Modbus-TCP on embedded systems

Mathieu ALLARD - Matthieu ROY

M1MNE 2014

Contents

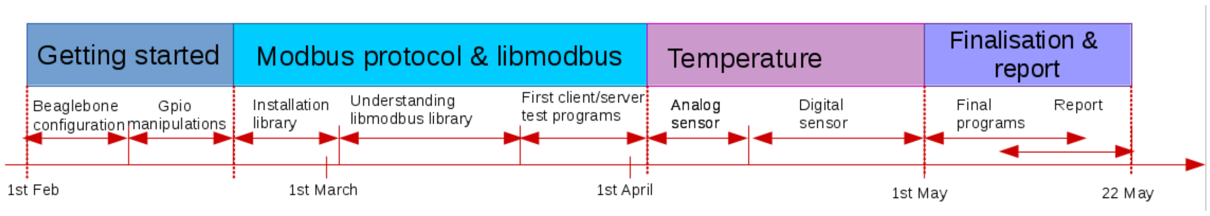
Chapter 1

Introduction

This project's goal is to communicate with an embedded system with the modbus protocol over a TCP/IP network, in order to get temperature acquisition, and then display it as a graphic.

The modbus protocol is part of automatons communication standard, we'll use a beagleboneblack as an automaton. We had the choice between C or python language to use the modbus protocol layer. We chose C because we were a bit more familiar with it, furthermore, C compiles on a lower level compared to python interpretation, which is more interesting for embedded systems.

The project is divided in four major parts, presented in the diagram below :



Chapter 2

The Beagleboneblack

2.1 Why the beagleboneblack

The beagleboneblack is a small development board, unlike a microcontroler, it comes with a full featured operating system running on an arm v7 processor. So we'll have to program on a high abstraction level to use it, which is necessary to use the modbus protocol over a TCP/IP network.

There are a few differents development boards running on arm chip, but the beaglebone features an impressive list of interfaces : i2c bus, can bus, SPI bus, analog inputs, GPIO pins. . . So the beaglebone is clearly oriented toward external interfacing, which is really interesting in an embedded system. Plus, unlike the raspberry pi, it works directly out of the box with Angström Linux installed on its eMMC, there's no need for an extra SD card. In top of that, the beaglebone is really affordable. (about 45 euros)

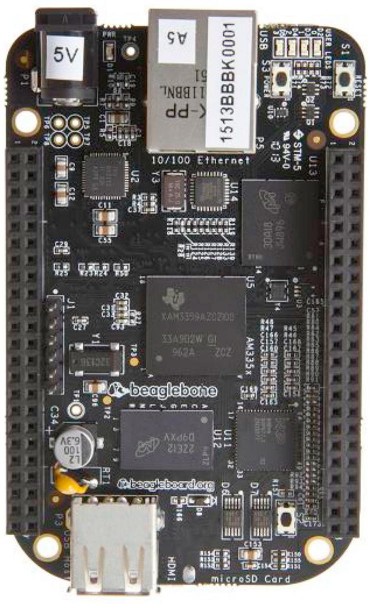


Figure 2.1: top view of the beagleboneblack

So the beaglebone fits very well for our project as an embedded system. We'll use it without a mouse, screen, or keyboard, just by connecting through it on our terminal emulator, without a graphical user interface.

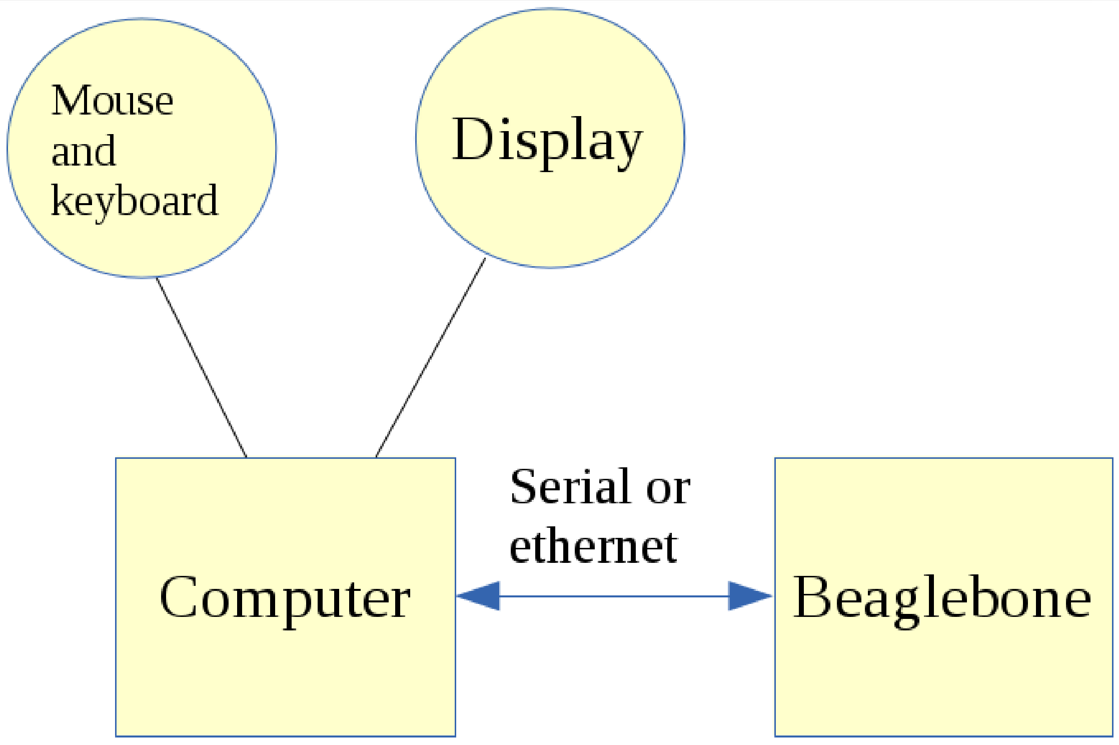


Figure 2.2: How we interfaced the beagleboneblack

2.2 Connecting to the device

2.2.1 Serial Port

The beaglebone features serial ports that can be accessed as a terminal, it's handy as a debug mode in the case of networks problems, so it's the first thing to setup. We connected a 3,3V FTDI TTL-232 adapter on these to be able to connect to the beagle via an usb port of our host computer.

There are several ways to access a terminal on linux, kermit works well with the apropriate config file, but screen is natively installed on most systems and easier to use. If the FTDI usb is plugged "ttyUSB0" should appear in /dev/

```
    $ > ls /dev | grep USB
```

We can now connect with :

```
        # screen /dev/ttyUSB0 115200
```

where 115200 is the baudrate.

2.2.2 Ethernet over USB

I couldn't use this feature with debian 7.0 and a 3.2 linux kernel. It was solved by updating to a newer kernel. Slackware 14.1 with 3.10 kernel handled it.
If the beaglebone is connected to a computer via usb, its filesystem can be mounted and then the network interface will see it. Its IP default adress is 192.168.7.2, so in order to connect to it we have to set up our IP adress in the same network, let's say 192.168.7.1, with our network manager of choice (typically wicd or NetworkManager) and check if the connexion is up. I won't detail anymore as we mainly used the ethernet port.

2.2.3 Network configuration

The best way to access the beaglebone is by using the "real" ethernet port to do the transmission : the beaglebone doesn't need to be wired by usb to the computer anymore. We could access it virtually from anywhere as long it's connected to the internet network. This is a really nice feature for an emebedded system.

For our project we'll still use it directly wired to the host computer, on a local network, so we wanted to setup a static IP adress to the beaglebone to access it easily.

