nodeG5 Lua application (fw_G5_2_x)

Lua interpreter version: 5.1.5

Revision log:

CONTENTS Page

A. Introduction to LUA

Lua is a powerful scripting language used in many embedded devices worldwide.

Official website www.lua.org

The Lua script interpreter is embedded into the Linux-Debian operating system of the nodeG5.

With Lua script running, user can execute customized functions to

- Save events/data to a file in flash memory.
- Handle Serial RS485 communications with serial devices.
- Handle TCP/UDP communications with host or client.
- Handle raw communications with Modbus RTU and TCP devices.
- Handle raw communications with CANbus devices.

Pause LUA program

sleep(x)
Sleeps for x seconds
Example
require “ampio”
sleep(0.5)
sleep(3)

Ping function

ping(host,interface)
returns 0 if ping failed
returns 1 if ping success
host=valid ip address or domain name eg “8.8.8.8” or “www.lua.org”
interface=”eth0”,”eth1” or omitted for default route
Example
require “ampio”
pingcam1=ping(“192.168.1.100”,”eth0”)
pingcam2=ping(“10.1.1.200”,”eth1”)
pingdnsserver=ping(“8.8.8.8”)
pingdomain=ping(“www.microsoft.com”)

B. SERIAL PORT functions

Open/close serial RS485

serial_open(port,baud_rate,data_bits,stop_bits,parity)
open serial port with defined port number, baud rate, data bits, stop bits and parity.

serial_close(port)
close serial port with defined port number

serial_raw(port)
set serial port with defined port number to raw mode

Port number	I/O module	Connector	Type	Signal description	Information
1	A	INDUSTRIAL I/O (P8)	RS485	RS485_POS (pin 1) RS485_NEG (pin 3) ISO_GND1 (pin 5)	2-wire half duplex
2	B	INDUSTRIAL I/O (P8)	RS485	RS485_POS (pin 7) RS485_NEG (pin 6) ISO_GND2 (pin 8)	2-wire half duplex

baud_rate = 2400, 4800, 9600, 19200, 38400, 57600, 115200
data_bits = 7, 8
stop_bits = 1, 2
parity = “NONE”, “EVEN”, “ODD”

Sending serial RS485

serial_send(port,message,timeout_sec)

Send a message in ASCII code
Example:

```
require "ampio"
serial_open(1,115200,8,1,”NONE”)          --opens port1 @115200baudrate,8databit,1stopbit,no parity
message="SEND MESSAGE FROM LUA\r\n"
serial_send(1,message,10)                 --send message via port1 with timeout=10sec
print("send="..message)
serial_close(1)                           --close port1
```

Send a message in HEX/BIN value.
Example:

```
require “ampio”
serial_raw(1)                             --set port1 to raw mode
serial_open(1,115200,8,1,”NONE”)          --opens port1 @115200baudrate,8databit,1stopbit,no parity
hexdata=string.char(0x01,0x04,0x1A,0x2B,0x3F)
serial_send(1,hexdata,10)                 --send hexdata via port1 with timeout=10sec
serial_close(1)                           --close port1
```

Reading serial RS485

serial_read(port,end_char,timeout_sec)
end_char = return when receive the end of message character
returns nil when there is no serial data received
returns serial data upon receiving end_char or upon timeout

Example (ASCII code):

```
require "ampio"
serial_open(1,115200,8,1,"NONE")        --opens port1 @115200baudrate,8databit,1stopbit,no parity
end_char="\n"                           --return on receiving newline char
rxstr=serial_read(1,end_char,10)        --read port1, returns only upon end_char or timeout=10sec
print("serial read data = "..rxstr)
serial_close(1)                         --close port1
```

Example (HEX/BIN code):

```
require "ampio"
serial_open(1,115200,8,1,"NONE")        --opens port1 @115200baudrate,8databit,1stopbit,no parity
end_char=string.char(0x00)              --return on receiving char value 0x00
rxstr=serial_read(1,end_char,10)        --read port1, returns only upon end_char or timeout=10sec
byte2 = string.byte(rxstr,2)            --convert non-printable char to numerical code
byte3 = string.byte(rxstr,3)
print(byte2, byte3)
serial_close(1)                         --close port1
```

serial_receive(port,length,timeout_sec)
length = serial input buffer maximum size
returns nil when there is no serial data received
returns serial data upon input buffer full or upon timeout

Example (ASCII code):

```
require "ampio"
serial_open(1,115200,8,1,"NONE")        --opens port1 @115200baudrate,8databit,1stopbit,no parity
length=10                               --input buffer max size
rxstr=serial_receive(1,length,10)       --read port1, returns only upon input buffer full or timeout=10sec
print("serial receive data = "..rxstr)
serial_close(1)                         --close port1
```

Example (HEX/BIN code):

```
require "ampio"
serial_open(1,115200,8,1,"NONE")        --opens port1 @115200baudrate,8databit,1stopbit,no parity
length=10                               -- input buffer max size
rxstr=serial_receive(1,length,10)       --read port1, returns only upon input buffer full or timeout=10sec
byte3 = string.byte(rxstr,3)            --convert non-printable char to numerical code
byte4 = string.byte(rxstr,4)
print(byte3, byte4)
serial_close(1)                         --close port1
```

C. Modbus master functions

Initialize Modbus master module

init_mbm()
Function to initialize Lua-modbus library into userspace.
This must be executed once before any Modbus function call in Lua script.

