开启定时器，并用 uniquetid 的方式停止定时器 local sms = require "sms" local function sendsms (num,data) sms.send(num,data) end --3 秒后发送短信，收件人号码是 10086，内容是"12345" local tid = timer_start(sendsms,3000,"10086","123456") 如果定时器到时前不想发送了，可以停止定时器 timer_stop(tid,"10086","123456")	
	例 2：开启定时器，并用 function 的方式停止定时器 还是上面的例子 timer_start(sendsms,3000, "10086","123456") timer_stop(sendsms, "10086","123456")	

- 判断定时器是否 active 状态

程序名	timer_is_active (val,...)	
功能	判断某个定时器是否激活状态	
输入参数	val	可以为 timer_id, 对应上面的 uniquetid, number 型。也可以为 function 类型, 对应上面的 fnc
	...	不定参数, fnc 的参数
返回值	无	

- 系统开机

程序名	poweron()	
功能	开机	
输入参数	无	
返回值	无	

- 系统初始化

程序名	init(mode)	
功能	系统初始化。ATC 口(即虚拟 AT 命令口)初始化	
输入参数	mode	整数型。目前只支持 mode=1 (这种模式下, 充电的时候不会自动开机)
返回值	无	
举例	<p>例 1: 设备充电不开机, 按键的时候才开机</p> <p>在 main.lua 中,</p> <pre> sys.init(1) if rtos.poweron_reason() ~= rtos.POWERON_CHARGER then sys.poweron() end sys.run() </pre>	

- 消息处理程序注册

程序名	regmsg (id,handler)	
功能	将某类消息 str 的处理程序 handler 注册到一个表 handlers 中	
输入参数	id	消息类型, 整数型
	handler	该类消息的处理程序
返回值	无	
举例	详细请参见 regmsg()用来注册相应的消息处理程序	

- app 程序注册

程序名	regapp (...)	
功能	目前支持 2 种注册方式： 1) 讲一个消息和对应的应用处理程序注册到 apps 表中 2) 以表的形式注册到 apps 表中。表中的每个元素 key 是消息， value 是对应的消息处理程序	
输入参数	...	不定参数
返回值	表或应用处理程序	
举例	详细请参见 regapp()用来注册应用程序	

● app 程序解注册

程序名	deregapp (id)	
功能	将 id 这个子程序或表从 apps 表中删除	
输入参数	id	需要解注册的应用处理程序或表
返回值	无	

● 物理串口处理程序注册

程序名	reguart(id,fnc)	
功能	将处理物理串口的程序 fnc 注册起来	
输入参数	id	整数型。物理串口号
	fnc	function 型。处理串口数据的程序
返回值	无	
举例	详细请参见 reguart()用来注册 uart 口的数据处理程序	

● 消息分发

程序名	dispatch(...)	
功能	消息分发	
输入参数	...	不定参数，但其中第一个参数是消息 id
返回值	无	
举例	详细请参见 消息分发函数 dispatch 详细介绍	

● 主循环程序

程序名	run()	
功能	主架构程序。一般情况下，这个程序是必须要有，而且必须要写在 main.lua 里。	
输入参数	无	
返回值	无	
举例	详细请参见 主架构 run()详解	

7.2 ril 模块详解

- 发送 AT 命令

程序名	request(cmd,arg,onrsp)	
功能	发送 AT 命令。其实是将发送的 AT 命令插入到 AT 命令队列中。队列中的 AT 命令会按 FIFO 的顺序及时执行	
输入参数	cmd	需要发送的 AT 命令，字符串型
	arg	跟该命令相关的参数，字符串型
	onrsp	对该命令的回复结果进行处理的回调函数，function 型。用 regrsp 也可以注册 AT 命令的回复结果进行处理的回调函数。onrsp 的优先级高于 regrsp 注册的回调函数，前者存在时，后者不起作用。onrsp 用于对回复结果做特殊处理。
返回值	无	
举例	<p>发送数据的函数需要调用到 ril.lua 中的 request 函数：</p> <pre>local ril = require"ril" local req = ril.request function send(id,data) req(string.format("AT+CIPSEND=%d,%d",id,string.len(data)),data) return true end</pre>	

- 注册 urc 处理程序

程序名	regurc(prefix,handler)	
功能	将 URC 处理程序注册到 urctable 表中	
输入参数	prefix	URC 命令的前缀，比如"+CREG"，字符串型
	handler	URC 处理程序，function 型
返回值	无	
举例	详细请参见 regurc()用来注册某些 URC 相应的处理程序	

- 解注册 urc 处理程序

程序名	deregurc(prefix)	
功能	将 URC 处理程序从 urctable 表解注册	
输入参数	prefix	URC 命令的前缀，比如"+CREG"，字符串型
返回值	无	

- 注册 rsp 程序
(rsp 是对 AT 命令响应的处理程序)

程序名	regrsp(head,fnc,typ)	
功能	注册 rsp 程序	
输入参数	head	以 AT 命令的关键字作为注册的 rsptable 表的 key，字符串型
	fnc	对 AT 命令返回结果的处理程序
	typ	head 的类型，整数型 1~10 0:no reuslt 1:number 2:sline 3:mline 4:string 10:spec
返回值	无	
举例	详细请参见 regrsp() 用来注册 AT 命令处理程序	

7.3 link 模块详解

- 设置 APN

程序名	setapn(a)	
功能	设置 APN	
输入参数	a	APN，字符串型
返回值	无	

- 建立一个使用 IP 协议的连接，并获取一个 TCP/UDP 链接号

程序名	open(notify,recv)	
功能	获取一个 TCP/UDP 连接的 id，并且如果还没建立 PPP 连接，则在终端与 GPRS 网络之间建立一个 IP 连接	
输入参数	notify	对此连接的状态通知和处理的程序，function 型
	recv	对此连接收到的数据进行处理程序，function 型
返回值	返回建立的这个连接的 id，整数型	
举例	<p>建立一条 IP 链接：</p> <p>在用户的应用程序中（比如取名 dataapp.lua）编写代码如下：</p> <pre>local link = require"link" --用户编写的 notify 程序参数数量和类型须与 link.lua 库文件中的 notify 函数保持一致 local function notify1 (id,evt,state) statements end --用户编写的 recv 程序参数数量和类型须与 link.lua 库文件中的 recv 函数保持一致 local function recv1(id,data) statements end --建立一个 IP 连接，并获取一个 TCP/UDP 链接号 local id = link.open(notify1,recv1)</pre>	

	--如果此前没 open 过，则打印 id=0 print("id=",id)
--	--

● 打开一个 TCP/UDP 连接

程序名	connect(id,protocol,address,port)	
功能	在终端与服务器之间建立一个 TCP 或 UDP 连接	
输入参数	id	TCP/UDP 连接的 id 号，通过 open 函数获取，整数型
	protocol	协议类型，TCP 还是 UDP，字符串型
	address	服务器地址，字符串型
	port	服务器端口，字符串型
返回值	boolean 型，true 或 false	
举例	<p>在一条 IP 链接的基础上建立 TCP 连接：</p> <p>在上个例子中已经获取到一个 IP 链接，并且通过 open 函数分配到了一个 TCP/UDP 连接号 id，在此基础上建立 TCP/UDP socket 连接即可，方法如下：</p> <p>在用户的应用程序中（比如取名 dataapp.lua）编写代码如下：</p> <pre>local svraddr,svrport = "180.181.25.6", "1238" link.connect(id,"TCP",svraddr,svrport)</pre> <p>注:IP 连接只需建立一次，在同一条 IP 链接上可以建立 8 条 TCP/UDP 连接（id=0~7）</p> <p>比如还想在此 IP 连接之上再建立一条 UDP 链接：</p> <pre>local svraddr,svrport = "180.181.25.6", "2299" id = link.open(notify2,recv2) link.connect(id,"UDP",svraddr,svrport)</pre>	

● 在某 TCP/UDP 连接上发送数据

程序名	send(id,data)	
功能	在某连接上发送数据	
输入参数	id	TCP/UDP 连接的 id 号，整数型
	data	数据，字符串型
返回值	boolean 型，返回 true 或 false	
举例	<p>在连接 0 上发送数据：</p> <pre>link.send(0,"1234567890")</pre>	

● 关闭某 TCP/UDP 连接

程序名	disconnect(id)	
功能	关闭某 TCP/UDP 连接	
输入参数	id	TCP/UDP 连接的 id 号，整数型

返回值	boolean 型，返回 true 或 false
-----	---------------------------

- 关闭某 TCP/UDP 连接，并清除 id 号

程序名	close(id)	
功能	关闭某 TCP/UDP 连接，并清除 id 号	
输入参数	id	TCP/UDP 连接的 id 号，整数型
返回值	boolean 型，返回 true 或 false	
说明	<p>disconnect(id) 和 close(id)的相同和不同之处：</p> <p>两者都是断掉一个 TCP/UDP 连接，不同之处在于：disconnect(id)只断掉连接，id 仍保留，下次需要再连接的时候直接用此 id 调用 connect 函数，不用先 open；而 close(id)不仅断掉连接，还将分配的 id 清掉，此 id 已不存在。也就是再连接的话，需要先调用 open 函数获取 id，再 connect</p>	