Setting it up

The network settings on Angstrom Linux aren't very easy to handle, we have to use Connman (Connection Manager), that provides many scripts to change the parameters.
So we followed a tutorial made by Derek Molloy, available on his website :
http://derekmolloy.ie/set-ip-address-to-be-static-on-the-beaglebone-black/
We'll focus here on the main steps.
The scripts are located in /usr/lib/connman/test First we need to know the name of our ethernet device:

```
    root@beaglebone:~# ls -la /var/lib/connman/
    total 28
    drwxr-xr-x  3 root root 4096 Jan  1 01:36 .
    drwxr-xr-x 17 root root 4096 Jan  1 01:00 ..
    drwx------  2 root root 4096 Jan  1 01:45 ethernet_9059af5cabcd_cable
    -rw-------  1 root root   68 Jan  1 01:36 settings
    -rw-------  1 root root   68 Jan  1 01:07 settings.4QQLXP
    -rw-------  1 root root   68 Jan  1 01:06 settings.MG4VXP
    -rw-------  1 root root   68 Jan  1 01:00 settings.S1ORXP
```

and then run the script to set it up as we want :

```
./set-ipv4-method ethernet_9059af5cabcd_cable manual 192.168.1.100
255.255.255.0 192.168.1.1
```

We can now reboot the beaglebone and try to connect with 192.168.1.101 on TEST PROGRAMME ANNEXE the host computer to 192.168.1.100 :

```
    $ > ping 192.168.1.100
    PING 192.168.1.100 (192.168.1.100) 56(84) bytes of data.
    64 bytes from 192.168.1.100: icmp_seq=1 ttl=64 time=0.261 ms
    64 bytes from 192.168.1.100: icmp_seq=2 ttl=64 time=0.269 ms
    64 bytes from 192.168.1.100: icmp_seq=3 ttl=64 time=0.266 ms
    64 bytes from 192.168.1.100: icmp_seq=4 ttl=64 time=0.271 ms
```

If it's successfull, ssh can be used to access to the device, which is far more practical than a serial transmission.

```
    $ > ssh rott@192.168.1.100
```

also, scp can be used to send or retreive files from the beaglebone :

```
   $ > scp /path/to/file root@192.168.1.100:/path/to/file #sending
   $ > scp root@192.168.1.100:/path/to/file /path/to/file #retrieving
```

2.3 tests and tips

Now that everything is set up, we can begin to work on the beaglebone :

2.3.1 Text editors

We only used the console on the beagleboneblack, so we hadn't access to graphical tools like gedit. We had to use vim or nano instead, they aren't difficult to use but require basic knoledge of some key combination to be used correctly.

2.3.2 first compilation

With our text editor of choice, we create a basic hello world C code : hello.c

```
#include <stdio.h>
void main(void){ printf(''hello world\n''); }
```

Compile it with gcc, and execute it :

```
root@beaglebone:~# gcc -o hello hello.c
root@beaglebone:~# ./hello
hello world
```

2.3.3 adding new users

We created users for our personal use of the device

```
    useradd -D username
```

the -D option stands for Default settings, so the user is already in a few groups. Then log with the user and add a password with passwd :

```
mathieu@beaglebone:~$ passwd
Changing password for mathieu.
(current) UNIX password: Enter new UNIX password: Retype new UNIX password:
passwd: password updated successfully
```

The "su" command allows to log back in root or execute some command with root.

2.3.4 giving a name to the IP address

In order to quickly connect to the beaglebone, a name can be associated with its IP address, as it's static. This can be easily done on the host computer by adding a line in the file /etc/hosts.

```
#> echo ''192.168.1.100 beaglebone'' >> /etc/hosts
```

Now the beagle bone can be accessed by typing

```
ssh root@beaglebone
```

2.3.5 setting the date properly

We experienced some problems at compilation time due to the time settings : the beaglebone doesn't have its own clock, so the time is reset to jan 1 2000 00:00 each time it's powered off. Make doesn't like to compile a file if it already compile it in the future, so we have to set the date each time we use the beaglebone

```
root@beaglebone:~/libmodbus-3.0.5# date -s ''thu may 22 2014 07:59''
Thu May 22 07:59:00 CEST 2014
```

2.4 using the GPIO ports

2.4.1 Manually

GPIO manipulation is done with files located in /sys/class/gpio

For example to initialize GPIO60 (on PIN12 of P9)

```
root@beaglebone:~# cd /sys/class/gpio
root@beaglebone:/sys/class/gpio# ls
export  gpiochip0  gpiochip32  gpiochip64  gpiochip96  unexport
root@beaglebone:/sys/class/gpio# echo 60 > export
root@beaglebone:/sys/class/gpio# ls
export  gpio60  gpiochip0  gpiochip32  gpiochip64  gpiochip96  unexport
```

Setting a gpio to output 1 :

```
root@beaglebone:/sys/class/gpio# cd gpio60
root@beaglebone:/sys/class/gpio/gpio60# ls
active_low  direction  edge  power  subsystem  uevent  value
root@beaglebone:/sys/class/gpio/gpio60# echo out > direction
root@beaglebone:/sys/class/gpio/gpio60# echo 1 > value
```

Confirmation the previous steps worked can be given by just measuring the voltage on the selected PIN, it should be 3.3V.

2.4.2 With iolib

Opening and editing files to use the gpio in C programs isn't very convenient, so we looked for a library implementing this and found iolib.h developped by

We sended the archive to the beaglebone, unarchived it and ran "make" to compile it. Examples are provided, helping to understand how it works. It's easy to use, here are the basic functions :

```
ioinit();          //initialize the library
iolib_setdir(9,12, DIR_OUT);    // setting an output port
iolib_setdir(9,13, DIR_IN);     // setting an input port
is_high(9,13);   // returns 1 if the input is at 1
is_low(9,13);    // returns 1 if the input is at 0
pin_low(9,12);   // sets the output to 0
pin_high(9,12);  // sets the output to 1
```

We wrote a simple program that take an input with a switch and lights a LED on output depending on the switch state.

testio.c source code A.1

NB : we later found that this library was forked, extending to pwms and analog inputs handling

compilation

To compile a file with this library it has to be declared at the begining with

```
#include ''/path/to/iolib.h''
```

And then gcc must be told to look for it :

```
gcc -o program program.c /path/to/iolib.c
```

In order to make the compilation easier, we can use the make tool by creating a Makefile with the gcc command in it :

```
program: program.c
	gcc -o program program.c /path/to/iolib.c
```

This allows to compile our file by just typing "make" instead of remembering a long gcc command.

2.5 About the operating system

It's nice to have a system running right out of the box, but we found Angström linux not that easy to handle correctly once we wanted to do some setup. For example with the network, compared to debian with wicd it's ridiculously complicated. Connman is lightweight so it fits well for embedded systems, but Angström runs gnome2 with an Xsession by default so the need of a really small and lightweight network manager is quite hard to understand here. And Angström Linux boots up with systemD which is fast but complicated to setup too, compared to an old sysV, or a BSD style init.

It could be interesting to install our customized OS on the beaglebone, with buildroot but we didn't have time to find a patch to make it work properly.

Chapter 3

Modbus and libmodbus

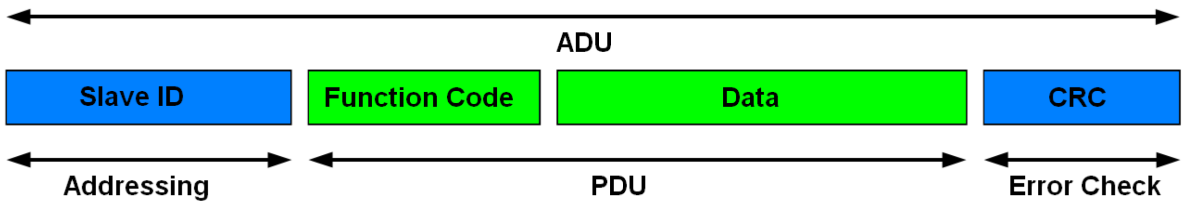
3.1 Modbus protocol

3.1.1 Introduction

Modbus protocol is a serial communication protocol published by Modicon in 1979. Basicaly made for the RS232C bus and later on RS422 and RS485, this protocol was popular for his simplicity. Other reasons are that modbus protocol is openly published and royalty-free and developped with industrial application in mind. There are 24 basics modbus functions.

