gpio.ipCore

Manual

gpio.ipCore: Manual

Copyright © 2015 taskit GmbH

All rights to this documentation and to the product(s) described herein are reserved by taskit GmbH.

This document was written with care, but errors cannot be excluded. Neither the company named above nor the seller assumes legal liability for mistakes, resulting operational errors or the consequences thereof. Trademarks, company names and product names may be protected by law. This document may not be reproduced, edited, copied or distributed in part or in whole without written permission.

This document was generated on 2015-07-30T16:51:18+02:00.

Table of Contents

List of Figures

List of Tables

1. Overview

The gpio.ipcore forms the interface of the GPIO system in an IP network. Along with its function as a gateway between several GPIO Modbus devices and network applications on the basis of HTTP-REST protocols, the ipCore offers a flexible script interface which allows to monitor and control the Modbus hardware connected. Both together turn the device into an independent IO controller on which it is easy to implement processes thanks to simple programmability.

Common scripting languages (Python, NodeJS and Lua) are preinstalled and configured on the ipCore. Communication is possible between different scripts or with external devices, via a lightweight database. Any keys can be exported from the database via HTTP/REST. In this way, simple communication between control script and visualisation is possible independently of any system. Simultaneous access to the Modbus is managed by the integrated Modbus TCP server.

The standard RS485 Modbus can be expanded by up to two USB Modbuses. By using taskit's GPIO modules, three separate RS485 buses (two via the USB gateway) can be managed and controlled by the ipCore. Networking of several gpio.ipCore amongst themselves is possible also.

2. Commissioning

Power is supplied via a 5V USB unit which is connected to micro USB port X14 (between the two DSUBs). The ipCore can provide supply for modules at the RS485 bus. Here, the maximum current that can be supplied via the power supply unit must be taken into account. After deduction of the own consumption of max. 1W@5V, 1800mA will remain in a 2A power supply unit for the IO cards connected.

2.1. RS485

The RS485 bus of the ipCore is located at X13 (DSUB socket) on pins 4 (RS485+) and 6 (RS485-). A gpio.Net IO card can be connected directly by means of a normal serial 1-1 cable. In a similar manner, all other cards can be hung in a row by connecting the DSUB socket to the DSUB plug.

2.2. USB

The two USB host ports X9 can also be used to connect a gpio.Net IO card. This card then serves as a gateway between the USB and all other IO cards connected to it.

2.3. Booting and usage

After switching on the ipCore (power supply is present, push button SW1 has been briefly pressed), the Modbus TCP server, the REST interface and the database start automatically. User-defined scripts can also be executed automatically.

The gpio.ipCore obtains its IP address on its own via DHCP and uses the mDNS protocol (Avahi, Bonjour, Zeroconf) to publish its IP. An ipCore can be accessed on “ipcore.local”. The device responds to command “ping ipcore.local” in a shell (Windows/MAC/Linux). The REST interface can be accessed via URL http://ipcore.local/gpio by using a browser.

To ensure to above method works, Bonjour (from Apple) must be installed in Windows, and Avahi must be installed in Linux. For more configurations of the system and to upload the scripts, the device can be accessed via SSH.

	User	**Password**
REST interface	user	password
SSH	root	taskit

Table 2.1. Default login information

2.4. Useful tools

The use of the following tools is recommended for easy access to the gpio.ipCore in Windows: Putty, curl, winscp. Linux usually brings all necessary tools along already.

2.5. Changing root password

We strongly advise the user to change the default password for root access. In order to do so, login via SSH using the information from Table 2.1, “Default login information”.

Run the command "passwd" to set a new password for root. It asks the user to enter the old and the new password. Afterwards - before any change is done - you need to repeat the new password. All following SSH login attempts are checked against the new password.

This does not change the default password for the REST interface! See Chapter 6, *Restinio* for details about that interface.

3. REST interface

The gpio.ipCore is accessed by HTTP GET, PUT or POST commands. The URL used either directly describes a Modbus resource (http://ipcore.local/gpio/...) or a key in the database (http://ipcore.local/shared/...). Data is transmitted in JSON format.

