CollSoft Programmer's Guide

Author: Daniele Berto (daniele.fratello@gmail.com)

Document version: 1.1

Document purposes: In this document is explained the architecture of the CollSoft system in order to provide useful information to the programmer who wants to improve it.

Github repository: *https://github.com/danready/collsoft*

CollSoft version: 1.1

Last update: 2016-12-15

Index

Index of figures

1 Introduction

CollSoft is free software. It is a 13,000 – 15,000 C/C++/Qt/Flex/LDAP line project, so a guideline is needful.

The system was created in: Linux Kernel 4.2.0-42-generic
Ubuntu 14.04 LTS 64 bit.
Intel® Core™ i7-6700HQ CPU @ 2.60GHz × 8.

Must need: Drivers LAM Technologie DS044 with the firmware v … and the encoders.

2 System schema

In this section you can find a high level overview of the system components. **Basically, all the components can be built using make command in the root directory.**
In figure 1 you can see an high level overview of the basic working flow of the system.

2.1 Requirements

The system was created using Ubuntu 14.04 LTS 64 bit. It is designed for the GNU/Linux operating system. In particular, three components are required:

- Flex
- Qt framework (in particular the qmake utility)
- OpenLDAP

2.2 How to obtain the system

In order to obtain the collsoft system it is sufficient to digit these commands in a shell:

git clone https://github.com/danready/collsoft

I suggest you to visit the web page https://github.com/danready in order to find additional material on my projects.

2.3 Special precaution

The executables files must be placed in the right locations in order to allow the system to work correctly. The right locations are indicated in the paragraphs below (Executable path). Using the make command the path of each program could be the right one but I recommend you to check it.

2.4 CollSoft

Names used to refer to it: CollSoft, Server Program, Server, CollSoft Server.

Base Directory: ServerProgram.

Language: C/C++/Flex

General Purpose: managing the communication between the user and the drivers.

Configuration file: DefineGeneral.h .

Input/output: TCP/IP or stdin in according to the setting in DefineGeneral.h .

How to compile it: makefile at the root directory or makefile in the ServerProgram directory.

Special attention: Flex package is needed (v. http://flex.sourceforge.net/).

Be careful to set TCP/IP parameters in DefineGeneral.h before compiling it.

The CollSoft supposes the firmware v. … installed in the drivers.

You have to execute the program with root privileges.

Where the program may be executed: the calculator connected with the LAM programmer.

Executable path: collsoft/ServerProgram/bin/Collsoft

2.5 Check Existence

Names used to refer to it: CheckExistence, Existence server, CheckExistence Server.

Base Directory: CheckExistence

Language: C/C++

General Purpose: allowing the clients to remote switch ON/OFF the Server Program.

Configuration file: DefineGeneral.h

Input/output: TCP/IP

How to compile it: makefile at the root directory or makefile in the CheckExistence directory.

Special attention: Killing Check Existence cause the death of the Server Program.

Be careful to set TCP/IP parameters in DefineGeneral.h before compiling it.

You have to execute the program with root privileges.

Where the program may be executed: the calculator connected with the LAM programmer as the CollSoft program.

Executable path: collsoft/CheckExistence/bin/CheckExistence

2.6 ExpertGUI

Names used to refer to it: ExpertGUI, expert mode.

Base Directory: ExpertGUI

Language: Qt

General Purpose: providing a sophisticated graphical interface to communicate with the Server Program and the Check Existence.

Configuration file: -

Input/output: GUI, TCP/IP with the Server Program and Check Existence.

How to compile it: makefile at the root directory or *"qmake ExpertGUI.pro && make"* or Qt Creator **disabling shadow building ("projects"-> "general" -> switch off "shadow build").**

Special attention: **It is required the *qmake* utility in order compile it using the makefile in the collsoft root directory.**

Where the program may be executed: workstation in control room.

Executable path: collsoft/ExpertGUI/bin/ExpertGUI

2.7 UserGUI

Names used to refer to it: UserGUI

Base Directory: UserGUI

Language: Qt/LDAP (C interface).

General Puropose: providing a simplified interface to communicate with the Server Program. Opening an ExpertGUI if necessary.

Configuration file: DefineGeneral.h

Input/output: TCP/IP

How to compile it: makefile at the root directory or *"qmake ExpertGUI.pro && make"* or Qt Creator **disabling shadow building ("projects"-> "general" -> switch off "shadow build").**

Special attention: Be careful to set TCP/IP parameters in DefineGeneral.h before compiling it. You have to set up a compatible LDAP server following the guideline in

this document. **If you want to use UserGUI without LDAP, you can insert username "admin" and password "admin".**

It is required the *qmake* utility in order compile it using the makefile in the collsoft root directory.

Where the program may be executed: workstation in control room.