3.1.2 Serial modbus protocol

A modbus Protodol Data Unit (PDU) is composed by function code and the data frame. Over serial mode, historically the first mode, the modbus serial line PDU is composed by an adress field (slave adress), the modbus PDU and a Cyclic Redundancy Check (CRC) frame.

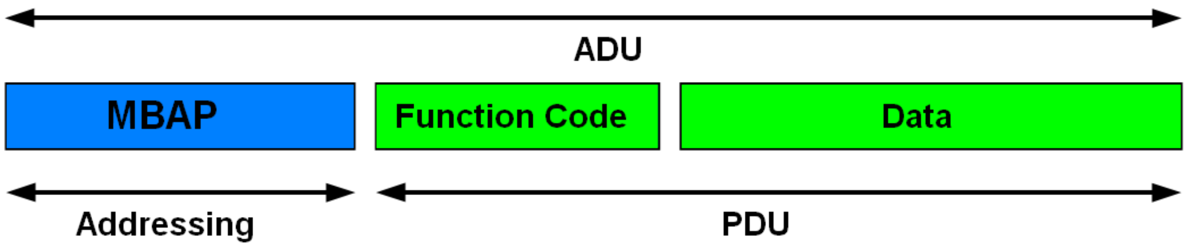


3.1.3 TCP/IP

TCP/IP is main transport protocol used on the internet, and become a standard for factory networking. Principle of modbus TCP/IP is easy, you take the Modbus PDU and just add a header TCP. Moreover it requires just standard PC ethernet card to communicate with a new implemented device. The TCP (Transmission Control Protocol) define transport layer. It divides data in segments and add a TCP header. A TCP segment consists in a segment TCP header and a data frame. The Internet Protocol (IP) is the principal communication protocol in the Internet protocol. It delivers segments made by the TCP layer from the source host to a destination host throup IP adresses written in the packet headers. The result is that the therms master/slave is remplaced by Client/server in modbus. Indeed the physical and data links layers are not specified and now Modbus is an IP bus. There are four steps during a communication between the client and the server:

- Master/client send a request to the slave/server
- The slave/server send an indication that the request was received
- The slave/server send the response to the request
- The Master/client send the confirmation that the response was received

3.1.4 Construction of a modbus TCP data packet



A modbus TCP ADU (Application Data Unit) is formed and composed by two parts : The Modbus Application Protocol header (MBAP header) and the Modbus protocol data unit.

The MBAP header is seven bytes long. Details are describes in the following table:

	Transaction identifier	Protocol Identifier	Length	Unit Identifier
Length	2 bytes	2 bytes	2 bytes	1 byte

The Transaction identifier field is used for transaction pairing when multiple messages are sent along the same TCP/IP connection by a client. The protocol identifier is different to zero when extensions of Modbus service are used (when new functions developped by user/programmer are used). There is a two bytes count of the remaining fields and the last byte is used to identify a remote server located on a non TCP/IP network.

3.2 libmodbus, C

3.2.1 installation

In the host computer, libmodbus may be available through the package manager. If it isn't we would have to compile it from source, as we did on the beaglebone.

After downloading the archive and uploading it to the beaglebone, we unarchived it and ran ./configure, then make install

```
root@beaglebone:~/libmodbus-3.0.5# ./configure && make install
```

I didn't print all the lines it outputs during the compilation as it isn't relevant. It also compiles all the test programs.

3.2.2 First Modbus tests

To verify if the library is functional, we tried the test programs. By just editing one line with the IP address we used and recompiling all the programs with the make utility. line 61 in unit-test-server.c and line 66 in unit-test-client.c change it to :

```
        ctx = modbus_new_tcp(``192.168.1.100'', 502);
```

The client is our host computer and the server on the beaglebone This test program performs several tests and output the results on the client side.

```
$ > ./unit-test-client
Connecting to 192.168.1.100
** UNIT TESTING **

TEST WRITE/READ:
[00][01][00][00][00][06][FF][05][00][13][FF][00]
Waiting for a confirmation\ldots
<00><01><00><00><00><06><FF><05><00><13><FF><00>
1/2 modbus_write_bit: OK
[00][02][00][00][00][06][FF][01][00][13][00][01]

#performs various tests
#outputs some results

ALL TESTS PASS WITH SUCCESS.
```

On the server side, it verboses the modbus frames.

```
t@beaglebone:~/libmodbus-3.0.5/tests# ./unit-test-server The client connection
from 192.168.1.101 is accepted
Waiting for a indication..
<00><01><00><00><00><06><FF><05><00><13><FF><00>
[00][01][00][00][00][06][FF][05][00][13][FF][00]
Waiting for a indication..
<00><02><00><00><00><06><FF><01><00><13><00><01>
[00][02][00][00][00][04][FF][01][01][01]
Waiting for a indication..

#prints various frames

ERROR Connection reset by peer: read
```

3.2.3 first program

We worked with the sample codes as a base for our program. To understand the different functions we referred to the man pages of the libmodbus documentation.

Client

First we created a pointer to a modbus_ t structure defined in "modbus.h".

```
modbus_t *ctx
```

Then we created the libmodbus context : if succesfull this function returns a pointer to a modbus_ t structure. The address is the server's IP address (192.168.1.100 in our case)and the port is set to 502 wich is reserved for the modbus protocol.

```
ctx = modbus_new_tcp(address ,port)
```

To establish the connection to our modbus server, we used:

```
modbus_connect(ctx)
```

Once the connection is established, the server's registers and coils can be accessed by various functions, here are some :

```
modbus_write_register(mb,reg number, value)
```

Corresponds to modbus function 06 : preset single register.

```
modbus_read_registers(mb,0,8, tab_reg)
```

Corresponds to modbus function 03 : read holding registers.

```
modbus_write_bit(mb,reg number, value)
```

Corresponds to modbus function 05 : force single coil.

```
modbus_read_bits(mb,0,8, dest)
```

Corresponds to modbus function 01 : read coil status.

The following function closes the modbus connection :

```
modbus_close(ctx)
```

And finally, to free the allocated modbus_ t structure :

```
modbus_free(ctx)
```

Server

As in the client program, a modbus_ t structure is defined:

```
modbus_t *ctx
```

On the server(slave) side, memory has to be allocated to the registers and coils.

```
modbus_mapping_t *mb_mapping
mb_mapping = modbus_mapping_new(bits, input bits, registers, input registers)
```

A context has to be initialized, with the same address and port as the client.

```
ctx = modbus_new_tcp( adress, port)
```

This is an optional function, helpful by displaying the bytes of the Modbus messages.

```
modbus_set_debug(ctx, TRUE)
```

A socket has to be created from the context and listened to.

```
socket = modbus_tcp_listen(ctx,1)
```

And a new connection can then be accepted.

```
modbus_tcp_accept(ctx, &socket)
```

An indication request can be received from the socket using : (sent by the client/master)

```
modbus_receive(ctx, query)
```

And a response to the said request is done with

```
modbus_reply(ctx, query, rc, mb_mapping)
```

It writes or read the registers/coils in the modbus mapping depending on the client(master) indication.

After an exchange with the client, the socket has to be closed prior to open another connection.

```
close(socket)
```

Finally, the memory allocated and the context can be freed.

```
modbus_mapping_free(mb_mapping)
modbus_free(ctx)
```

3.2.4 compilation

programs using the libmodbus are meant to be compiled using pkg-config to link the library. There's an example in the README file.

```
    gcc program.c -o program `pkg-config --libs --cflags libmodbus`
```

pkg-config didn't find the libmodbus.so files, in the /usr/local directory, it only searcg in /usr/lib. So we created symbolic links from /usr/share/local/lib to /usr/lib

```
    ls -s /usr/local/lib/libmodbus.so /usr/lib/
    ln -s/usr/local/lib/libmodbus.so.5 /usr/lib/
```

But it should also be able to work when linked like we did with the iolib.

We then created a Makefile to store the gcc command like we did with iolib

3.2.5 tests with our programs

Testclient.c and testserver.c respectively on the computer and the beaglebone.