3.1. Example 1: Access a card at the RS485 Modbus

Let's assume several gpio.net cards at the bus. Address 5 is a gpio.relay card.

Enquiries to the REST interface (GET) either provide a list of continuative suffixes or a simple object containing the result of the enquiry:

```
http://ipcorelocal/gpio/RS485
→  [1, 3, 4, 5, 9]
http://ipcore.local/gpio/RS485/5
→  ["out","in","relay","counter"]
http://ipcore.local/gpio/RS485/5/in/A1
→  {"Value": 1}
```

3.2. Example 2: Access to the database

Queries (GET) about any keys will result in a tuple consisting of the key (string) and its saved value (string):

```
http://ipcore.local/shared/myKey
→  {"key": "myKey", "value": "null"}
→  {"key": "myKey", "value": "data associated with myKey"}
```

The setting (PUT/POST) of keys is performed by writing a value to any (even if non-existent) key. There, the data type of the value is always a string.

```
curl anyauth user user:password data "Test" ipcore.local/shared/newKey
curl anyauth user user:password data "123" ipcore.local/shared/new/test
```

4. Scripting

The behaviour of the gpio.ipCore can be automated through scripts. In the process, the database (Redis), and Modbus and the REST servers form uniform interfaces. Supported scripting languages are Lua, Python and JavaScript. Figure 4.1, “Internal structure” shows the interaction of the components involved.