- 设置连接服务器的定时器时间

程序名	setconnectnoretrestart(flag,interval)	
功能	<p>开始连接服务器时，会打开一个连接超时定时器，定时器 timeout 以后，会重启模块。这个定时器是为了防止连接服务器时一直无回应导致程序跑死。</p> <p>本程序是设置定时器的开始标志，以及时长。</p>	
输入参数	flag	定时器开启标志
	interval	定时器时长
返回值	无	
举例	<p>设置定时器的实例：</p> <pre>local link = require"link" -- 定时器时长为 100000ms (100s) link.setconnectnoretrestart(true, 100000)</pre>	

- 获取某连接的状态

程序名	getstate(id)	
功能	获取某 TCP/UDP 连接的状态	
输入参数	id	TCP/UDP 连接的 id 号，整数型
返回值	无	
举例	<p>获取连接 1 的状态</p> <pre>local link = require"link" link.getstate(1)</pre>	

- 重新连接 IP

程序名	reset()
功能	重新建立 IP 连接
输入参数	无
返回值	无

- 关闭 IP 连接

程序名	shut()
功能	关闭当前 IP 连接
输入参数	无
返回值	无

7.4 sms 模块详解

我司提供的 sms.lua 模块，短信缺省格式为 TEXT 型，编码格式是 UCS2，使用默认的存储方式（一般默认为 SIM）。

- 获取短信初始化状态

程序名	getsmstate()
功能	获取短信初始化状态
输入参数	无
返回值	boolean 型 true: 初始化完成; false: 初始化未完成

- 发送短信

程序名	send(num,data)	
功能	发送短信。发送	
输入参数	num	发送短信的目的号码，字符串型
	data	短信内容，字符串型
返回值	返回 boolean 型，true （已发送）或 false （未发送）	

- 读短信

程序名	read(pos)
功能	读短信，读的内容以如下形式 dispatch 出来： dispatch("SMS_READ_CNF",success,num,data,pos,t,name) success: boolean 型，true （成功）或 false （失败） num:

	发送者号码，字符串型 data: 短信内容，UCS2 编码，字符串型 pos: 短信在存储空间的位置，字符串型 t: 短信中心时间戳，字符串型。格式如："13/01/06,10:11:47+32"（+32 表示时区，东八区） name: 发送者姓名。字符串型。如果为陌生号码，则姓名为空字符串""	
输入参数	pos	短信在 ME 或 SM 存储器中的位置，字符串型
返回值	返回 boolean 型，true（已读取） 或 false（未读取）	

- 删除短信

程序名	delete(pos)	
功能	删除短信。删除这个动作的执行结果以如下方式 dispatch 出来： dispatch("SMS_DELETE_CNF",success) success: boolean 型，true（成功）或 false（失败）	
输入参数	pos	短信在 ME 或 SM 存储器中的位置，字符串型
返回值	返回 boolean 型，true（已做删除动作） 或 false（未做删除动作）	

7.5 cc 模块详解

- 拨打语音电话

程序名	dial(number)	
功能	拨打语音电话	
输入参数	number	拨打的电话号码，字符串型
返回值	返回 boolean 型，true（已经拨出电话）或 false（未能拨出电话）	

- 挂断电话

程序名	hangup()	
功能	挂断所有当前语音电话	
输入参数	无	
返回值	无	

- 接听电话

程序名	accept()	
-----	-----------------	--

功能	接听当前打入的语音电话
输入参数	无
返回值	无

● 判断是否是紧急呼叫号码

程序名	isemergencynum(num)	
功能	判断是否是紧急呼叫号码	
输入参数	num	号码，字符串型
返回值	返回 boolean 型，true 或 false	

7.6 net 模块详解

- 获取当前网络注册状态

程序名	getstate()
功能	获取当前 GSM 网络注册状态
输入参数	无
返回值	网络注册状态，字符串型： INIT(刚开机的状态) UNREGISTER（没有注册网络） REGISTERED（已经注册了网络）

- 获取当前位置区号 LAC

程序名	getlac()
功能	获取当前 GSM 网络当前的位置区号
输入参数	无
返回值	位置区 LAC，字符串型

- 获取当前小区号 CI

程序名	getci()
功能	获取当前 GSM 网络当前的小区号
输入参数	无
返回值	小区号 CI，字符串型

- 获取当前 rssi 值

程序名	getrssi()
功能	获取当前 GSM 网络接收信号强度
输入参数	无
返回值	rssi 值，0~31

- 周期性发送 AT+CREG? 查询网络注册状态

程序名	startquerytimer()
功能	每隔 2 秒查询一次是否注册上网络，如果注册上网络，就停止了查询； 如果没有插 SIM 卡，不会查询； 如果插卡了，但是一直没有注册上网络，会一直查询
输入参数	无
返回值	无

- 周期性发送 AT+CSQ 查询信号强度

程序名	startcsqtimer()
功能	每隔 csqqueryperiod 秒查询一次信号强度
输入参数	无
返回值	无

- 设置信号强度查询周期

程序名	setcsqueryperiod(period)
功能	设置信号强度查询周期，并开始周期性信号查询
输入参数	period 信号强度查询周期，整数型，单位为毫秒
返回值	无

- 查询一次邻小区信息

程序名	cengquery()
功能	查询一次邻小区信息。查询结果 dispatch 出来。
输入参数	无
返回值	无

- 周期性发送 AT+CENG? 查询邻小区信息

程序名	startcengtimer()
功能	通过 AT+CENG?周期性查询邻小区信息（LAC,CI,rssi）
输入参数	无
返回值	无

- 设置邻小区信息查询周期

程序名	setcsqueryperiod(period)
功能	设置邻小区信息查询周期。 如果 period≤0，则停止查询； 如果 period>0，则开始周期性查询
输入参数	period 邻小区信息查询周期，整数型，单位为毫秒
返回值	无

7.7 pm 模块详解

- 唤醒模块

程序名	wake(tag)	
功能	将模块唤醒	
输入参数	tag	module 名，字符串型。tag 可以是 lib 中的库文件，比如 cc，net 等，也可以是用户自己编写的应用 module。引入 tag 的目的是只要存在任何一个模块唤醒，则整个模块都不睡眠。
返回值	无	

- 将模块休眠

程序名	sleep(tag)	
功能	将模块休眠	
输入参数	tag	同 wake(tag)
返回值	无	

7.8 common 模块详解

- 将 UCS2 码转成 ASCII 码

程序名	ucs2toascii(inum)	
功能	将 UCS2 码字符串转成 ASCII 码字符串	
输入参数	inum	待转换字符串
返回值	转换后的字符串	

- 将号码（ASCII 字符串）转换成 UCS2 码字符串

程序名	nstrToUcs2Hex(inum)	
功能	将号码（ASCII 字符串）转换成 UCS2 码字符串	
输入参数	inum	号码，字符串型
返回值	字符串型	
举例	"+1234" -> "002B0031003200330034"	

- 十六进制串转成可见字符串

程序名	binstohexs(bins)	
功能	十六进制串转成可见字符串	
输入参数	bins	十六进制串
返回值	可见的字符串	
举例	(122B5699) hex -> "122B5699"	

- 十六进制可见字符串转换为 ASCII 字符串

程序名	hexstobins(hexs)
-----	------------------

功能	十六进制可见字符串转换为 ASCII 字符串	
输入参数	hexs	十六进制可见字符串，字符串型
返回值	ASCII 字符串，字符串型	
举例	"2B3132" -> "+12"	

- UCS2 编码的字符串转换为 GB2312 编码的字符串

程序名	ucs2togb2312(ucs2s)	
功能	UCS2 编码的字符串转换为 GB2312 编码的字符串	
输入参数	ucs2s:	UCS2 编码的字符串，字符串型
返回值	GB2312 编码的字符串，字符串型	

- GB2312 编码的字符串转换为 UCS2 编码的字符串

程序名	gb2312toucs2(gb2312s)	
功能	GB2312 编码的字符串转换为 UCS2 编码的字符串	
输入参数	gb2312s	GB2312 编码的字符串，字符串型
返回值	UCS2 编码的字符串，字符串型	

- UCS2BE(大端在前的 UCS2)编码的字符串转换为 GB2312 编码的字符串

程序名	ucs2betogb2312(ucs2s)	
功能	UCS2BE 编码的字符串转换为 GB2312 编码的字符串	
输入参数	ucs2s	UCS2BE 编码的字符串，字符串型
返回值	GB2312 编码的字符串，字符串型	

- GB2312 编码的字符串转换为 UCS2BE(大端在前的 UCS2)编码的字符串

程序名	gb2312toucs2be(gb2312s)	
功能	GB2312 编码的字符串转换为 UCS2BE(大端在前的 UCS2)编码的字符串	
输入参数	b2312s	GB2312 编码的字符串，字符串型
返回值	UCS2BE 编码的字符串，字符串型	

7.9 pb 模块详解

- 查找电话本的一笔记录

程序名	find(name)	
功能	查找电话本的一笔记录	
输入参数	name	姓名，字符串型
返回值	boolean 型 true: 表示查找这个动作已经发起	

	false: 表示查找这个动作未成功发起
--	-----------------------------

- 读取电话本的一笔记录

程序名	read(index)	
功能	读取电话本的一笔记录	
输入参数	index	该记录在电话本的位置，整数型
返回值	boolean 型 true: 表示读取这个动作已经发起 false: 表示读取这个动作未成功发起	