Using the functions explained previously, we wrote some simple tests programs :

client

See the code for testclient.c A.2

The client writes 1 in the coil 3 and 5 in the register number 3, read the coil number 3 and 8 registers from number 0 to 7, and outputs the results.

```
        $ > ./testclient     coil[3] is at 1
        reg[0]=0 (0x0)
        reg[1]=0 (0x0)
        reg[2]=0 (0x0)
        reg[3]=5 (0x5)
        reg[4]=0 (0x0)
        reg[5]=0 (0x0)
        reg[6]=0 (0x0)
        reg[7]=0 (0x0)
```

server

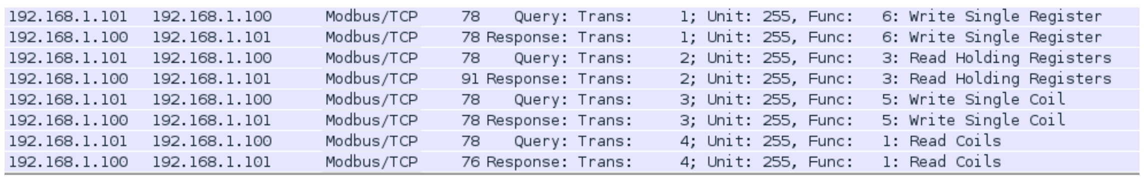
See source code of testserver.c A.3

The server only waits for the client indication, then writes informations on registers or coils and send response to the client.

```
root@beaglebone:/home/matthieu# ./testserver
The client connection from 192.168.1.101 is accepted
Waiting for a indication..
<00><01><00><00><00><06><FF><06><00><03><00><05>
[00][01][00][00][00][06][FF][06][00][03][00][05]
Waiting for a indication..
<00><02><00><00><00><06><FF><03><00><00><00><08>
[00][02][00][00][00][13][FF][03][10][00][00][00][00][00][00][00][05][00][00][00][00][00]
Waiting for a indication..
<00><03><00><00><00><06><FF><05><00><03><FF><00>
[00][03][00][00][00][06][FF][05][00][03][FF][00]
Waiting for a indication..
<00><04><00><00><00><06><FF><01><00><00><00><08>
[00][04][00][00][00][04][FF][01][01][08]
Waiting for a indication..
ERROR Connection reset by peer: read
Quit the loop: Connection reset by peer
byte=-1003847680
```

TCP packets analizer

Wireshark packet analizer allows us to see what transits through the network, we listend to the eth0 interface, and got the modbus-TCP expression to only see the modbus frames. Here's the output :



```
192.168.1.101   192.168.1.100   Modbus/TCP   78    Query: Trans:   1; Unit: 255, Func:   6: Write Single Register
192.168.1.100   192.168.1.101   Modbus/TCP   78 Response: Trans:   1; Unit: 255, Func:   6: Write Single Register
192.168.1.101   192.168.1.100   Modbus/TCP   78    Query: Trans:   2; Unit: 255, Func:   3: Read Holding Registers
192.168.1.100   192.168.1.101   Modbus/TCP   91 Response: Trans:   2; Unit: 255, Func:   3: Read Holding Registers
192.168.1.101   192.168.1.100   Modbus/TCP   78    Query: Trans:   3; Unit: 255, Func:   5: Write Single Coil
192.168.1.100   192.168.1.101   Modbus/TCP   78 Response: Trans:   3; Unit: 255, Func:   5: Write Single Coil
192.168.1.101   192.168.1.100   Modbus/TCP   78    Query: Trans:   4; Unit: 255, Func:   1: Read Coils
192.168.1.100   192.168.1.101   Modbus/TCP   76 Response: Trans:   4; Unit: 255, Func:   1: Read Coils
```

Combining with iolib

In the end we added a condition in the server program launch our iotest program with switch and LED if the coil number 3 is at 1. See source code for serverled.c A.4

Chapter 4

acquiring temperature

We wanted to do temperature recordings. Our first idea was to use an analog input with a simple sensor outputing a voltage proportional to the temperature. But it wouldn't be very accurate, therefore we looked for another solution. Using I2C sensor seemed like a great idea and an interesting implementation to our project.

4.1 analog input

We just tried to do some voltage acquisitions with the beaglebone ADC, same as the gpio, it can be done through some file.

Simple circuit with potentiometer wired between vdd and gnd.

We wanted to read the value in the file
/sys/bus/iio/devices/iio:device0/in_ voltageX_ raw

We did it with a simple program with fopen A.5 As we didn't followed this way for our project, we didn't reconvert it to its voltage value.

NB : we later discovered that the bbbio library can handle the analog inputs

4.2 I2C sensor

We used a Texas instrument TMP102 wich is a low power digital temperature sensor. We have chosen this sensor for several reasons:

Port I2C We looked for a sensor that can easily communicate with the beaglebone. Documentation about I2C interfacing is relatively abundant and the therefore we looked for another solution and an I2C sensor came as an interestingTMP102 features two-wire interface compatibility. Moreover Angström Linux' kernel possess a native TMP102 module. Thereafter we realised that this module is useless for our application.

Values converted As it is a digital sensor it features analog-digital converter ($\Sigma\Delta$ module). We have directly temperature's value in 2 data bytes contrary to previous temperature sensor.

The resolution The resolution for the Temp ADC in Internal Temperature mode is 0.0625Â°C/count and the device is specified for operation over a temperature range of $40\deg C$ to $+125\deg C$. Temperature is coded on 12 bits. Negative numbers are represented in binary twos complement format. We are sure to have a good temperature resolution with this device.

4.3 I2C protocol

4.4 Principle

The Inter-IC-Communication was made for release the link between integrated circuits of the same connector board. The bus is bidirectionnal and half-duplex and the data transfert rate is fixed about 100kbps.

4.4.1 Material

This bus is composed by 3 wires. Two wires for signals exchanges and a third for the ground reference. This imply a serial transmission mode.
The two signals wires :

- Serial Data (SDA) : For transmission data
- Serial Clock (SCL) : For transport of the master clock

A fourth whire is used for emitter's polarisation.

4.4.2 Frame structure

A classical I2C frame features 6 parts:

- One start bit
- Seven adress bit
- One Read/Write bit (read =>'1', write => '0')
- One bit acknoledge
- N data bytes (8 data bit + 1 ack)
- One stop bit

A data symbol must be stable during a clock high state, it must be changed during a clock low state. It is different for a start bit which is a transition High-Low during a clock high state contrary to a stop bit which is a transition Low-High.

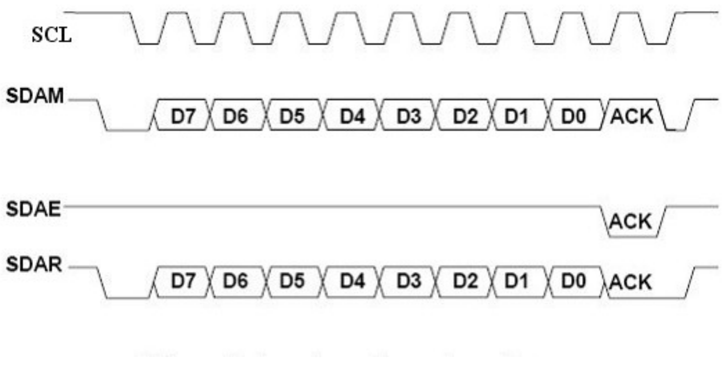


Figure 4.1: i2c frame
SCL : Clock set by master
SDAM: SDA levels set by master
SDAE: SDA levels set by slave
SDAR: resultant SDA levels

4.5 Calibration

4.5.1 Emitter's polarisation

The emitter's output structure is an open-collector type. The link between electrical level and the logic state is founded compared to V_{GND}.

$$V_{GND} < V < V_{IL} =>' 0'$$

$$V_{HI} < V < V_{DD} =>' 1'$$

We impose that : $V_{IL} = 0.3V_{DD}$ and $V_{HI} = 0.7V_{DD}$ for a current of 3mA. In our case, polarisation voltage V_{DD} is 3.3V. For sizing external pull-up resistors we solve the equation:

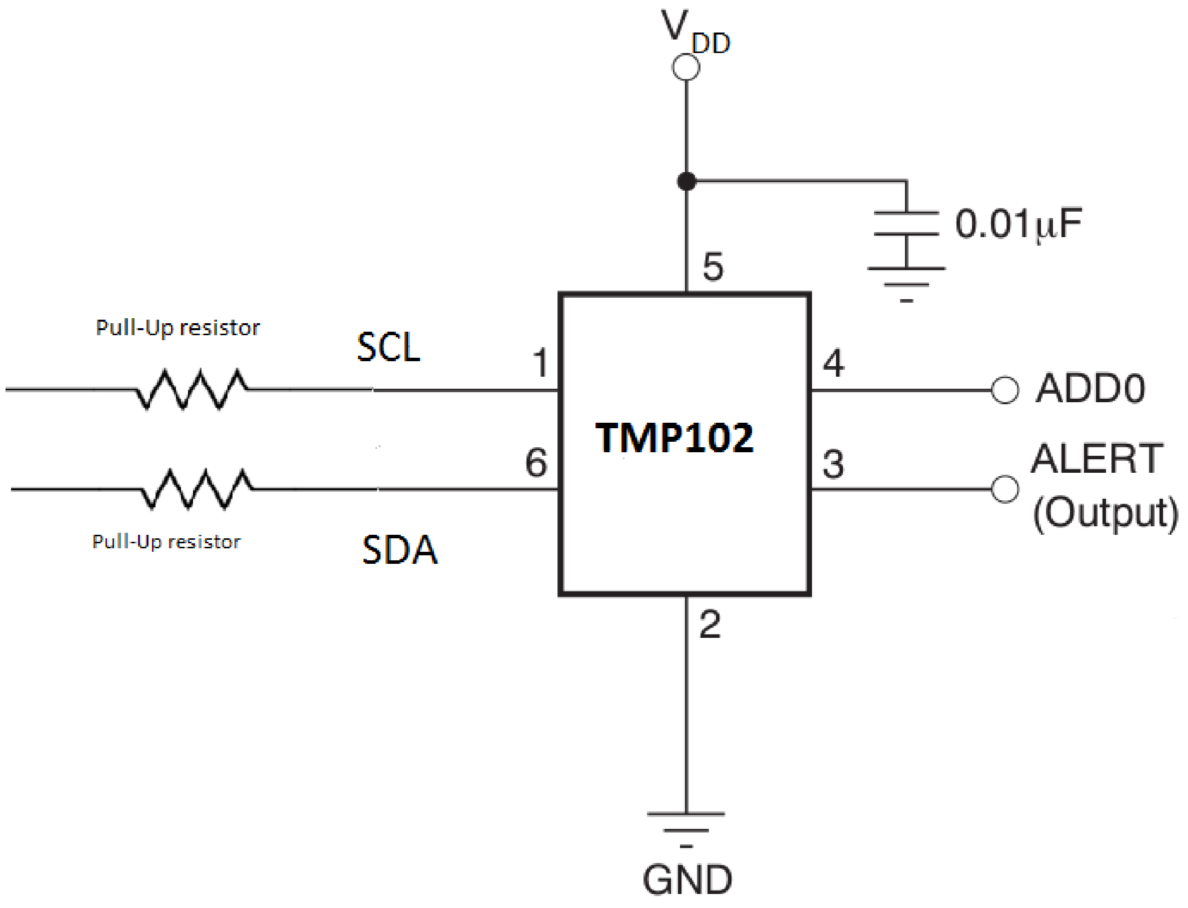
$$R_P = \frac{(V_{DD} - V_{IL})}{3.10^{-3}} = \frac{0.7.V_{DD}}{3.10^{-3}}$$

For $V_{DD} = 3.3V$:

$$R_P = 770\Omega$$

4.5.2 Capacitance

The bit rate must be 100kbps so we must impose arise time less than $1\mu s$. The rise time of a first order electrical system is function of $2.2R\_PC$. It is recommanded to use a capacitance of $0.1nF$ in the tmp102's datasheet. Final circuit is :



4.6 Test with the terminal

In this part we talk how to interface with an I2C temperature sensor and with a I2C sensor in general. We have used the temperature sensor tmp102.

We have linked sensor's pins with beaglebone i2c pins :

- SDA : P9, 19
- SCL : P9, 20
- gnd : P9, 1
- V_{DD} : P9, 3

After connection the sensor to beaglebone's i2c ports,we should detect the device. In the terminal, we use the command : "lsmod" "mode probe tmp102". To detect devices : we use the command i2cdetect in order to list the i2C adapters:

```
i2cdetect -l
```

The "-r" option specifies the use of SMBus "read byte" commands for probing, "1" is the number of i2c adapter we want use

```
i2cdetect -r 1
```

We obtain:

```
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- 48 -- -- -- -- -- -- --
50: -- -- -- -- UU UU UU UU -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

Wich corespond to sensor adress (1001000).
So we go in the directory of the i2c adapter we want use (i2c-1):

```
root@beaglebone:/sys/class/i2c-adapter/i2c-1# ls
1-0054  1-0056  delete_device  i2c-dev  new_device  subsystem
1-0055  1-0057  device         name     power       uevent
```

We can see the reserved adresses "UU".
In this directory we should define a new device with the hexadecimal adress 48:

```
/sys/class/i2c-adapter/i2c-1# echo tmp102 0x48 > new_device
```

and you can see the new class 1-0048:

```
root@beaglebone:/sys/class/i2c-adapter/i2c-1# ls
1-0048  1-0055  1-0057         device   name        power      uevent
1-0054  1-0056  delete_device  i2c-dev  new_device  subsystem
```

To verify if tmp102 is mounted we use :

```
root@beaglebone:/sys/class/i2c-adapter/i2c-1# dmesg | grep tmp102
[    0.029368] devtmpfs: initialized
[    2.816245] devtmpfs: mounted
[ 2962.148742] tmp102 1-0048: initialized
[ 2962.151751] i2c i2c-1: new_device: Instantiated device tmp102 at 0x48
```

```
root@beaglebone:/sys/bus/i2c/drivers/tmp102/1-0048# ls
driver  modalias  power      temp1_input  temp1_max_hyst
hwmon   name      subsystem  temp1_max    uevent
```

To read temporary register of i2c :

```
root@beaglebone:/sys/bus/i2c/drivers/tmp102/1-0048# echo scale=2 \;
$(cat /sys/bus/i2c/drivers/tmp102/1-0048/temp1_input) / 1000  |  bc
25.31
```

Which is the value of temperature.

4.7 C program

4.7.1 Functions used

We translated previous protocol in a C program using two C libraries: fcntl.h and i2c-dev.h. The library "i2c-dev.h" gives access to the i2c protocols and "fcntl.h" define some requests and arguments for use by the functions "fcntl()" and "open()".

We present functions i2c-dev used for our C program:

```
open("/dev/i2c-1", O_RDWR)
```

Where O_ RDWR is the file access mode "Open for reading and writing" include in the library "fcntl.h" and "i2c-1" is the device file to port i2c number 1. We have opened i2c port number 1 in mode reading and writing.

We must now define with what device we want communicate:

```
ioctl(tmp102, I2C_SLAVE, address)
```

Where "tmp102" is the name of the device, I2C_ SLAVE is defined in i2c-dev.h and categorize the device as a slave and address is his address (72 in this case).

For send a read request we use this function "read".