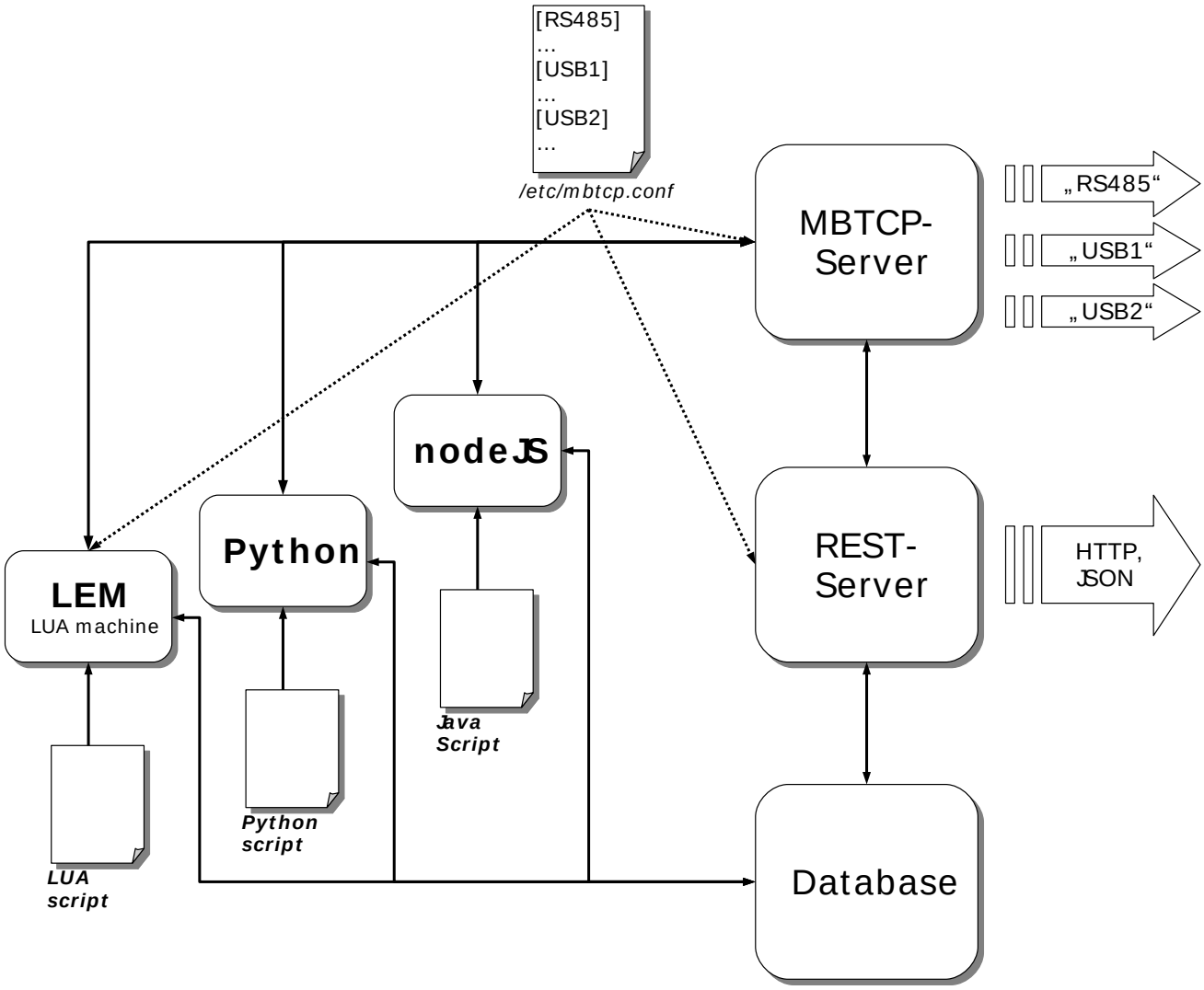

Figure 4.1. Internal structure

The scripts may run in parallel to each other and send enquiries to the Modbus server. The necessary Modbus libraries are preinstalled for the respective languages. Processes that, for instance, check on threshold values or intervene in a regulating manner without external assistance, that is, without additional logic from outside, can be realised completely in the ipCore itself. Nonetheless, the entire hardware connected is also available to the outside via a REST interface. The system gains full flexibility through the connection between script and database and between database and the REST server. In this way, it is possible to transmit user-defined keys whose meaning is defined by a script, to an external front end. Even more complex control instructions that require changes to several Modbus modules or that are to effect state changes to the standard script, can thus be given efficiently with just one HTTP access.

Python, Lua or NodeJS scripts are saved in directory “/var/ipcore/”. It is recommended to create a separate subdirectory for each script.

```
cd /var/ipcore
mkdir myscript
cd myscript
vim myscript.py
chmod +x myscript.py
```

The following scripts are examples of this:

4.1. LUA

The following script searches for a gpio.AI module in all buses available and continuously records measurements issued by the standard output.

```
#! /usr/bin/lem /etc/mbtcp.conf
--[[
1. Look for a gpio.AI module on all busses
2. Configure gpio.AI
3. Get data
]]

-- return card type
function get_card_type(id)
 _, _, ct, _, _ = string.match(id, "^(.*):(.*):(.*):(.*):(.*)$")
 ct = "0x" .. ct
 ct = tonumber(ct)

 if ct == 0x0200 then
  return "AO"
 elseif ct == 0x0100 then
  return "AI"
 elseif (ct >= 0x0010) and (ct <= 0x009F) then
  return "DIO"
 elseif (ct >= 0x00A0) and (ct <= 0x00FF) then
  return "Relais"
 end

 return nil
end

-- return bus and Modbus ID of the first card matching type
function find_card_type(type)
 bus_list = tml.get_busses()

 for _, bus in ipairs(bus_list) do
  for i = 1, 16, 1 do
   ret, id = tml.report_id(bus, i)
   if ret == 0 then
    if get_card_type(id) == type then
     return bus, i
    end
   end
  end
 end
 return nil
end

local bus, mb_id = find_card_type("AI")
tml.write_regs(bus, mb_id, 0x0100, {65535, 2})
print("bus: " .. bus .. "  id: " .. mb_id)
print("\nIN0 ... IN7 in mV")

while true do
 -- read IN0 to IN7
 ret, v = tml.read_input_regs(bus, mb_id, 0, 8)
```

```
  if ret == 0 then
   -- output data as mV
   for i = 1, #v, 1 do
    io.write(string.format("%7.2f  ", v[i]*4096/65535))
   end
   io.write("\r")
   io.flush()
  else
   print("ERROR: " .. ret)
  end

  -- sleep 1 second
  lem.sleep_msec(1000)
 end
```

The Lua interpreter has been expanded by various functions for easy access to the Modbus hardware. Table 4.1, “Extensions to the LUA language on gpio.ipCore” provides information on the functions added by taskit GmbH.