- 写电话本的一笔记录

程序名	writeitem(index,name,num)	
功能	写入电话本的一笔记录	
输入参数	index	该记录在电话本的位置，整数型
	name	姓名，字符串型
	num	号码，字符串型
返回值	boolean 型 true: 表示写入一笔电话本记录这个动作已经发起 false: 表示写入这个动作未成功发起	

- 删除电话本的一笔记录

程序名	deleteitem(index)	
功能	删除电话本的一笔记录	
输入参数	index	该记录在电话本的位置，整数型
返回值	boolean 型 true: 表示删除一笔电话本记录这个动作已经发起 false: 表示删除这个动作未成功发起	

7.10 audio 模块详解

- 打开 DTMF 检测开关

程序名	dtmfdetect(enable,sens)	
功能	打开 DTMF 检测开关 如果打开开关的话，就可以对接收到的 DTMF 音进行解析	
输入参数	enable	DTMF 检测开关打开还是关闭，取值为 true 或 false 。 boolean 型
	sens	灵敏度，取值为 1 或 2（默认），1 的灵敏度比 2 高。 整数型
返回值	无	

- 通话中向对端播放 DTMF 音或单频音

程序名	senddtmf(str,playtime,intvl)	
功能	该命令可以在通话的时候，向对方播放DTMF音或单频音	
输入参数	str	字符串型。取值范围为：0~9,A,B,C,D,*,# 注： 1. 当发送单频音（固定为2500hz）时，取值为“WWWW”（一个W代表一个单频音,想发几个单频音就要写几个） 例如：“WWWWWW” 用来发送6个单频音 2. 不要用任何符号间隔，且必须为大写必须用大写 例如：“0222111ABCD*#”
	playtime	单个音的播放时间，取值为1~5000，单位为ms。整数型
	intvl	相邻两个音之间的时间间隔，取值为1~5000，单位为ms。整数型
返回值	Boolean 型 true: 表示这个动作已经发起 false: 表示这个动作未成功发起	

- 播放 TTS

程序名	playtts(text,path)	
功能	本地播放或通话时向对端播放 TTS	
输入参数	text	播放的内容，字符串型，UNICODE 码(但是需要将小端放前)。例如： 播放"123" 需要写成"310032003300" "net": 是向对端播放 "speaker": 是本地播放
	path	播放的路径，字符串型。
返回值	无	

- 停止 TTS 播放

程序名	stoptts()	
功能	停止 TTS 播放	
输入参数	无	
返回值	无	

- 通话中播放 AMR 声音文件到对端

程序名	transvoice(data,loop)	
功能	通话中播放声音文件到对端（文件格式必须是 12.2K AMR）	
输入参数	data	声音文件，格式为 12.2K AMR，二进制字符串
	loop	是否循环播放，boolean 型
返回值	boolean 型 true: 表示这个动作已经发起 false: 表示这个动作未成功发起	

- 播放声音文件

程序名	play(name)	
功能	播放声音文件	
输入参数	name	声音文件名称。 声音文件由 luaDB 下载工具缺省下载到模块的/ldata 目录中，调用的时候形式如： "/ldata/filename" filename 是以.amr 或 .mp3 或 .midi 为后缀的音频文件的名字
返回值	无	

- 停止播放声音文件

程序名	stop()	
功能	停止播放当前正在播放的声音文件	
输入参数	无	
返回值	无	

- 设置听筒或喇叭的音量

程序名	setspeakervol(vol)	
功能	设置听筒或喇叭的音量	
输入参数	vol	音量，取值为 0~100，整数型
返回值	无	

- 查询听筒或喇叭的音量

程序名	getspeakervol()	
功能	查询听筒或喇叭的音量	
输入参数	无	
返回值	音量，取值为 0~100，整数型	

- 设置音频通道号

程序名	setaudiochannel(channel)	
功能	设置音频通道	

输入参数	channel	音频通道，整数型，取值 0~6	
		取值	对取值的说明
		0	通道 0（手柄上行+手柄下行）
		1	通道 1（免提上行+耳机下行）
		2	通道 2（免提上行+免提下行）
		3	通道 3（免提上行+手柄下行）
		4	通道 4（手柄上行+免提下行）
		5	通道 5（手柄上行+耳机下行）
返回值	无		

- 查询音频通道号

程序名	getaudiochannel()		
功能	查询当前音频通道号		
输入参数	无		
返回值	音频通道，整数型，取值 0~6		

- 设置 mic 音量

程序名	setmicrophonegain(mode,vol)	
功能	设置 mic 音量	
输入参数	mode	音频通道，整数型，定义同 setaudiochannel(channel)中 channel
	vol	mic 音量，整数型，取值为 0~15
返回值	无	

- 查询 mic 音量

程序名	getmicrophonegain()		
功能	查询 mic 音量		
输入参数	无		
返回值	mic 音量，整数型，取值为 0~15		

- 开始录音

程序名	beginrecord(id,duration)		
功能	开始录音		
输入参数	id	录音开始后，产生一个录音文件放在模块 NV 中，以 id 来标记 id 与文件系统中录音文件的对应关系：format("/RecDir/rec%03d",id)	
	duration	录音文件的时长。1~50000 毫秒	
返回值	ture	布尔变量	

- 停止录音

程序名	stoprecord(id,duration)	
功能	停止录音	
输入参数	id	定义同 beginrecord
	duration	定义同 beginrecord
返回值	ture	布尔变量

● 开始播放录音

程序名	playrecord(dl,loop,id,duration)	
功能	开始播放录音	
输入参数	dl	模块下行（耳机或手柄或喇叭）是否可以听到录音播放的声音。 0：不出音，1：出音
	loop	录音的播放模式。 0：只播放一次录音，1：循环播放录音
	id	定义同 beginrecord
	duration	定义同 beginrecord
返回值	ture	布尔变量

● 停止播放录音

程序名	stoprecord(dl,loop,id,duration)	
功能	停止播放录音	
输入参数	dl	定义同 playrecord
	loop	
	id	
	duration	
返回值	ture	布尔变量

● 删除录音

程序名	delrecord(id,duration)	
功能	删除录音文件	
输入参数	id	定义同 beginrecord
	duration	定义同 beginrecord
返回值	ture	布尔变量

7.11 misc 模块详解

● 设置当前时钟

程序名	setclock(t,rsfunc)	
功能	设置当前时钟	
输入参数	t	是一个表，键值有 year,month,day,hour,min,sec, 取值均为字符型，其中 year 是 4 位字符的字符串，其余是 1~2 位字符的字符串

	rspfunc	设置时钟时所开启的功能， function 型。如果不用，该参数可以为空
返回值	无	

- 获取当前时间串

程序名	getclockstr()
功能	获取当前时间串
输入参数	无
返回值	当前时间，字符串型。形式如：“YYMMDDHHMMSS”，年月日时分秒均为 2 位字符

- 判断当前时间是星期几

程序名	getweek()
功能	判断当前是星期几
输入参数	无
返回值	星期几，整数型。

- 获取当前日期和时间

程序名	getclock()
功能	获取当前日期和时间
输入参数	无
返回值	当前日期和时间，以表的形式，该表键值有 year,month,day,hour,min,sec ，取值均为字符型，其中 year 是 4 位字符的字符串，其余是 1~2 位字符的字符串

- 获取模块中的用户 SN 号码

程序名	getsn()
功能	获取模块 SN 号码 该号码由模块开机自动发 AT+WISN? 获得，用户只需要调用 getsn() 这个接口函数就能获得 SN 号码
输入参数	无
返回值	用户定制化的/专属的 SN 号，字符串型

- 获取模块 IMEI 号码

程序名	getimei()
功能	获取模块 IMEI 号码 该号码由模块开机自动发 AT+CGSN 获得，用户只需要调用 getimei() 这个接口函数就能获得 IMEI 号码
输入参数	无
返回值	模块 IMEI 号，字符串型

- 获取基础软件版本号

程序名	getbasever()
功能	获取模块基础软件版本号（即：Lod 版本号） 该号码由模块开机自动发 AT+VER 获得，用户只需要调用 getbasever() 这个接口函数就能获得 Lod 版本号
输入参数	无
返回值	模块基础软件版本号，字符串型

7.12 gps 模块详解

- 初始化 gps

程序名	initgps(ionum,dir,edge,period,id,baud,databits,parity,stopbits)	
功能	初始化 gps	
输入参数	ionum	GPIO 口的端口号。 pio.P0_0 - pio.P0_31 表示 GPIO_0 - GPIO_31 pio.P1_0 - pio.P1_9 表示 GPO_0 - GPO_9
	dir	GPIO 方向。通过 pio.pin.setdir（）函数来设置。本函数中目前只支持 pio.OUTPUT
	edge	boolean 型，上升沿开机还是下降沿开机 true: 上升沿开机 false: 下降沿开机
	period	读 gps 通讯端口的周期。整数型，单位是毫秒 gps 通讯端口目前只支持 uart 口
	id	gps uart 口的端口号，整数型
	baud	gps uart 口的波特率，整数型
	databits	gps uart 口的数据位，整数型
	parity	gps uart 口的校验位，整数型，有三种取值： uart.PAR_EVEN, uart.PAR_ODD 或 uart.PAR_NONE
	stopbits	gps uart 口的停止位，整数型，有三种取值： uart.STOP_1 (for 1 stop bit), uart.STOP_1_5 (for 1.5 stop bits) 或 uart.STOP_2 (for 2 stop bits)
返回值	无	