Open new connection context to Modbus slave

L1=mbm.open("rtu_*", baudrate, parity, data, stop, timeout)
returns a valid context L1 (>=0)
"rtu_a" = **Modbus/RTU** device connected to SERIAL_PORT_A (RS485)
"rtu_b" = **Modbus/RTU** device connected to SERIAL_PORT_B (RS485)
baudrate = serial baudrate of RTU device
(1200,2400,4800,9600,19200,38400,57600,115200)
parity = serial parity of RTU device ("N","E","O")
data = serial data bit of RTU device (7,8)
stop = serial stop bit of RTU device (1,2)
timeout = response timeout (seconds)

L1=mbm.open("tcp", ip_address, ip_port, timeout)
returns a valid context L1 (>=0)
"tcp" = **Modbus/TCP** device connected to Ethernet port Eth0 or Eth1
ip_address = IP address of Modbus device (eg "192.168.1.100")
ip_port = IP port of Modbus device (default 502)
timeout = response timeout (seconds)

Connect using connection context to Modbus slave

mbm.connect(L1)
L1 = a valid connection context (>=0) from mbm.open() function
returns 0 if connection successful
returns -1 if connection failed/timeout

Read Boolean status from Modbus slave

FC=01 for read discrete coils
data_8bitTable, status = mbm.fc1(L1, node, coil_address, coil_count, timeout)

FC=02 for read discrete inputs
data_8bitTable, status = mbm.fc2(L1, node, input_address, input_count, timeout)

Write Boolean state to Modbus slave

FC=05 for write single discrete coil
status = mbm.fc5(L1, node, coil_address, coil_state, timeout)

FC=15 for write multiple coils
status = mbm.fc15(L1, node, coil_address, coil_count, timeout, data_8bitTable)

Read data registers from Modbus slave

FC=03 for read holding registers (40,001 in old Modicon convention)
data_16bitTable, status = mbm.fc3(L1, node, reg_address, reg_count, timeout)

FC=04 for read input registers (30,001 in old Modicon convention)
data_16bitTable, status = mbm.fc4(L1, node, reg_address, reg_count, timeout)

Write data registers to Modbus slave

FC=06 for write single register
status = mbm.fc6(L1, node, reg_address, data_16bit, timeout)

FC=16 for write multiple registers
status = mbm.fc16(L1, node, reg_address, reg_count, timeout, data_16bitTable)

xxx_address	= address of first coil/input/register to be read/write
xxx_count	= number of coils/inputs/registers to be read/write
coilstate	= integer with value 0 or 1
data_16bit	= 16bit unsigned integer
data_8bitTable	= array of 8bit unsigned integer elements
data_16bitTable	= array of 16bit unsigned integer elements
L1	= a valid connection context for the Modbus device
node	= Modbus/RTU slave address
	= Modbus/TCP node = 1
timeout	= response timeout (seconds)
status	= returns 1 for read/write success

Turn on/off the debug messages

mbm.debug(L1,debug)

debug	= 0	(turn off)
	= 1	(turn on)

Note: Debug messages when running script in console only.

Disconnect from Modbus slave using connection context

mbm.disconnect(L1)
Disconnect the Modbus connection context L1.

Close/free the Modbus connection context

mbm.close(L1)
Close and free the Modbus connection context L1.

Example for Modbus/RTU:

```
require “ampio”

init_mbm()

baud=115200
parity=”N”
data=8
stop=1
timeout=10

L1 = mbm.open(“rtu_a”,baud,parity,data,stop,timeout)
status1 = mbm.connect(L1)
mbm.debug(L1,1)

if status1==0 then

        print("mbm.connect test connect success")
        node=3
        reg_addr=2000
        reg_cnt=10
        datawrite_16bitTable={0x1111,0x2222,0x3333,0x4444,0x5555,0x6666,0x7777,0x8888,0x9999,0xAAAA}

        stat1 = mbm.fc16(L1,node,reg_addr,reg_cnt,timeout,datawrite_16bitTable)
        if stat1==1 then
                print("mbm.fc16 test write register success”)
        end

        dataread_16bitTable,stat2 = mbm.fc3(L1,node,reg_addr,reg_cnt,timeout)
        if stat2==1 then
                print ("mbm.fc3 test read register success")
                i=0
                while (i<reg_cnt) do
                        i=i+1
                        print(dataread_16bitTable[i])
                end
        end

        mbm.disconnect(L1)
end
mbm.close(L1)
```