```
read(tmp102, registre_lecture,2)
```

Where "registre_ lecture" is the register where the temperatures data will be stored. '2' is to read 2 uint8 because data frames are divided in 2 octets. Each data octet will be stored in a different register.

4.7.2 Difficulties

We have faced to some difficulties to convert data frame received in a real variable for two reasons: the resolution of tmp102 is 12 bits but the device send a 16 bits data frame and it does twos compliment but has the negative bit in the wrong spot.

To convert the two registers in an integer we have write :

```
temperature = ((registre_lecture[0]) << 8) | (registre_lecture[1]);
temperature >>= 4;
```

And to solve the negative bit problem, we test for it and correct if needed:

```
if (temperature & (1 << 11))
temperature |= 0xF800;
```

see progrecup.c A.6

We will integrate this program to the final program server at the end of the project.

Chapter 5

Results

5.1 Realization

A server and a client C program have been written and compiled using all of the above to recreate a situation between an automaton and a PC master.
Final client source code A.7
Final server source code A.8

The automaton is in charge of the followings :

- acquiring and temporary storing the temperature through the i2c sensor
- indicating temperature with a warning light
- sending his registers contents by a master device's request

The master computer handles :

- request of automaton data registers with modbus protocol
- conversion of these registers content in a real temperature value
- outputing said temperature values in real time as graphic with "gnuplot"

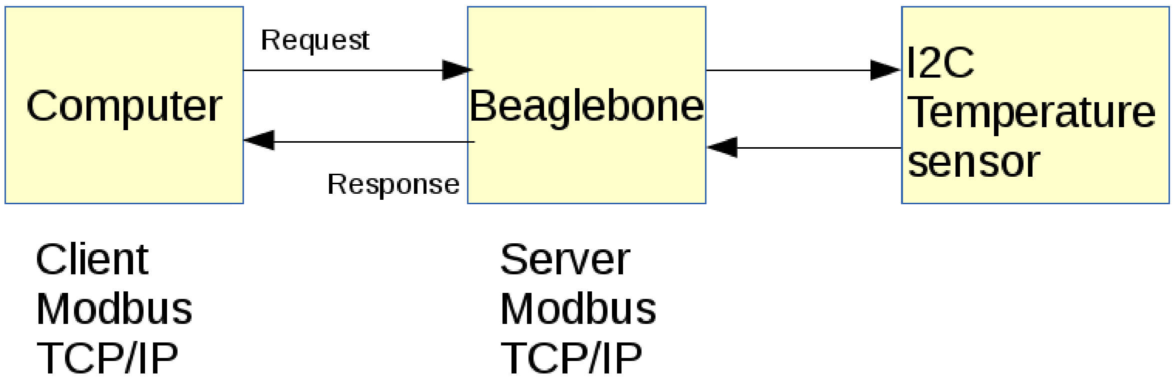
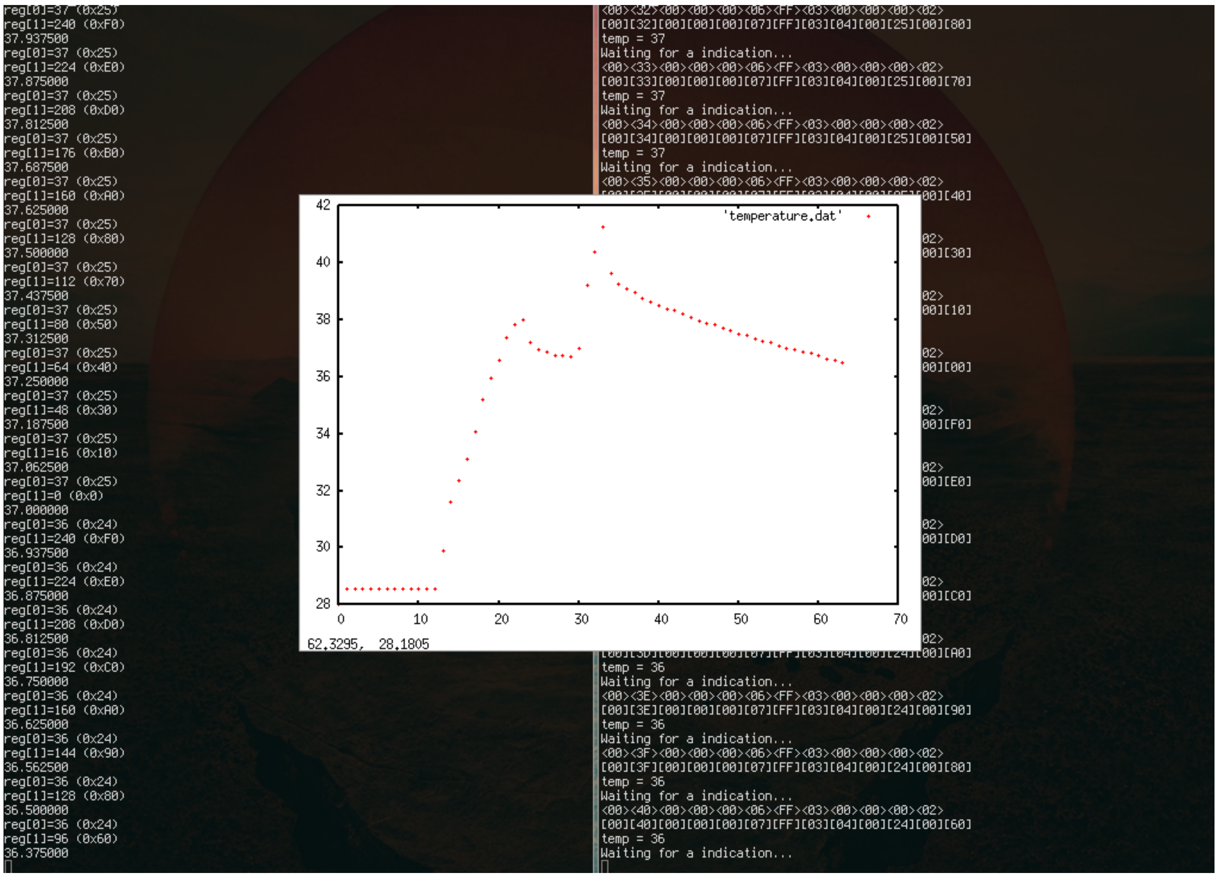


Figure 5.1: represenation of our setup

5.2 Results, observations

Here is the output of our final program

X axis is the duration in seconds, Y axis is the temperature in °C.



```
reg[0]=37 (0x25)
reg[1]=240 (0xF0)
37.937500
reg[0]=37 (0x25)
reg[1]=224 (0xE0)
37.875000
reg[0]=37 (0x25)
reg[1]=208 (0xD0)
37.812500
reg[0]=37 (0x25)
reg[1]=176 (0xB0)
37.687500
reg[0]=37 (0x25)
reg[1]=160 (0xA0)
37.625000
reg[0]=37 (0x25)
reg[1]=128 (0x80)
37.500000
reg[0]=37 (0x25)
reg[1]=112 (0x70)
37.437500
reg[0]=37 (0x25)
reg[1]=80 (0x50)
37.312500
reg[0]=37 (0x25)
reg[1]=64 (0x40)
37.250000
reg[0]=37 (0x25)
reg[1]=48 (0x30)
37.187500
reg[0]=37 (0x25)
reg[1]=16 (0x10)
37.062500
reg[0]=37 (0x25)
reg[1]=0 (0x0)
37.000000
reg[0]=36 (0x24)
reg[1]=240 (0xF0)
36.937500
reg[0]=36 (0x24)
reg[1]=224 (0xE0)
36.875000
reg[0]=36 (0x24)
reg[1]=208 (0xD0)
36.812500
reg[0]=36 (0x24)
reg[1]=192 (0xC0)
36.750000
reg[0]=36 (0x24)
reg[1]=160 (0xA0)
36.625000
reg[0]=36 (0x24)
reg[1]=144 (0x90)
36.562500
reg[0]=36 (0x24)
reg[1]=128 (0x80)
36.500000
reg[0]=36 (0x24)
reg[1]=96 (0x60)
36.375000
```