- 打开 gps

程序名	opengps(tag)	
功能	打开 gps。gps 芯片上电，并且打开 GPS uart 口	
输入参数	tag	应用模块的名称，字符串型。同 wake(tag) 中的 tag 定义
返回值	无	

- 关闭 gps

程序名	closegps(tag)	
功能	关闭 gps。gps 芯片下电，并且关闭 GPS uart 口	
输入参数	tag	应用模块的名称，字符串型。同 wake(tag)中的 tag 定义
返回值	无	

- 设置 gps 时区

程序名	settimezone(zone)	
功能	设置 gps 时区	
输入参数	zone	时区，整数型。目前支持 2 个取值： GPS_GREENWICH_TIME 和 GPS_BEIJING_TIME
返回值	无	
举例	<p>设置时区为北京时区（东八区）</p> <pre>local gps = require"gps" gps.settimezone(gps.GPS_ BEIJING_TIME)</pre>	

- 设置 gps 芯片类型

程序名	setchiptype(typ)	
功能	设置 gps 芯片类型	
输入参数	typ	gps 芯片类型，整数型。目前支持 2 个取值： GPS_UBLOX：是外部 gps 芯片 GPS_RDA：是模块内部 RDA gps 芯片
返回值	无	

- 设置 gps 速度类型

程序名	setspdtype(typ)	
功能	设置 gps 速度类型	
输入参数	typ	gps 芯片上报的类型，整数型。目前支持 2 个取值： GPS_KNOT_SPD：海里/小时 GPS_KILOMETER_SPD = 公里/小时
返回值	无	
举例	<p>设置 gps 上报的速度类型是公里/小时</p> <pre>local gps = require"gps" gps.setspdtyp(gps.GPS_KILOMETER_SPD)</pre>	

- 获取终端经纬度位置信息

程序名	getgpslocation(format)	
功能	获取终端经纬度位置信息	
输入参数	format	经纬度的类型，整数型。目前支持 2 个取值： GPS_DEGREES：度。例如：纬度为 XX.YYYYYY，则为 XX.YYYYYY 度 GPS_DEGREES_MINUTES：度分。例如纬度为 XX.YYYYYY，则 XX 为度，YY.YYY 为分。
返回值	位置信息。格式如：E/W,long,N/S,lati E/W:东经或西经 long: 经度 N/S:北纬或南纬 lati: 纬度 举例：E,121.5259850,N,31.2356616	

- 获取 gps 卫星信息

程序名	getsatesinfo()	
功能	获取 gps 卫星信息	
输入参数	无	
返回值	卫星信息，字符串型。格式如：0428 0130 3230 2028 1133 0729 04 01 31 20 11 07 是找到的卫星的 id 28 30 30 28 33 29 是每个卫星的信号强度	

- 获取卫星数量

程序名	getgpssatenum()	
功能	获取找到的卫星的数量	
输入参数	无	
返回值	找到的卫星的数量，整数型	

- 获取终端速度

程序名	getgpsspd()	
功能	获取由 gps 侦测到的终端速度，速度单位由 setspdtype(typ)来确定是公里/小时还是海里/小时	
输入参数	无	
返回值	终端速度，整数型	

- 获取终端方位角

程序名	getgpscog()	
功能	获取由 gps 侦测到的终端方位角	

输入参数	无
返回值	终端速度，整数型。取值：0~359.9

- 判断 **gps** 是否已经打开

程序名	isopen()
功能	判断 gps 是否已经打开
输入参数	无
返回值	Boolean 型 true: 表示 gps 已经打开 false: 表示 gps 未打开

- 判断终端 **gps** 是否已经成功定位

程序名	isfix()
功能	判断终端 gps 是否已经成功定位
输入参数	无
返回值	Boolean 型 true: 表示 gps 已经成功定位 false: 表示 gps 未定位

8. 消息分发函数 **dispatch** 详细介绍

在程序任何地方 **dispatch** 出来的消息和参数, 会先放入内部消息队列 **qmsg** 中, 然后通过 **sys.run()** 中的 **runqmsg()** 函数自动在 **apps** 表中查找并执行对应的回调处理函数 (在本文前面的章节中已经提到过, 对应的回调处理函数是通过 **regapp** 注册的, 注册形式有 **table** 和 **function** 两种)。

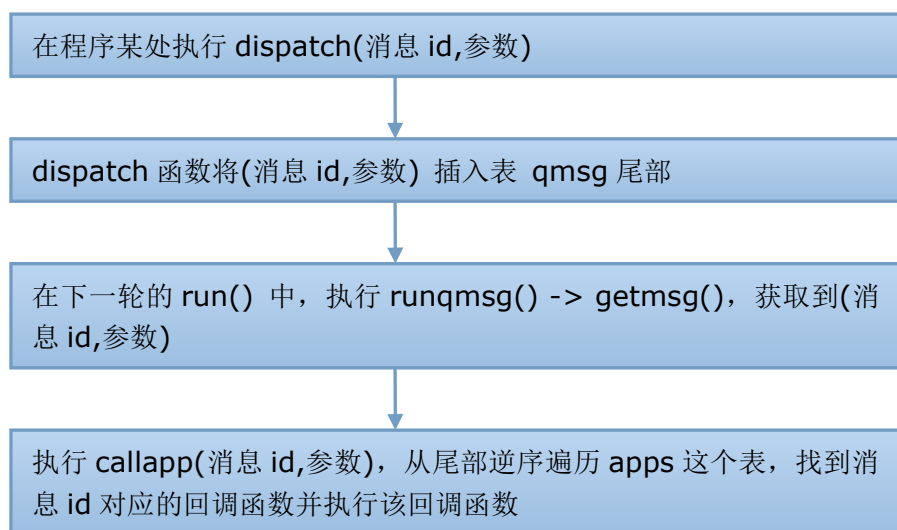


图 dispatch 过程流程图

有一点需要注意，在应用程序中，同一个消息 **id** 可以对应不同的处理回调函数，比如需要广播的一些消息。这种情况下如果要把所有的回调函数都能执行到，需要在所有回调函数中需要返回的地方都要 **return true**，因为在任意回调函数中遇到 **return false** 的情况，就会退出查找并执行所有回调函数这个过程，导致其余的回调执行不到。

在本章中将着重介绍 lib 库各个 module 中 dispatch 出来的消息和相关的参数，以方便客户的使用。

8.1 cc 模块中的 dispatch

dispatch("CALL_CONNECTED")	当 Lua 从 ATC 口收到 CONNECT 这个 URC 时，此时会 dispatch 这个消息出来，表示通话接通
dispatch("CALL_READY")	当 Lua 从 ATC 口收到 CALL READY 这个 URC 时，此时会 dispatch 这个消息出来，表示已经注册网络，可以发起电话呼叫了
dispatch("CALL_DISCONNECTED",reason)	当电话被挂断时（包括主动挂断和对方挂断），或无法接通时，会 dispatch 这样一个消息 消息 id: CALL_DISCONNECTED; 参数 reason: 挂断原因, string 型
dispatch("CALL_INCOMING",number)	当有电话呼入时，会 dispatch 这个消息出来告知，并把呼入号码一同告知 消息 id: CALL_INCOMING 参数 number: 呼入号码, string 型

8.2 net 模块中的 dispatch

dispatch("NET_STATE_CHANGED",s)	当网络注册状态发生改变时，会 dispatch 这个消息出来告知，并将网络状态一同告知 消息 id: NET_STATE_CHANGED 参数 s: 网络当前的状态，string 型 (INIT/EGISTERED/UNREGISTER)
dispatch("GSM_SIGNAL_REPORT_IND", success,rssi)	当收到信号查询命令 AT+CSQ 的返回结果时，会 dispatch 此消息告知接收信号强度。 rssi: 接收信号强度，number 型

8.3 sms 模块中的 dispatch

dispatch("SMS_READ_CNF",success,num, data,pos,t,name)	得到读一条短信 AT+CMGR=pos 这个命令的返回结果时会 dispatch 这个消息，告知读短信的结果。 success: 查询这个动作是否成功，true/false，boolean 型 num: 发件人号码，number 型 data: 短信内容，string 型 pos: 短信在存储空间的位置，string 型 name: 发件人姓名，string 型
dispatch("SMS_DELETE_CNF",success)	得到删除一条短信 AT+CMGD=pos 这条 AT 命令的返回结果后，会 dispatch 这个消息出来，告知删除短信是否成功。 success: 删除短信这个动作是否成功，true/false，boolean 型
dispatch("SMS_SEND_CNF",success)	用 AT+CMGS 这个 AT 命令发送完一条短信，并得到该命令的返回消息后，会 dispatch 这个消息，告知短信发送情况 success: 发送短信这个动作是否成功，true/false，boolean 型
dispatch("SMS_READY")	收到 SMS READY 这个 URC 后，会 dispatch 这个消息出来，告知短信初始化已经完成
dispatch("SMS_NEW_MSG_IND",pos)	收到+CMTI: pos 这个 URC 后，会 dispatch 这个消息出来，告知收到了一条新短信 pos: 短信在存储空间的位置，string 型