Function definition	Signature	Description
name = lem.get_name()	nil → string	Returns the interpreter's name and version information.
time = lem.get_msec()	nil → number	Returns system timer in milliseconds.
rc, obuf = tml.modbus_command(bus, id, cmd, ibuf)	(string, number, number, table) → (number, table)	A generic Modbus command cmd is send to the server identified by bus. Parameters are stored as numbers in the table ibuf. The Modbus device is addressed using id. An error code rc and output values (e.g. Modbus registers) as table of numbers obuf are returned.
rc, regs = tml.read_holding_regs(bus, id, addr, n)	(string, number, number, number) → (number, table)	Reads n holding registers starting at addr. Returns errorcode rc and a table of registers (numbers) regs.
rc, regs = tml.read_input_regs(bus, id, addr, n)	(string, number, number, number) → (number, table)	Same as tml.read_holding_regs but for input registers.
rc = tml.write_regs(bus, id, addr, regs)	(string, number, number, table) → number	Writes register set regs (table of numbers) starting at addr.
rc, bits = tml.read_coils(bus, id, addr, n)	(string, number, number, number) → (number, table)	Similar to tml.read_holding_regs but for read-/writable bit values. Bits are returned in table bits (numbers). A logic false is represented by 0, anything else equals true.
rc, bits = tml.read_inputs(bus, id, addr, n)	(string, number, number, number) → (number, table)	Same as tml.read_coils but for read-only bit values.
rc = tml.write_coils(bus, id, addr, bits)	(string, number, number, table) → number	Same as tml.write_regs but for bit-wise data. Bits are taken from table bits (numbers). A logic false is represented by 0, anything else equals true.
rc, id_string = tml.report_id(bus, id)	(string, number) → (number, string)	Returns the identification string id_string for the addressed Modbus device (bus, id).

Function definition	Signature	Description
bus_list = tml.get_busses()	nil -> table	Returns a list of available Modbus server connections as table of strings. Those strings are used as bus parameter in all tml.* commands.

Table 4.1. Extensions to the LUA language on gpio.ipCore

4.2. Python

In this example, a digital input and a relay of the gpio.relay card is connected to the internal database. These can then be used via the REST interface on URL “http://ipcore.local/shared/button” and “http://ipcore.local/shared/relay”. The actions standing behind these URLs can, of course, turn out to be far more complex than in this example.

```
#!/usr/bin/env python2.7
import redis
from pymodbus.client.sync import ModbusTcpClient

r = redis.StrictRedis(host = '127.0.0.1', port = 6379, db = 0)

mb = ModbusTcpClient("127.0.0.1", 12346)

oldButtonState = False
oldRelayState = False

while True:
 buttonState = mb.read_discrete_inputs(unit = 8, address = 2).bits[0]
 if oldButtonState != buttonState:
  oldButtonState = buttonState

  if buttonState == True:
   print "Button pressed"
   r.incr('button')

 relayState = r.get('relay')
 if oldRelayState != relayState:
  oldRelayState = relayState

  if relayState == "1":
   print "Turning relay on"
   mb.write_coil(unit = 8, address = 4, value = 1)
  else:
   print "Turning relay off"
   mb.write_coil(unit = 8, address = 4, value = 0)
```

4.3. Automated starting of scripts

The two previous scripts both contain in their first line a construct in the form of “#! command parameter”. This helps the system to start the appropriate interpreter for the script. If the data has been generated as described in the last paragraph, it can be easily started and stopped by the system automatically. The following steps are required to have “myscript.py” managed by the system:

```
mkdir /service/myscript
cd /service/myscript
ln s /var/ipcore/myscript/myscript.py run
```

The script is now executed promptly and after each start of the system. Should it crash for any reason, it will be restarted. Here, there is a pause of one second to prevent a system from being blocked due to an erroneous script. The script can be stopped via “svc –d / service/myscript” and be restarted via “svc –u /service/myscript”. If it is meant to be removed permanently, the above directory can simply be deleted (“rm –rf /service/myscript”) without touching on the original (“/var/ipcore/myscript/myscript.py”).