```
<00><32><00><00><00><06><FF><03><00><00><00><02>
[00][32][00][00][00][07][FF][03][04][00][25][00][80]
temp = 37
Waiting for a indication...
<00><33><00><00><00><06><FF><03><00><00><00><02>
[00][33][00][00][00][07][FF][03][04][00][25][00][70]
temp = 37
Waiting for a indication...
<00><34><00><00><00><06><FF><03><00><00><00><02>
[00][34][00][00][00][07][FF][03][04][00][25][00][50]
temp = 37
Waiting for a indication...
<00><35><00><00><00><06><FF><03><00><00><00><02>
                                            00][40]

                                            02>
                                            00][30]

                                            02>
                                            00][10]

                                            02>
                                            00][00]

                                            02>
                                            00][F0]

                                            02>
                                            00][E0]

                                            02>
                                            00][D0]

                                            02>
                                            00][C0]

                                            02>
                                            00][A0]
temp = 36
Waiting for a indication...
<00><3E><00><00><00><06><FF><03><00><00><00><02>
[00][3E][00][00][00][07][FF][03][04][00][24][00][90]
temp = 36
Waiting for a indication...
<00><3F><00><00><00><06><FF><03><00><00><00><02>
[00][3F][00][00][00][07][FF][03][04][00][24][00][80]
temp = 36
Waiting for a indication...
<00><40><00><00><00><06><FF><03><00><00><00><02>
[00][40][00][00][00][07][FF][03][04][00][24][00][60]
temp = 36
Waiting for a indication...
```

The LED lights up when the temperature gets above 37°C.

Chapter 6

Conclusion

Major goals of the project were achieved :

We have now an embedded device that behaves as a modbus slave, but is still really extensible with other sensors or that could pilot small servomotors with pwm outputs or almost anything imaginable.

Using a communication protocol on an higher abstract layer than we were used to was a really interesting thing to do, and our skills with linux and C language were largely improved during this project.

But we couldn't get further due to some difficulties we faced :

We wasted time while making some choices : trying to create a graphical user interface for our client with gtk or qt too early, without enough coding experience. We opted for a simple tool called dialog that can be simply used through bash scripts. The Choice of the temperature sensor took us too much time too, as we didn't have any prior experience chosing a sensor and wondering if it would be compatible with our device.

Then some steps took us a bit too much time too :
Angström linux seems easy to use at first sigh, but trying to configure it properly is a bit more difficult. Our coding experience wasn't very important, so it took us time to learn to include properly afew different library to the same program and compile it. It was quite hard to figure out how works and how to use the modbus protocol and the libmodbus by looking to various documentations.

What else could have been done in this project if we had more time is an extension to the can bus, or using multiple i2c devices on the same bus. Wiring the beaglebone to the internet through a router in order to monitor the temperature from anywhere with modbus could have been a fun thing to do too. Or simply using other automatons on a network all controled by the same master computer could be another implementation.

Appendix A

A.1 IO test

```c

#include <stdio.h>
#include <stdlib.h>
#include "iolib.h"

int
main(void)
{
	int ret;
	int del;
	int i;
	int portnum, pinnum, dirval;
	iolib_init();
	while(1)
	{
		printf("Enter port number (8-9):");
		scanf("%d", &portnum);
		printf("Enter pin number (1-46):");
		scanf("%d", &pinnum);
		printf("Input(0) or output(1)? (0-1):");
		scanf("%d", &dirval);
		printf("Setting direction...");
		ret=iolib_setdir(portnum,pinnum, dirval);
		if (ret!=0)
		{
			printf("error!\n\n");
			continue;
		}
		else
		{
			printf("done\n");
		}
		if (dirval==1)
		{
			printf("toggling output for 5 seconds...");
			fflush(stdout);
			for (i=0; i<25; i++)
			{
				pin_high(portnum, pinnum);
				iolib_delay_ms(100);
				pin_low(portnum, pinnum);
				iolib_delay_ms(100);
			}
			printf("done\n\n");
			continue;
		}
```

```
                if (dirval==0)
                {
                        printf("Observing input for 5 seconds\n");
                        printf("..........\n");

                        for (i=0; i<10; i++)
                        {
                                if (is_high(portnum, pinnum))
                                {
                                        printf("1");
                                }
                                else if (is_low(portnum, pinnum))
                                {
                                        printf("0");
                                }
                                else
                                {
                                        printf(".");
                                }
                                fflush(stdout);
                                iolib_delay_ms(500);
                        }
                        printf("  ..done\n\n");
                        continue;
                }
        } // end while(1)

        iolib_free();
        return(0);
}
```

A.2 Modbus test client

```
/*
 * *    test porgram   *
 *
 *
 *
 *
 * */

#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>
#include <modbus.h>


/* this program does nothing else than read some registers */


int main(void) // the program returns nothing
{
int i;
int rc ;        //received
//int addr=0    //addresses
//int nb=5              //number of addresses

modbus_t *mb;

uint16_t tab_reg[32];
uint8_t dest[8];

/* we use TCP */
mb = modbus_new_tcp("192.168.1.100", 502);
modbus_connect(mb);
modbus_write_register(mb, 3, 5 );  // write 5 in register 3
rc = modbus_read_registers(mb, 0, 8,tab_reg); // lit

modbus_write_bit(mb, 3, 1); //write 1 in coil 3
modbus_read_bits(mb,0,8,dest);
printf("coil[%d] is at %d \n ", 3,dest[3]);

for (i=0; i < rc; i++) {
printf("reg[%d]=%d (0x%X)\n", i, tab_reg[i], tab_reg[i]);    }


modbus_close(mb);
modbus_free(mb);
}
```

A.3 Modbus test server

```
/*
 * Copyright Â© 2008-2010 StÃ©phane Raimbault <stephane.raimbault@gmail.com>
 *
 * This program is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 3 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program.  If not, see <http://www.gnu.org/licenses/>.
 */

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>

#include <modbus.h>

int main(void)
{
    const uint16_t *src;
    float n=0;
    int socket;
    modbus_t *ctx;
    modbus_mapping_t *mb_mapping;

    ctx = modbus_new_tcp("192.168.1.100", 1502);
     modbus_set_debug(ctx, TRUE);

    mb_mapping = modbus_mapping_new(0, 0, 10, 10);
    if (mb_mapping == NULL) {
        fprintf(stderr, "Failed to allocate the mapping: %s\n",
                modbus_strerror(errno));
        modbus_free(ctx);
        return -1;
    }

    socket = modbus_tcp_listen(ctx, 1);
    modbus_tcp_accept(ctx, &socket);

    for (;;) {
        uint8_t query[MODBUS_TCP_MAX_ADU_LENGTH];
        int rc;

        rc = modbus_receive(ctx, query);
        if (rc != -1) {
        modbus_reply(ctx, query, rc, mb_mapping);
        }else { break;}
    }
        n =  modbus_get_float(src);

    printf("Quit the loop: %s\n", modbus_strerror(errno));
    printf("byte=%d\n",n);

    modbus_mapping_free(mb_mapping);
    modbus_close(ctx);
    modbus_free(ctx);

    return 0;
}
```

A.4 modbus and IO

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>

#include <modbus.h>


int main(void)
{
    int n;
    int i=0;
    int socket;
    modbus_t *ctx;
    modbus_mapping_t *mb_mapping;

    ctx = modbus_new_tcp("192.168.1.100", 1502);
//     modbus_set_debug(ctx, TRUE);

    mb_mapping = modbus_mapping_new(8,8,8,8);
    if (mb_mapping == NULL) {
        fprintf(stderr, "Failed to allocate the mapping: %s\n",
                modbus_strerror(errno));
        modbus_free(ctx);
        return -1;
    }

    socket = modbus_tcp_listen(ctx, 1);
    modbus_tcp_accept(ctx, &socket);

        uint8_t reg[MODBUS_TCP_MAX_ADU_LENGTH];
        int rc;
    for (;;) {

        rc = modbus_receive(ctx, reg);
        if (rc != -1) {
        n = rc;
        modbus_reply(ctx, reg, rc, mb_mapping);
        }else { break;}
    }
// si le coil 3 est actif : on execute le programme de test io
if ((*mb_mapping).tab_bits[3] == 1) {
        int del;
        iolib_init();
        iolib_setdir(8,7, DIR_IN);
        iolib_setdir(8,8, DIR_OUT);

        while(1)
        {
                if (is_high(8,7))
                {

                pin_high(8,8);

                }
                if (is_low(8,7))
                {

                pin_low(8,8);
                }
        }
        iolib_free();   }
    printf("Quit the loop: %s\n", modbus_strerror(errno));

    modbus_mapping_free(mb_mapping);
    modbus_close(ctx);
    modbus_free(ctx);

    return 0;
}
```