8.4 pb 模块中的 dispatch

<code>dispatch("PB_FIND_CNF",success,index,n,name)</code>	当收到查找电话本记录 AT+CPBF 这个 AT 命令的返回结果时会 dispatch 这么一条消息，告知查询结果。 success : 查找电话本记录这个动作是否成功， true/false , boolean 型 index : 查到的电话本记录的索引， string 型 n : 电话号码， string 型 name : 名字， string 型
<code>dispatch("PB_READ_CNF",success,index,n,name)</code>	当收到读取一条电话本记录 AT+CPBR 这个 AT 命令的返回结果时会 dispatch 这么一条消息，告知读取结果。 success : 读取电话本记录这个动作是否成功， true/false , boolean 型 index : 查到的电话本记录的索引， string 型 n : 电话号码， string 型 name : 名字， string 型
<code>dispatch("CPBS_READ_CNF",success,storage,used,total)</code>	当收到查询电话本存储类型 AT+CPBS? 这个 AT 命令的返回结果时，会 dispatch 这样一条消息，以告知查询结果 success : 查询这个动作是否成功， true/false , boolean 型 storage : 存储类型（ SM/ME/VM/ON 等）， string 型 used : 已用的记录个数， number 型 total : 总共可存储的记录个数， number 型

8.5 audio 模块中的 dispatch

<code>dispatch("AUDIO_DTMF_DETECT",dtmf)</code>	当收到 +DTMFDET: 这个 URC，会 dispatch 这个消息出来，以告知检测到 DTMF 或单频音，并将 DTMF 或单频音的值一并告知 dtmf : DTMF 或单频音， string 型，取值范围： 0~9,A~D,*,#,1000Hz 单频音， 1400Hz 单频音， 2300Hz 单频音
<code>dispatch("AUDIO_PLAY_END_IND")</code>	当用 audio.play 播放完音频文件时，会 dispatch 这个消息以告知成功播放完毕
<code>dispatch("AUDIO_PLAY_ERROR_IND")</code>	当用 audio.play 播放完音频文件时，会 dispatch 这个消息以告知播放失败
<code>dispatch("SPEAKER_VOLUME_SET_CNF")</code>	用 AT+CLVL 设置下行音频通道

,success)	(SPEAKER/RECEIVER) 的音量并得到返回结果时，会 dispatch 这个消息出来以告知设置成功或失败 success: 设置音量这个动作是否成功，true/false, boolean 型
dispatch("AUDIO_CHANNEL_SET_CNF", success)	用 AT+CHFA 设置音频通道并得到返回结果时，会 dispatch 这个消息出来以告知设置成功或失败 success: 设置音频通道这个动作是否成功，true/false, boolean 型
dispatch("MICROPHONE_GAIN_SET_CNF", success)	用 AT+CMIC 设置上行通道 (MIC) 的增益并得到返回结果时，会 dispatch 这个消息出来以告知设置成功或失败 success: 设置 MIC 音量这个动作是否成功，true/false, boolean 型

8.6 update 模块中的 dispatch

dispatch("UP_EVT", "UP_PROGRESS_IND", packid * 100 / total)	
dispatch("UP_EVT", "UP_END_IND", succ)	
dispatch("UP_EVT", "NEW_VER_IND", upselcb)	

8.7 misc 模块中的 dispatch

sys.dispatch("SIM_IND", "RDY")	当收到 +CPIN: READY 这个 URC 时，会 dispatch 这个消息出来以告知 SIM 处于在位状态
sys.dispatch("SIM_IND", "NIST")	当收到 +CPIN: NOT INSERTED 这个 URC 时，会 dispatch 这个消息出来以告知 SIM 处于不在位状态

9. 快速入手

本章从一个常用的例子入手，简要介绍根据框架结构和库文件编写应用程序的方法。

举一个例子：开机后检验网络注册状态，如果注册了网络就发起一个语音呼叫，接通后挂断呼叫

第一步：编写 main.lua 程序

-- 加载相关应用或 lib 模块

require "ccapp"

```

sys.init(1)          --AT 口初始化
sys.poweron()        --开机

sys.run()            --主循环程序，程序就在这里循环运行

```

第二步：编写 **ccapp** 这个应用模块（应用程序名称可以随便取）

-- 用 **module(...,package.seeall)** 创建模块（例如 **a**），可以看到并使用其他模块（例如 **c**）的全局变量和全局函数。当然有个前提：在整个脚本程序范围内，曾经 **require** 过 **c**（不一定在 **a** 中 **require**）

```
module(...,package.seeall)
```

```

require"ril"
require"sys"
require"cc"
local req = ril.request          --将 AT 命令发送函数 ril.request 用变量定义以简化一下

```

```

local function setupcall()
    cc.dail("13812345678")      --拨打一个手机号码
end

```

```

local function disconnectcall()
    cc.hangup()                --主动挂断电话
end

```

```

local cctable = {
    --当 Lua 从 ATC 口收到 CALL READY 这个 URC 时，此时会 dispatch 这个 CALL_READY 消息
    --cctable 用 regapp 注册后，每当 CALL_READY 被 dispatch 出来，就会自动执行 setupcall 这
    --个回调函数
    CALL_READY = setupcall,
    CALL_CONNECTED = disconnectcall,
}

```

```

local function cc_init()
    req("AT+CHFA=4")           --设置音频通道
    sys.regapp(cctable)         --以表的形式注册回调函数
end

```

```
cc_init()
```

第三步：用 **lod+lua** 脚本下载工具将基础版本和所有脚本下载到模块中，重新上电开机就能运行借助 RDA 的 **coolwatcher trace** 工具查看脚本中的打印语句输出的内容，可以对脚本进行调试。

10. 例程详解

10.1 定时器如何使用

举例：**10s** 定时器到时后点亮屏幕背光灯

-- 定义一个程序

```
local toffLcd
```

```
toffLcd = function()
```

```
    pmd.ldoset(1,pmd.LDO_LCD)
```

```
end
```

-- 开启定时器

```
sys.timer_start(toffLcd,10000) -- 由于在 sys.run()中有定时器的处理程序，定时器到时后会自动  
                                执行 toffLcd() 程序。sys.run()中在下述程序中处理定时器：
```

```
                                if msg.id == rtos.MSG_TIMER then  
                                    timerfnc(msg.timer_id)
```

10.2 如何映射键盘

我们按照行值*8+列值 作为索引填充矩阵按键映射表，所以只要知道一个 key 的 row, col 值，就能和该键的丝印对应上。例如 MOM 键的 (row, col) = (0,1)，DAD 键的 (row, col) = (2,0)。则映射如下：

```
local keymap = {
```

```
[0] = "END",[1] = "MOM",[3] = "SCHOOL",[16] = "DAD",[18] = "HOME",[9] = "SOS"
```

```
}
```

如果不知道键的行值和列值，可以在 keypad.lua 中的 keymsg () 中加一句打印，把 row, col 打印出来：

```
print("msg.key_matrix_row,msg.key_matrix_col=",msg.key_matrix_row,msg.key_matrix_col)
```

10.3 如何加入一个按键处理程序

--第一步：将 keypad.lua 中的键盘处理程序 prockey 放在 procs 表中 MMI_KEYPAD_IND 的位置

```
local procs = {
```

```
    MMI_REFRESH_IND = lcd.scrrefresh,
```

```
    MMI_KEYPAD_IND = keypad.prockey,
```

```
}
```

```
sys.regapp(idleapp)
```


由于 procs 被注册到 apps 表中，所以当键盘按下的时候，MMI_KEYPAD_IND 消息会被 dispatch，从而会根据消息 MMI_KEYPAD_IND 找到 prockey 程序并执行。

--第二步：在 prockey（）程序中增加键盘处理流程即可。

```
function prockey(key)
    if key == "MOM" then
        ccapp.dial("138*****")
    elseif key == "END" then
        cc.hangup()
    end
end
end
```

10.4 接收短信，并处理短信内容

目前的框架已经支持短信收发，但是需要在 main.lua 中加写应用方面的处理

```
local sms = require"sms"

local function DealSmsInfo (success,num,data,pos,t,name)
    -- 对短信内容的处理
    ... ..
    -- 回复"OK"给短信发送者
    sms.send("OK",num)
end

local smapp = {
    SMS_NEW_MSG_IND = sms.read,
    SMS_READ_CNF = DealSmsInfo
}
sys.regapp(smapp)
```

当出现+CMTI 这个 URC 的时候，SMS_NEW_MSG_IND 会被 dispatch，此时会找到 sms.read 并执行，sms.read 的功能是读短信，是 sms.lua 这个 lib 文件里的函数，所以需要加载 sms 这个 lib 文件。

读短信内容时，会有一个 SMS_READ_CNF 消息 dispatch 出来。DealSmsInfo 程序是对读取到短信的内容进行处理，这个程序放在 main.lua 中，客户根据自己的功能需求自己补充细节。

