nodeG5 Python application (fw_G5_2_7 and later)

Python interpreter version: 3.9.2

Revision log:

CONTENTS Page

A. Introduction to Python 2
B. Serial Port functions 3
C. Modbus master functions 5
D. Python-CAN support 10

A. Introduction to Python

Python is a widely used scripting language for task automation and also as high-level object-oriented programming language for data processing/analysis, websites and etc .

Official website www.python.org

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

With Python 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.

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.python.org”
interface=”eth0”,”eth1” or “” for default route
Example

```
import ampio
pingcam1=ampio.ping(“192.168.1.100”,”eth0”)
pingcam2=ampio.ping(“10.1.1.200”,”eth1”)
pingdnsserver=ampio.ping(“8.8.8.8”,””)
pingdomain=ampio.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_GND (pin 8)	2-wire hald duplex

baud_rate = 2400, 4800, 9600, 19200, 38400, 57600, 115200
data_bits = 7, 8
stop_bits = 1, 2
parity = ‘N’, ‘E’, ‘O’   (i.e. None, Even, Odd parity)

Sending serial RS232/485

serial_send(port,message,timeout_sec)

Send a message in ASCII code
Example:

```
import ampio
ampio.serial_open(1,115200,8,1,’N’)         #opens port1 @115200baudrate, 8databit, 1stopbit, no parity
message="SEND MESSAGE FROM PYTHON\r\n"
ampio.serial_send(1,message,10)             #send message via port1 with timeout=10sec
print "send=",message
ampio.serial_close(1)                       #close port1
```

Send a message in HEX/BIN value.
Example:

```
İmport ampio
ampio.serial_raw(1)                         #set port1 to raw mode
ampio.serial_open(1,115200,8,1,’N’)         #opens port1 @115200baudrate, 8databit, 1stopbit, no parity
hexdata=[0x01,0x04,0x1A,0x2B,0x3F]
ampio.serial_send(1,hexdata,10)             #send hexdata via port1 with timeout=10sec
ampio.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 None when there is no serial data received
returns serial data upon receiving end_char or upon timeout

Example (ASCII code):

```
İmport ampio
ampio.serial_open(1,115200, 8,1,’N’)          #opens port1 @115200baudrate, 8databit, 1stopbit, no parity
end_char=chr(10)                              #return on receiving char value 10 (NEW LINE)
rxstr=ampio.serial_read(1,end_char,10)        #read port1, returns only upon end_char or timeout=10sec
print "serial read data = ", rxstr
ampio.serial_close(1)                         #close port1
```

Example (HEX/BIN code):

```
İmport ampio
ampio.serial_open(1,115200, 8,1,’N’)          #opens port1 @115200baudrate, 8databit, 1stopbit, no parity
end_char=chr(27)                              #return on receiving char value 27 (ESC)
rxstr=ampio.serial_read(1,end_char,10)        #read port1, returns only upon end_char or timeout=10sec
byte2 = ord(rxstr[2])                         #convert non-printable char to char value
byte3 = ord(rxstr[3])
print “serial read data = ”, byte2, byte3
ampio.serial_close(1)                         #close port1
```

serial_receive(port,length,timeout_sec)
length = serial input buffer maximum size
returns None when there is no serial data received
returns serial data upon input buffer full or upon timeout

Example (ASCII code):

```
import ampio
ampio.serial_open(1,115200, 8,1,’N’)          #opens port1 @115200baudrate, 8databit, 1stopbit, no parity
length=10                                     #input buffer max size
rxstr=ampio.serial_receive(1,length,10)       #read port1, returns only upon input buffer full or timeout=10sec
print “serial receive data = “, rxstr
ampio.serial_close(1)                         #close port1
```