Example for Modbus/TCP:

```
require “ampio”

init_mbm()

ip_addr=”192.168.1.100”
ip_port=502
timeout=10

L1 = mbm.open(“tcp”,ip_addr,ip_port,timeout)
status1 = mbm.connect(L1)
mbm.debug(L1,1)

if status1==0 then

	print("mbm.connect test connect success")
	node=1
	reg_addr=1000
	reg_cnt=10
	datawrite_16bitTable={0x1111,0x2222,0x3333,0x4444,0x5555,0x6666,0x7777,0x8888,0x9999,0xAAAA}

	stat1 = mbm.fc16(L1,node,reg_addr,reg_cnt,timeout,datawrite_16bitTable)
	if stat1==1 then
		print ("mbm.fc16 test write register success")
	end

	dataread_16bitTable,stat2 = mbm.fc3(L1,node,reg_addr,reg_cnt,timeout)
	if stat2==1 then
		print ("mbm.fc3 test read register success")
		i=0
		while (i<reg_cnt) do
			i=i+1
			print(dataread_16bitTable[i])
		end
	end

	mbm.disconnect(L1)
end
mbm.close(L1)
```

D. CANbus functions

Initialise on-board CANbus interface (specs: CAN 2.0A/B)

can_open(port,baud_rate)

port = canE (main CAN port)
= canC (secondary CAN port)
baud_rate = target CANbus system baud rate eg 100000,125000,250000,500000,etc
This must be executed once before any CANbus function call.

Sending CANbus messages

can_send(port,canid,extended,data_send)

port = canE (main CAN port)
= canC (secondary CAN port)
canid = 000-7FF (3 hex chars for CAN 2.0A)
= 00000000-1FFFFFFF (8 hex chars for CAN 2.0B)
extended = 0 (11-bit message ID for CAN 2.0A)
= 1 (29-bit message ID for CAN 2.0B)
data_send = ASCII hex chars of value 1-byte (eg 1A) to 8-bytes (eg 1A2B3C4D5E6F7A8B)
option to separate bytes by '.' , e.g. 11.22.33.44.55.66.77.88

Receiving CANbus messages

data_receive = can_receive(port,canid,canmask,timeout)

returns 0 when no validated CANbus message received on timeout
port = canE (main CAN port)
= canC (secondary CAN port)
canid = 000-7FF (3 hex chars for CAN 2.0A)
= 00000000-1FFFFFFF (8 hex chars for CAN 2.0B)
canmask = 000-7FF (bit value 0 mask out message)
= 00000000-1FFFFFFF (bit value 0 mask out message)
timeout = receive timeout (seconds)

Mask usage example:
canid=0x5A0, canmask=0x7FF
canid & canmask = 0x5A0
Above mask will receive message with ID 0x5A0

canid=0x280, canmask=0x280
canid & canmask = 0x280
Above mask will receive message with ID 0x280 to 0x28F.

Storing CANbus messages into buffer

can_storebuff(port,canid,canmask)
This must be executed once to store CANbus messages into buffer.
port = canE (main CAN port)
= canC (secondary CAN port)
canid = 000-7FF (3 hex chars for CAN 2.0A (standard))
= 00000000-1FFFFFFF (8 hex chars for CAN 2.0B (extended))
canmask = 000-7FF (bit value 0 mask out message)
= 00000000-1FFFFFFF (bit value 0 mask out message)

Reading CANbus messages from buffer

can_buff = can_readbuff(port)
returns buffer content and buffer emptied after reading
port = canE (main CAN port)
= canC (secondary CAN port)

Close the CANbus interface

can_close(port)
Close the CANbus interface.
Port = canE (main CAN port)
= canC (secondary CAN port)

Example:

```
require “ampio”
port=”canE”
can_open(port,125000)               --baud rate=125000bps
sleep(0.2)

id1=”280”                           --canid=0x280
extended1=0                         --standard mode
data1=”AABBCC”                      --data=0xAA,0xBB,0xCC
can_send(port,id1,extended1,data1)

id2="1F177678"                      --canid=0x1F177678
extended2=1                         --extended mode
data2="11.22.33.44.55.66.77.88"   --data=0x11,0x22,0x33,0x44,0x55,0x66,0x77,0x88
can_send(port,id2,extended2,data2)

canid={}
canmask={}
canid[1]="111";canid[2]="222";canid[3]="333";canid[4]="444";canid[5]="555"
canmask[1]="7FF";canmask[2]="7FF";canmask[3]="7FF";canmask[4]="7FF";canmask[5]="7FF";
timeout=10        --seconds

can_storebuff(port,canid, canmask) --store can messages to buffer

while true do
        can_buff=can_readbuff(port)          --read messages from buffer
        print(can_buff)
        sleep(1)
end

can_close(port)
```