A.5 reading analog input

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>


int main(int argc, char *argv[])
{
        FILE* fichier=NULL;
        int t=0;
        char buff[4];
        int reg1; int reg2; int reg3; int reg4;

        /* Lecture du fichier  */

fichier = fopen("test", "r+");

if (fichier != NULL)
{
        fscanf(fichier , "%d" , &t);
        printf("la variable t contient maintenant : %d\n", t);
        printf("soit en hex : %X\n",t);
}
else
{       printf("Error when trying to open the file"); }

fclose(fichier);

return 0;
}
```

A.6 i2c acquisition

```c
#include <stdio.h>
#include <stdlib.h>
#include </usr/include/linux/fcntl.h>
#include </usr/include/linux/i2c-dev.h>

int main()
{
          int tmp102;
          float data;
          int temp;
          int adress ;
          //declaration of registers wich are used for storage data from tmp102
          char registre_lecture[1] = {0};

          //selection of the address device
          printf("adresse \n");
          scanf("%d", &adress);
          //open the i2c bus as a read/write bus
          if(( tmp102 = open("/dev/i2c-1", O_RDWR)) <0){
                    perror("failed to open the i2c bus \n");
                    exit(1);
                    }
          // set tmp102 as a slave i2c device
          if((ioctl( tmp102,I2C_SLAVE, adress))<0){
                    printf("communication error \n");
          }

          //request contents of device's register
          if(read(tmp102, registre_lecture,2)!=2){
                    perror("read error");
                    }
          else{  //storage data in two 8 bits registers
                    temp = ((registre_lecture[0]) << 8) | (registre_lecture[1]);
                    }
                    temp >>= 4;

                    // test two compliment
                    if (temp & (1 << 11)){
                    temp |= 0xF800;
                    }

                    //convert the value in a float
                    data = temp * 0.0625;
                    // print data value
          printf("data = %f \t temperature = %04f \n",data, temp * 0.0625);
          // close the connection
          close( tmp102);
          return 0;
}
```

A.7 final client

```c
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>
#include <modbus.h>


/* this program does nothing else than read some registers */
struct timeval old_response_timeout;
struct timeval response_timeout;

int main(void) // the program returns nothing
{
FILE* fichier_sortie = NULL;

int i,j;
int temp;
float data = 28.0;
float pas = 0.0;
int rc ;        //received

modbus_t *ctx; //initialisation context modbus

uint16_t tab_reg[32];
uint8_t dest[8];

/* we use TCP  */
ctx = modbus_new_tcp("192.168.1.100", 502); // normally modbus connexions are on port 50


modbus_connect(ctx);

fichier_sortie = fopen("temperature.dat", "w");
fprintf(fichier_sortie, "%f %f  \n", pas, data);
fclose(fichier_sortie);

system("gnuplot -p -e \"plot 'temperature.dat'\" loop.plt &");
for(j=0; j<300; j++){
//modbus_write_register(ctx, 3, 5 );  // ecrit 5 dans le registre 3
rc = modbus_read_registers(ctx, 0, 2,tab_reg); // lit

//modbus_write_bit(ctx, 3, 1); //write 1 in coil 3
//modbus_read_bits(ctx,0,8,dest);
//printf("coil[%d] is at %d \n ", 3,dest[3]);
pas = pas +1.0;

for (i=0; i < rc; i++) {

printf("reg[%d]=%d (0x%X)\n", i, tab_reg[i], tab_reg[i]);    }

temp = ((tab_reg[0] <<8) |( tab_reg[1]));
temp >>=4;
        if (temp & (1 << 11)){
                temp |= 0xF800;
        }
data = temp * 0.0625;
printf("%04f  \n",data);
fichier_sortie = fopen("temperature.dat", "a");
fprintf(fichier_sortie, "%f %f  \n", pas, data);
fclose(fichier_sortie);

usleep(1000000);
}
fichier_sortie = fopen("temperature.dat", "a");
fprintf(fichier_sortie, "&");
fclose(fichier_sortie);
//scanf("%d", &fin);
modbus_close(ctx);
modbus_free(ctx);
}
```

A.8 final server

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>
#include </usr/include/linux/fcntl.h>
#include </usr/include/linux/i2c-dev.h>
#include <modbus.h>
#include "/home/root/BBBIOlib-master/BBBio_lib/BBBiolib.h"


int main(void)
{
int tmp102;
float data;
int temp;
int adress ;
char registre_lecture[1] = {0};
int LIMIT=37;

        printf("i2c adresse \n");
        scanf("%d", &adress);
//i2c init part
        if(( tmp102 = open("/dev/i2c-1", O_RDWR)) <0){
                perror("failed to open the i2c bus \n");
                exit(1);
                }

        if((ioctl( tmp102,I2C_SLAVE, adress))<0){
                printf("can't dialog with bus or slave \n");
        }
//gpio init part
    iolib_init();
    iolib_setdir(8,12, BBBIO_DIR_OUT);


//main part
        while(1){
    float n=0;
    int socket, rc;
    uint8_t query[MODBUS_TCP_MAX_ADU_LENGTH];
    modbus_t *ctx;
    modbus_mapping_t *mb_mapping;


    ctx = modbus_new_tcp("192.168.1.100", 502);
    modbus_set_debug(ctx, TRUE);
    mb_mapping = modbus_mapping_new(0, 0, 10, 10);
    if (mb_mapping == NULL) {
        fprintf(stderr, "Failed to allocate the mapping: %s\n",
                modbus_strerror(errno));
        modbus_free(ctx);
        return -1;
    }

    socket = modbus_tcp_listen(ctx, 1);
    modbus_tcp_accept(ctx, &socket);

    for (;;) {

        if(read(tmp102, registre_lecture,2)!=2){
                perror("read error");
                }
    //rise a flag if temperature is too high, here 30°C
printf("temp = %i\n"; registre_lecture[0]);
               if (registre_lecture[0] > LIMIT) then
               {
                pin_high(8,12);      //allume la LED
               }
               else{
                pin_low(8,12);       //eteint la LED
               }

    (*mb_mapping).tab_registers[0] = registre_lecture[0];
    (*mb_mapping).tab_registers[1] = registre_lecture[1];

        rc = modbus_receive(ctx, query);
        if (rc != -1) {
        modbus_reply(ctx, query, rc, mb_mapping);
        }else {
        printf("Connexion ended \n");
        break;
        }
    }

    modbus_mapping_free(mb_mapping);
    modbus_close(socket);
    modbus_free(ctx);
        }
    return 0;
}
```

A.9 Bibliography

Bibliography

[1] Getting started with the beaglebone, Math Richardson

[2] iolib.h library
http://www.element14.com/community/community/knode/single-board_computers/next-ge
n_beaglebone/blog/2013/10/10/bbb–beaglebone-black-io-library-for-c

[3] beaglebone network configuration: http://derekmolloy.ie/category/embedded-systems/beaglebone/

[4] ANR-LAN Communication protocol modbus-TCP quick overview

[5] The EXTENSION 2008 - introduction to modbus serial and modbus tcp

[6] libmodbus manual page: http://libmodbus.org/docs/v3.1.1/

[7] http://modbus.org/faq.php

[8] modicon modbus protocol reference guide

[9] i2c course from Jacques Michel M1MNE

[10] TI datasheet for the TMP102

[11] using i2c-dev.h : http://makingaquadrotor.wordpress.com/2012/07/08/i2c-on-the-beaglebone/

[12] information about the linux kernel : https://www.kernel.org

[13] gnuplot guide: http://www.gnuplot.info/documentation.html