Example (HEX/BIN code):

```
import ampio
ampio.serial_open(1,115200, 8,1,’N’)          #opens port1 @115200baudrate, 8databit, 1stopbit, no parity
length=10                                     #input buffer max size
rxstr=ampio.serial_receive(1,length,10)       #read port1, returns only upon input buffer full or timeout=10sec
byte3 = ord(rxstr[3])                         #convert non-printable char to char value
byte4 = ord(rxstr[4])
print “serial receive data = ”, byte3, byte4
ampio.serial_close(1)                         #close port1
```

C. Modbus master functions

Interfacing with Modbus master library using ctypes

```
from ctypes import *
cdll.LoadLibrary(“libg5modbus.so”)
g5mb = CDLL(“libg5modbus.so”)
```

Python ctypes interface with modbus library to allow calling functions in the shared library. This must be executed once before any Modbus function call in Python script.

Open new connection context to Modbus slave

L1 = g5mb.mbmOpenRTU (port, baudrate, parity, data, stop, timeout)
returns a valid context L1 (>=0)
Modbus/RTU device connected to SERIAL RS485 port of nodeG5
port = serial port of nodeG5 (“rtu_a”,“rtu_b”)
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 = g5mb.mbmOpenTCP (ip_address, ip_port, timeout)
returns a valid context L1 (>=0)
Modbus/TCP device connected to Ethernet port of nodeG5
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

status = g5mb.mbmConnect(L1)
L1 = a valid connection context (>=0) from mbmOpenXXX() function
returns 0 if connection successful
returns -1 if connection failed/timeout

Read Boolean status from Modbus slave

FC=01 for read discrete coils
status = g5mb.mbmFC1 (L1, node, coil_address, coil_count, timeout, byref (data_8bitTable), byref (size))

FC=02 for read discrete inputs
status = g5mb.mbmFC2 (L1, node, input_address, input_count, timeout, byref (data_8bitTable), byref (size))

Write Boolean state to Modbus slave

FC=05 for write single discrete coil
status = g5mb.mbmFC5 (L1, node, coil_address, coil_state, timeout)

FC=15 for write multiple coils
status = g5mb.mbm.FC15 (L1, node, coil_address, coil_count, timeout, data_8bitTable, size)

Read data registers from Modbus slave

FC=03 for read holding registers (40,001 in old Modicon convention)
status = g5mb.mbmFC3 (L1, node, reg_address, reg_count, timeout, byref (data_16bitTable), byref (size))

FC=04 for read input registers (30,001 in old Modicon convention)
status = g5mb.mbmFC4 (L1, node, reg_address, reg_count, timeout, byref (data_16bitTable), byref (size))

Write data registers to Modbus slave

FC=06 for write single register
status = g5mb.mbmFC6 (L1, node, reg_address, data_16bit, timeout)

FC=16 for write multiple registers
status = g5mb.mbmFC16 (L1, node, reg_address, reg_count, timeout, data_16bitTable, size)

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
size	= number of elements in data_xxTable array
byref()	= passing parameters by reference (not by value)
L1	= a valid connection context for the Modbus device (>=0)
node	= Modbus/RTU slave address
	= Modbus/TCP node = 1
timeout	= response timeout (seconds)
status	= returns 0 for read/write success

Turn on/off the debug messages

g5mb.mbmDebug(L1,debug)

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

Note: Debug messages when running script in console mode only.

Disconnect the Modbus context

g5mb.mbmDisconnect(L1)
Disconnect the Modbus connection context L1.

Close the Modbus context

g5mb.mbmClose(L1)
Close and free the Modbus connection context L1.