Lua scripts are executed by interpreter lem. This is a Lua interpreter already offering Modbus support and some auxiliary functions. The configuration file of the Modbus server is transmitted to lem as a parameter, simplifying to select the appropriate port and IP address of an individual bus in your script.

5. Modbus server (MBTCP)

For each of the ipCore's serial ports - including the two USB hosts - an instance of the Modbus TCP server (MBTCP) can be launched. The server manages access to the devices that are attached to its related serial port. It configures the hardware, queques requests and finally submits the results to their originators.

MBTCP provides a standard Modbus TCP and an ASCII connection for debugging and simple programs that do not need the Modbus TCP stack. The user is free to use either.

The configuration file is located in "/etc/mbtcp.conf". It contains all the vital information for the MBTCP servers and clients. By default, there are three sections: RS485, USB1 and USB2. Parameters used for LEM and the Modbus servers are stored as simple key-value pairs with an equals sign as delimiter between key and value. A sections begins with its name in square brackets and ends with the start of the next section or the end of file. The following except makes this more clear.

```
[USB1]
dev=/dev/USB1
mode=ascii
tcp_port=12346
ascii_port=33334
host=localhost
timeout=100
[RS485]
...
```

Table 5.1, “MBTCP configuration” describes the config items, their use and default values. Some items from this configuration are also used by the LUA machine (LEM).

Item	MBTCP	LEM	Default value	Description
dev	yes	no		The device file used as Modbus connection
baud_rate	yes	no	9600	Selects connection speed
data_bits	yes	no	8	Bits used per character
parity	yes	no	even	Method user for parity checking
stop_bits	yes	no	1	Number of stop bits per character
media	yes	no	rs485	Selects between RS232 and RS485 handling
mode	yes	no	rtu	Modbus mode on serial line (rtu or ascii)
timeout	yes	no	1000	Max. time in ms used for packet reception
host	no	yes		Host to connect to
bind	yes	no	0.0.0.0	Interface for listening
max_conn	yes	no	16	Max. number of open connections handled by server
tcp_port	yes	no		Port used for Modbus TCP
ascii_port	yes	yes		Port used for Modbus ASCII protocol over TCP

Table 5.1. MBTCP configuration

6. Restinio

Restinio is an optional service that provides the REST interfaces to Modbus and the Redis database. It is launched by default and can be stopped or restarted just like user defined scripts using the svc command. The link to its run script resides in "/service/restinio".

During normal operation, Restinio does background scans of the connected Modbus devices. Thus, the REST interface offers a plug'n'play mechanism. It can be configured for each Modbus connection in Restinio's config file "/var/ipcore/restinio/restinio.conf". This file also holds the user/password combination and basic Redis setup.

Since user scripts can act as HTTP servers, you might consider to disable Restinio completely. In that case, just rename the link to the run script.

```
svc -d /service/restinio
mv /service/restinio/run /service/restinio/run.disabled
```

7. Technical data

Specification	Value	Unit
Power supply	5	V
Power consumption	<700	mW
Operating temperature	-30 .. +85 [a] [b]	°C

[a] The buzzer is limited to -20 .. +70 °C but unused.
[b] SD card socket is limited to -25 .. +85 °C.

Table 7.1. Electrical charachteristics

7.1. Modbus port

MBTCP system uses this port as configured in "/etc/mbtcp.conf" section [RS485].

Pin	Description
1	unused
2	unused[a]
3	unused[b]
4	RS485+
5	GND
6	RS485-
7	unused
8	unused
9	+5V[c]

[a] Can be RXD of RS232 "/dev/ttyS3".
[b] Can be TXD of RS232 "/dev/ttyS3".
[c] This pin can power connected gpio.net modules.

Table 7.2. RS485 (DSUB female connector X13)

7.2. Debug console

This port runs a debug/emergency shell at 115200 baud 8N1.

Pin	Description
1	unused
2	RXD
3	TXD
4	unused
5	GND
6	unused
7	unused
8	unused
9	unused

Table 7.3. RS232 (DSUB male connector X6)