10.5 如何建立一个 TCP 连接

需求：开机 GPRS 注册网络后，连接某个服务器，用 TCP 连接。而且一直保持这个连接。

实现方法：

建立一个应用模块，比如取名 sck.lua：

```

local base = _G
local string = require"string"
local table = require"table"
local ril = require"ril"
local link = require"link"
module(...)

local prot = "TCP"
-- 举一个网站的例子作为说明，
local svr,port = "device.cmmat.com","1087"

local function sckrecv(id,data)
    --此处写上对接收到的数据报文的处理流程，我司可以提供例程
    print("id,data = ",id,data)
end

local function linksta(id,evt,sta)
    --此处写上对链路状态变化的处理，我司可以提供例程
    print("id,evt,sta = ",id,evt,sta)
end

function StartLink()
    local scklid
    scklid = link.open(linksta,sckrecv)
    link.connect(scklid,prot,svr,port)
end

在 main.lua 中加入：
local sck = require"sck"

sck.StartLink()

```

10.6 如何加入 Lcd 驱动

需求：实现 LCD 屏的驱动

实现方法：

我司已经给出了屏幕的初始化代码，客户只需要将屏幕对应的初始化命令导入即可。

即在初始化 lcd 程序 lcd_init 中，修改屏参 lcd_param 中的初始化命令 initcmd

在 Lua 中，

高 16 位 0x0001 表示 延时

高 16 位 0x0002 表示 发送命令

高 16 位 0x0003 表示 发送数据

将厂家给出的屏幕初始化命令翻译成 Lua 中对应的参数

（一般带 CMD 的表示发送命令，带 INDEX 的表示发送数据，带 delay 的表示超时）

```
LCD_ILI9163B_CMD(0x11); //Exit Sleep  => 0x00020011
Delays(20);  => 0x00010020
LCD_ILI9163B_CMD(0x26); //Set Default Gamma  => 0x00020026
LCD_ILI9163B_INDEX(0x04);  => 0x00030004
... ..
```

10.7 如何在界面显示文字和图片

```
local disp = require"disp"
```

```
disp.puttext      ---显示文字
disp.putimage     ---显示图片
```

具体请参考《Lua 扩展库.chm》

10.8 如何点亮各种灯

点亮键盘灯：

```
local function lightLdo(...)
    for _,v in pairs(arg) do
        -- pmd.ldoset()函数的第一个参数是设置 ldo 亮度，0 - 7 级 0 级关闭
        -- 一般键盘灯亮度等级为 1 就够了，否则电流过大，耗电
        pmd.ldoset(1,v)
    end
end
```

--三个键盘灯同时亮

```
lightLdo(pmd.KP_LEDB, pmd.KP_LEDG, pmd.KP_LEDR)
```

点亮 **GPIO** 所连接的 **LED**：

```
local ioRED = pio.P1_0    -- 红色 LED 灯连接的是 GPIO0
local ioBLUE = pio.P0_1   -- 蓝灯 LED 灯连接的是 GPIO0
```

```
function LtOnGPIO(...)    -- 点亮 GPIO，形参是不定参数
    for _,v in ipairs(arg) do
        pio.pin.sethigh(v)
    end
end
```

```

function LtOffGPIO(...)      -- 灭掉 GPIO
    for _,v in ipairs(arg) do
        pio.pin.setlow(v)
    end
end

LtOnGPIO(ioRED, ioBLUE)      -- 红灯蓝灯都点亮
LtOffGPIO(ioRED, ioBLUE)     -- 红灯蓝灯都灭掉

```

10.9 如何使马达震动

比如使马达震动三秒钟

```

function MotorStart()
    pmd.ldoset(1,pmd.LDO_VIB)
end

```

```

function MotorStop()
    pmd.ldoset(0,pmd.LDO_VIB)
end

```

-- t : duration of vibration,ms

```

function MotorVib(t)
    MotorStart()
    sys.timer_start(MotorStop,t)
end

```

```
MotorVib(3000)
```

10.10 如何使用电池管理消息

请参考 5.1 章节例 3 所做的示例。

10.11 如何使模块休眠和唤醒

lib 库中的 pm.lua 是用来休眠唤醒用的。

pm 中的 sleep(tag): 休眠, wake(tag): 唤醒。tag 是调用模块的名称, 字符串型。

任意应用程序模块都可以调用 pm.sleep 和 pm.wake。

只要存在任何一个应用程序唤醒,则不睡眠。

例如: 脚本中应用程序 a 中需要休眠:

```
pm=require"pm" --将 lib 库中的 pm.lua 文件加载进来
```

```
pm.sleep("a")
```

而脚本中应用程序 b 中需要唤醒：

```
pm.wake("b")
```

最后的结果是：整个模块处于唤醒状态。

10.12 如何使用 debug uart 串口与上位机通讯

假如模块跟上位机通讯的程序是 comm.lua

第一步：设置 debug uart 口

```
local function uartset()  
    -- uart 3 - host uart 波特率必须是 921600 mode=2 使用 ID=0xA2 数据透传  
    uart.setup(3,921600,8,uart.PAR_NONE,uart.STOP_1,2)  
end
```

第二步：编写 debug uart 串口消息处理流程

```
local function uartreader()  
    local s  
  
    while true do  
        s = uart.read(3,"*l",0)  
  
        if string.len(s) ~= 0 then  
            PCcmd(s) --对 debug uart 口消息的具体处理的程序  
        else  
            break  
        end  
    end  
end
```

第三步：设置串口，注册 debug uart 口处理程序

```
pm.wake("comm") -- mm.lua 中使模块不要休眠，休眠状态下无法使模块物理串口通讯  
uartset()  
sys.reguart(3,uartreader)
```

10.13 如何使用物理串口与外部设备通讯

常见的外部设备是 GPS 装置，这里就以 GPS 装置为例。

假如模块跟上位机通讯的程序是 `gps.lua`

第一步：在 `gps.lua` 中配置物理串口

```
local function comset()  
    -- 第一个参数 0:物理串口的 id  
    -- 目前模块只有一个物理串口 (id 是 0 或 2) 和一个 debug uart 口 (id 是 3)  
    uart.setup(0,115200,8,uart.PAR_NONE,uart.STOP_1)  
end
```

第二步：编写 `gps` 数据处理程序

```
local function gpsdata(s)  
  
end
```

第三步：编写物理串口消息处理流程

```
local function comreader()  
    local s  
  
    while true do  
        s = uart.read(0,"*l",0)  
  
        if string.len(s) ~= 0 then  
            gpsdata(s)  --对物理串口消息的具体处理的程序  
        else  
            break  
        end  
    end  
end  
  
end
```

第四步：设置串口，注册物理串口处理程序

```
comset()  
sys.reguart(0,comreader)
```

11. 程序调试环境

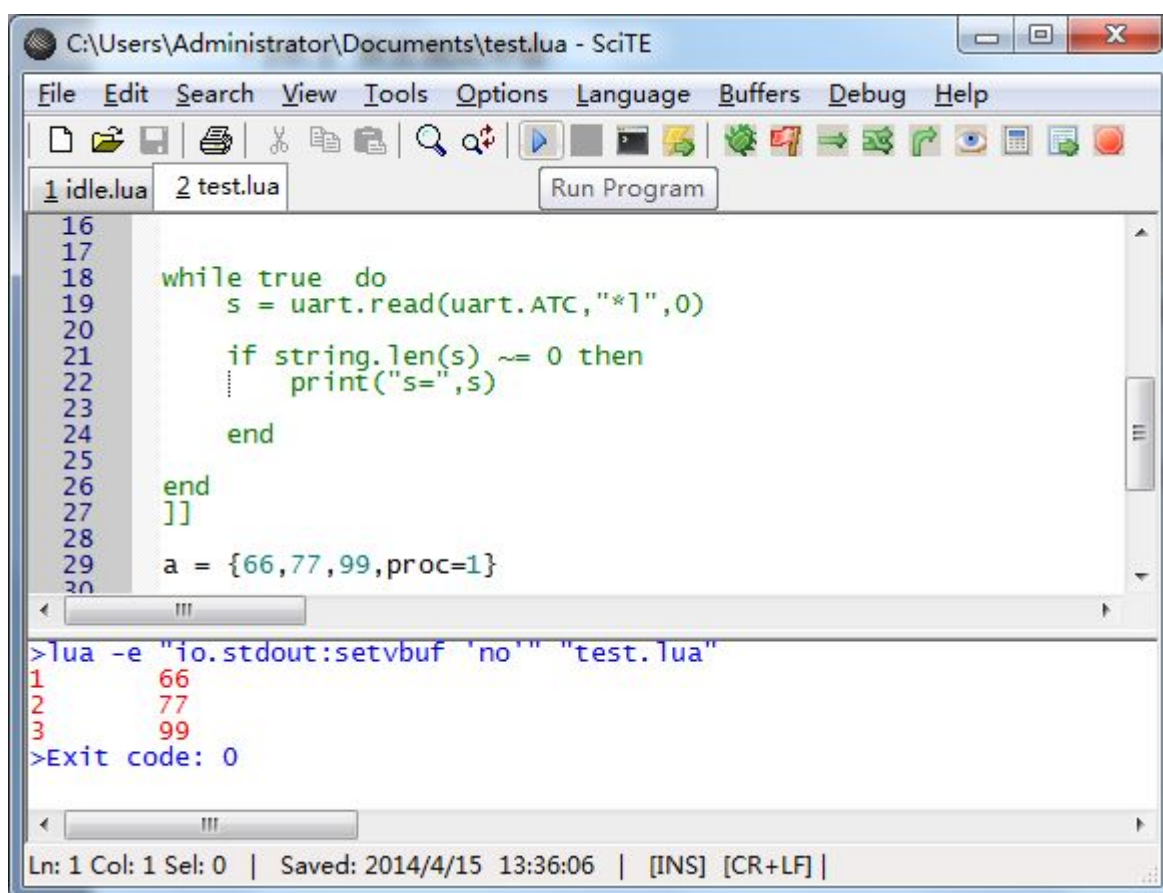
A6390 lua 调试环境包括真实调试环境和 PC 模拟调试环境 2 种。

模块的 `lod` 目前分两种，一种是标准数据模块版本，不支持 `lua` 运行环境；另外一种是本 `doc` 前述的带 `lua` 运行环境的数据模块版本。

在真实的调试环境下，模块需要运行带 `lua` 运行环境的数据模块版本，并可能需要将脚本下载到模块中(详见 10.1 部分)；在模拟调试环境下，模块只需要运行标准数据模块版本的 `lod` (详见 10.2 部分)。