Example Modbus/RTU:

```
from ctypes import *

try:
    cdll.LoadLibrary("libg5modbus.so")
except:
    print "Unable to load libg5modbus.so"
    exit(0)

g5mb = CDLL("libg5modbus.so")

port=”rtu_a”
baud=115200
parity=”N”
data=8
stop=1
timeout=10
L1 = g5mb.mbmOpenRTU(port,baud,parity,data,stop,timeout)
status1 = g5mb.mbmConnect(L1)
g5mb.mbmDebug(L1,1)

if status1==0:

    print "mbmConnect test connect success"
    node=3
    reg_addr=2000
    reg_cnt=10
    IntArray1 = c_ushort * 10
    datawrite_16bitTable = IntArray1(0x1111,0x2222,0x3333,0x4444,0x5555,0x6666,0x7777,
    0x8888,0x9999,0xAAAA)
    sizearr1 = len(datawrite_16bitTable)

    stat1 = g5mb.mbmFC16(L1,node,reg_addr,reg_cnt,timeout,datawrite_16bitTable,sizearr1)
    if stat1==0:
        print "mbmFC16 test write register success”

    IntArray2 = c_ushort * 10
    dataread_16bitTable = IntArray2(0,0,0,0,0,0,0,0,0,0)
    sizearr2 = c_ushort(0)

    stat2 = g5mb.mbmFC3(L1,node,reg_addr,reg_cnt,timeout,byref(dataread_16bitTable),byref(sizearr2))
    if stat2==0:
        print "mbmFC3 test read register success”
        for i in range (0,reg_cnt):
            print (i, dataread_16bitTable[i])

    g5mb.mbmDisconnect(L1)
g5mb.mbmClose(L1)
```

Example Modbus/TCP:

```
from ctypes import *

try:
        cdll.LoadLibrary("libg5modbus.so")
except:
        print "Unable to load libg5modbus.so"
        exit(0)

g5mb = CDLL("libg5modbus.so")

ip_addr=”192.168.1.100”
ip_port=502
timeout=10
L1 = g5mb.mbmOpenTCP(ip_addr,ip_port,timeout)
status1 = g5mb.mbmConnect(L1)
g5mb.mbmDebug(L1,1)

if status1==0:

        print "mbmConnect test connect success"
        node=1
        reg_addr=1000
        reg_cnt=10
        IntArray1 = c_ushort * 10
        datawrite_16bitTable = IntArray1(0x1111,0x2222,0x3333,0x4444,0x5555,0x6666,0x7777,
        0x8888,0x9999,0xAAAA)
        sizearr1 = len(datawrite_16bitTable)

        stat1 = g5mb.mbmFC16(L1,node,reg_addr,reg_cnt,timeout,datawrite_16bitTable,sizearr1)
        if stat1==0:
                print "mbmFC16 test write register success”

        IntArray2 = c_ushort * 10
        dataread_16bitTable = IntArray2(0,0,0,0,0,0,0,0,0,0)
        sizearr2 = c_ushort(0)

        stat2 = g5mb.mbmFC3(L1,node,reg_addr,reg_cnt,timeout,byref(dataread_16bitTable),byref(sizearr2))
        if stat2==0:
                print "mbmFC3 test read register success”
                for i in range (0,reg_cnt):
                        print (i, dataread_16bitTable[i])

        g5mb.mbmDisconnect(L1)
g5mb.mbmClose(L1)
```

D. python-can library v4.5.0

The **python-can** library provides Controller Area Network support for Python, providing common abstractions to different hardware devices, and a suite of utilities for sending and receiving messages on a CAN bus.

Official website https://pypi.org/project/python-can/

Documents https://python-can.readthedocs.io/en/stable/

Create a bus instance

```
import can
with can.Bus(interface='socketcan',
                channel='canE',
                bitrate=250000,
                receive_own_messages=True) as bus:
```

Send a CAN message

```
        message = can.Message(arbitration_id=0xC0FFEE,
                                is_extended_id=True,
                                data=[0x11, 0x22, 0x33, 0, 0, 0xA6, 0xB7, 0xC8])
        try:
                bus.send(message)
                print(f”Message sent on {bus.channel_info}”)
        except can.CanError:
                print(“Message NOT sent”)
```

Receiving CAN messages

```
        for msg in bus:
                print(f"{msg.arbitration_id:X}: {msg.data}")
```