11.1 真实调试环境

- 在程序调试过程中，不涉及跟模块交互的程序部分，可以直接在 windows 的 LUA 编辑运行环境下调试（需要先安装 LuaForWindows.exe）。把相关的程序块 copy 到一个以.lua 为后缀的新建文档中，点击运行按钮即可看到运行结果。

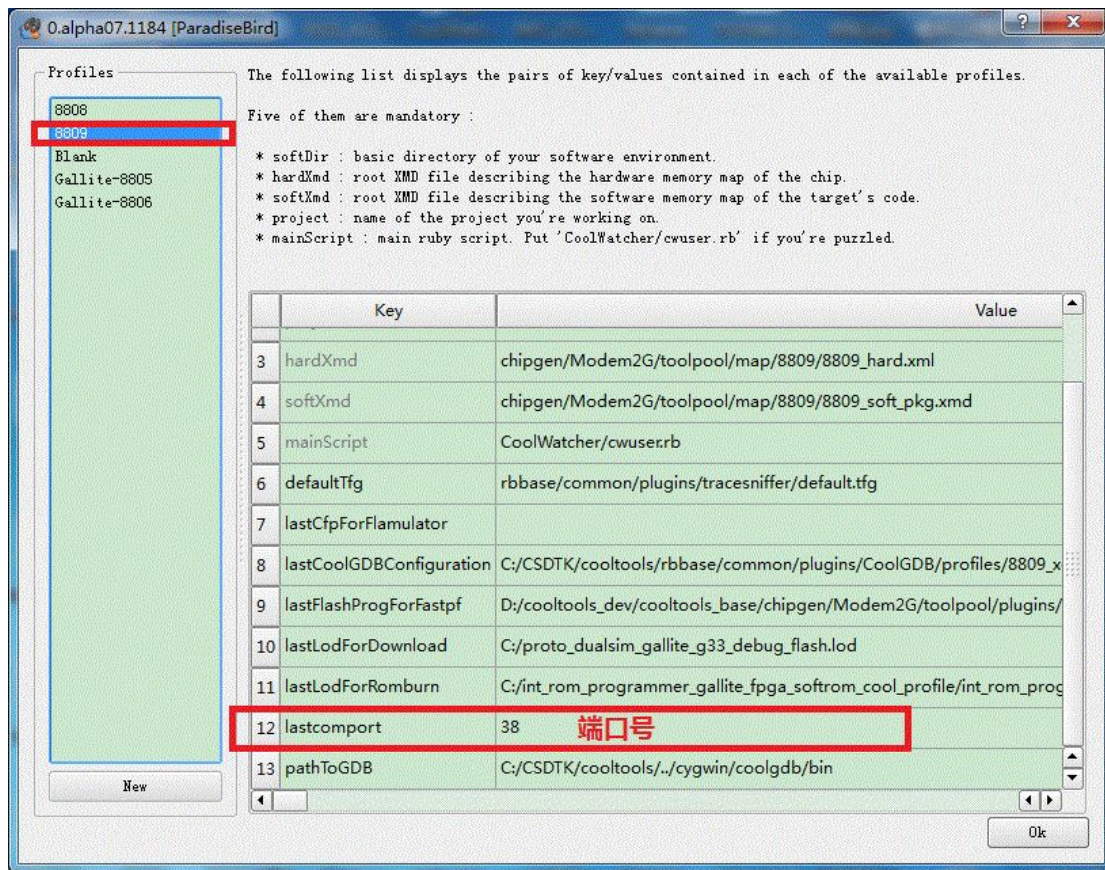


- 另外在程序调试过程中还会涉及到跟模块交互的部分。这种情况下调试程序，需要先将编写的脚本用我司提供的工具下载到模块中，并依靠打印出的语句来调试程序。

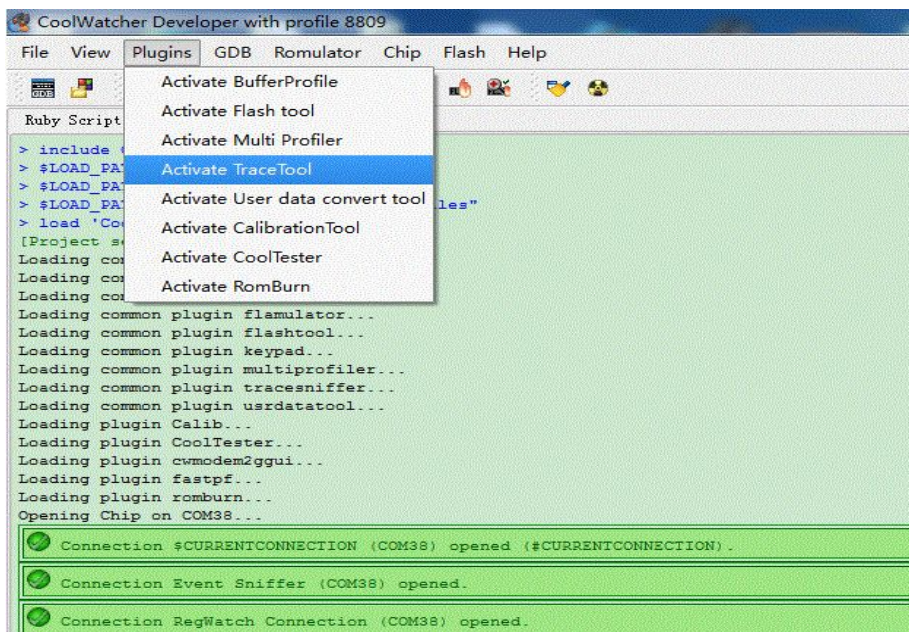
在第二章的 Lua 和模块关系架构图中，可以看到，lua 脚本中的 print 语句的打印内容可以通过 debug uart 口，打印到 trace 工具里。从 trace 提供的信息，可以对脚本进行调试。

目前的 trace 工具有 2 种，一种是 **Coolwatcher**，由模块平台商锐迪科提供。安装和使用步骤简介如下：

1. 安装 CSDTK3.7_Cygwin1.5.25_Svn_1.5.4_Full_Setup.exe，全部默认下一步到完成。
2. 单击 WINDOWS 菜单->所有程序->CoolSand Development ToolKit -> CoolWatcher。
3. 左边选择 8809，右边在 lastcomport 处填入实际设备连接的端口号，此端口号对应于板子的下载口(HST_RX,HST_TX,GND)。

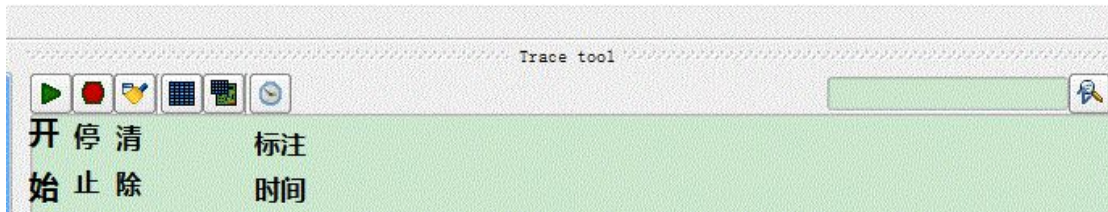


4. 等待串口打开后，在菜单中选择 Plugins->Activate TraceTool

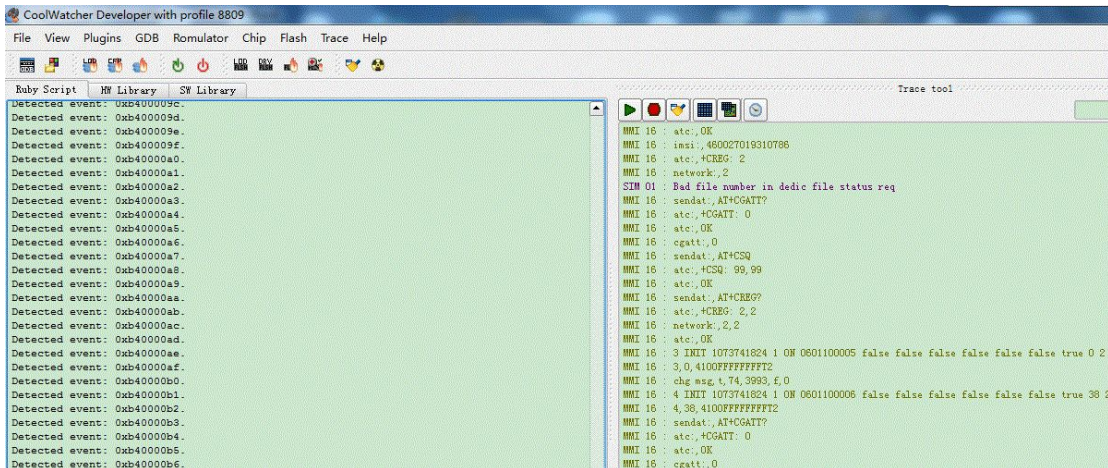


5. 右边的 Trace Tool 图标含义如下：

按下  按钮，会在每条 trace 之前显示当前的电脑时间。



6. 上电开机后，左边窗口会有数据响应。右边窗口会显示出 Lua 脚本的 `print` 语句打印出的内容。用户可以根据打印内容对脚本进行调试。



coolwatcher 的缺点是文件太大。启动速度较慢。

还有一种 `trace` 工具是我司自开发的工具软件 `trace tool`。软件比较小，启动速度也快。



trace tool 软件使用步骤:

1. 运行 `exe` 文件
2. 选择下载口的端口号，并打开端口
3. 开启模块或模块重新上电，就会有打印出来

11.2 PC 模拟调试环境

PC 模拟调试环境由我司开发提供，客户编写的.lua 程序可以在这个环境下运行调试。

这个模拟调试软件相当于把图 1 中的 lua 运行环境和脚本搬到 PC 上运行，通过模块的串口把 AT 命令或自定义命令发送到模块中，模块将命令返回结果传回 PC 调试软件中。示意图见下图。

图 1 所示的环境是真实的运行环境，客户编写的脚本运行在模块之中，此时模块使用的 lod 版本是将 LUA 支撑运行环境合入的；而图 4 的环境是 PC 模拟运行环境，客户编写的脚本运行在此环境中，此时模块所使用的 lod 版本是标准的数据模块版本，不需要支持 lua 运行环境。

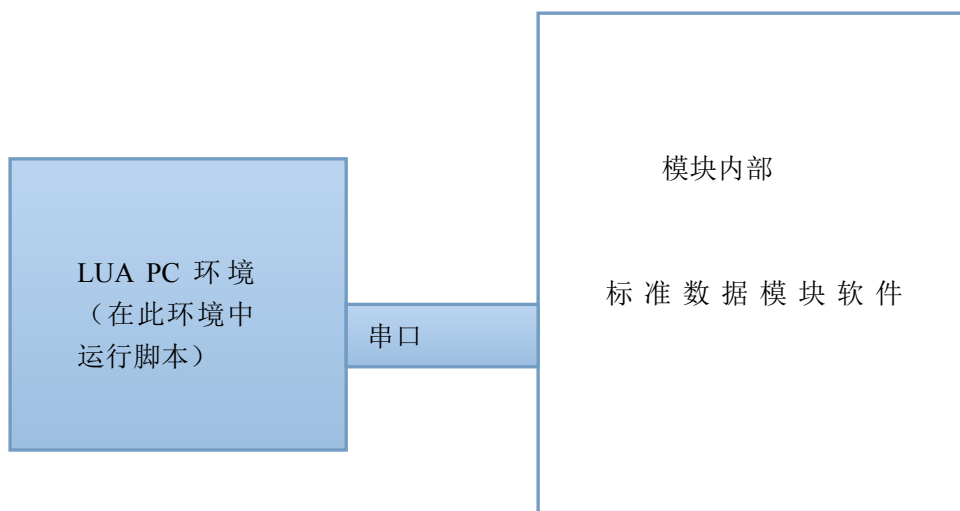
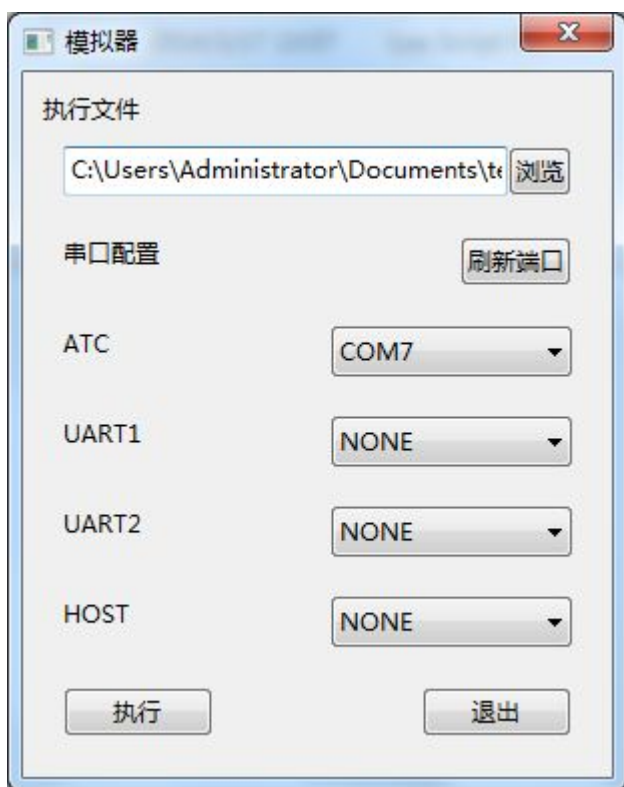


图 模拟调试环境示意图

这个模拟调试环境运行在 PC 上，界面如下：



使用方法如下：

1) 选择好串口类型和端口号；

串口类型和端口号选择方法如下：

如果是调试跟 LOD 进行 AT 命令交互部分的代码，此时需要将标准数据模块的 AT 口用串口线与 PC 相连，在这种情况下，请在设备管理器中查看下端口号，并选择选择相应的 ATC 端口。

如果需要调试的用户程序是通过 UART 口跟外部设备交互（例如 GPS，蓝牙等），则需要将外部设备通过 uart 口与 PC 相连，在这种情况下，请在设备管理器中查看下端口号，并选择选择相应的 UART 端口。

如果需要调试的用户程序是通过 HOST 口与 PC 工具交互，则需要在 PC 上运行一个虚拟串口软件，然后运行 PC 工具并选择端口，同时在虚拟运行环境中选择相应的端口。

2) 选择需要调试的 lua 脚本（.lua 或.bin 为后缀），按【执行】按钮运行。

如果客户编写的程序只有一个.lua 文件，则可以直接在该环境下运行；如果客户编写的应程序由好几个.lua 程序组成，则需要先用我司提供的工具软件先合并成.bin 文件，再在该环境下运行。

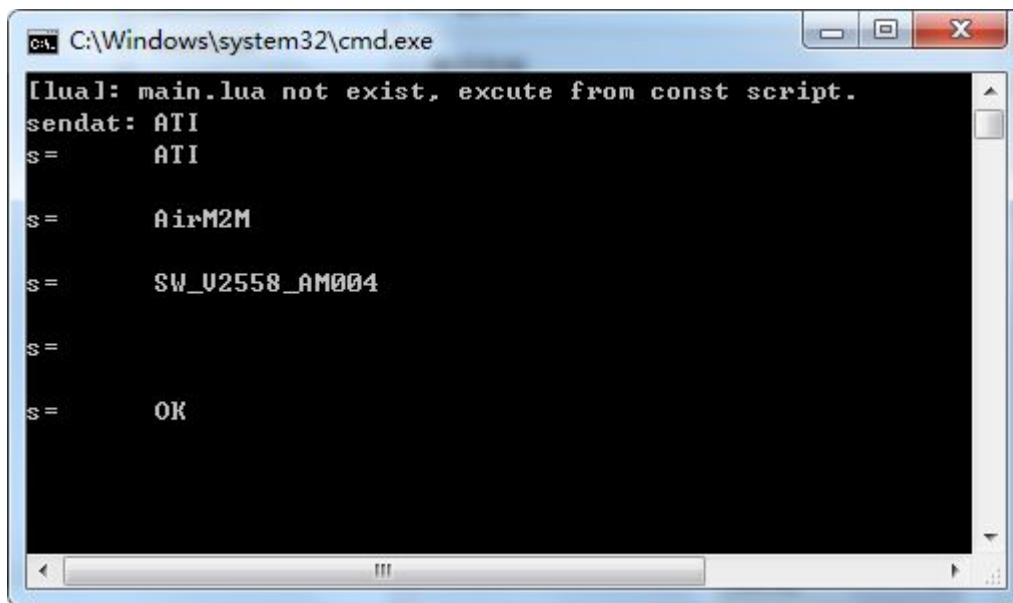
下面以单个待调试脚本 test.lua 为示例，简要介绍使用方法：

```
local function req(currcmd)
    print("sendat:",currcmd)
    uart.write (uart.ATC,currcmd .. "\r")
end
```

```
uart.setup(uart.ATC,0,0,uart.PAR_NONE,uart.STOP_1)
--发送 ATI 这个 AT 命令来查询模块软件的版本
req("ATI")
```

```
while true do
    s = uart.read(uart.ATC,"*l",0)
    if string.len(s) ~= 0 then
        print("s=",s)
    end
end
```

下面是在模拟调试环境中运行的结果：



```
C:\Windows\system32\cmd.exe
[lua]: main.lua not exist, excute from const script.
sendat: ATI
s=      ATI
s=      AirM2M
s=      SW_U2558_AM004
s=
s=      OK
```