Executable path: collsoft/UserGUI/bin/UserGUI

2.8 Firmware

…

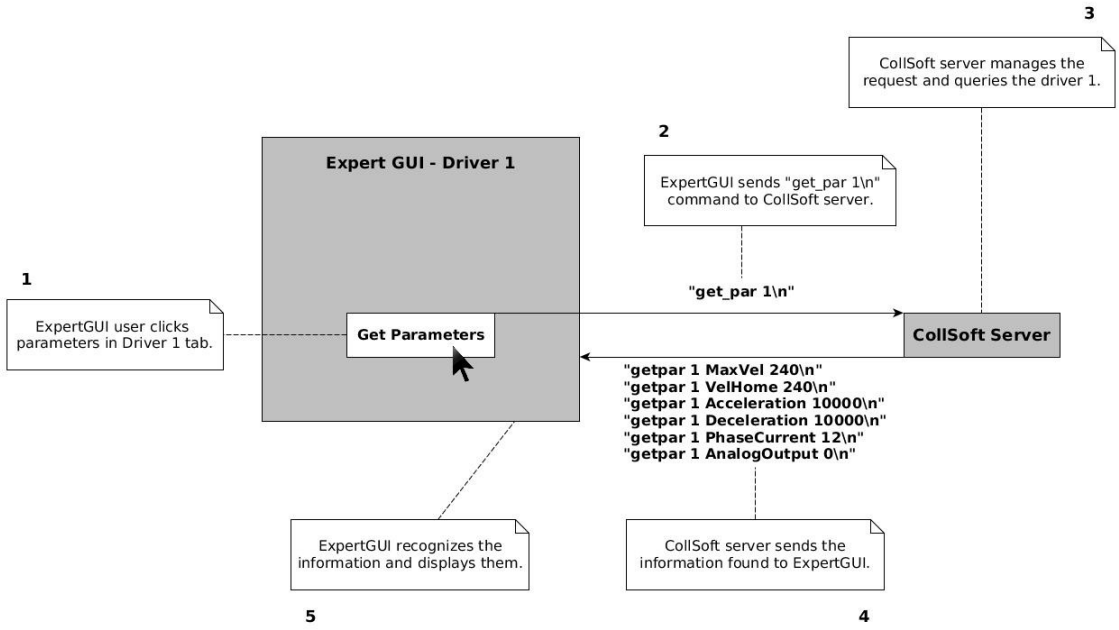

Figure 1: High level overview of the system working flow

2.9 Special annotation

The functions that communicate with the programmer are designed assuming that int/unsigned int = 32 bit.

3 Ready to start

I have to point out that if you want to install CollSoft you might install all the mechanical system before. In particular, you have to update the driver with the Firmware.

However, you can compile and run all the programs without any device connected to the PC.

Before the compilation you have to find out and open the DefineGeneral.h files and change the IP address and the socket Port:

- **CheckExistence/DefineGeneral.h**
- **ServerProgram/SourceCode/DefineGeneral.h:**

```
//SERVER_IP defines the IPv4 address of this application
#define SERVER_IP   "127.0.0.1"

//SERVER_PORT defines the port number of this application
#define SERVER_PORT 1111
```

- **UserGUI/UserGUI/DefineGeneral.h (in this file are contained also useful information about LDAP parameters).**

```
//#define COLLSOFT_IP "192.84.144.245"
#define COLLSOFT_IP "127.0.0.1"
#define COLLSOFT_PORT 1111
#define EXISTENCE_IP "127.0.0.1"
#define EXISTENCE_PORT 1112

#define LDAP_HOST "127.0.0.1"
#define LDAP_PORT 389
#define LDAP_DN "cn=admin, dc=elinp, dc=com"
```

#define LDAP_PW "fantinodivaren"

#define LDAP_BASE_DN "ou=wp09,dc=elinp,dc=com"

I suggest you to remember them because these are the parameters used to interact with the applications via TCP/IP.

N.b.: you have to manage your firewall to profitably use that addresses with that ports!

To compile the CheckExistence Server, the CollSoft server, the ExpertGUI and the UserGUI you have to install the **flex** package (ex. *yum install flex* or *sudo apt-get install flex*, check your linux distribution manual for the correct procedure) and the **openldap-devel** package (ex. *yum install openldap-devel* or *sudo apt-get install openldap-devel*, check your linux distribution manual for the correct procedure). You also to set up the Qt framework. **In particular, your system must have the qmake utility in order to use the general make command.**
Qt version greater than 5.6 is also required for the **ExpertGUI** and **UserGUI**.

Than you have to move in the header directory and run *make* command.
The root directoy could be, for example: */home/daniele/Desktop/collsoft.*

The ***make*** command produces:

CheckExistence *in CheckExistence/bin/*

CollSoft *in ServerProgram/bin/*

ExpertGUI *in ExpertGUI/bin/*

UserGUI *in UserGUI/bin/*

N.b. Be careful! If you compile the system in a different way, you may be check the executable files to be placed in the path indicated above!

N.b.: Everytime you modify a DefineGeneral.h file, you have to clean the previous executable using make clean command and you have to rerun make command.

Another way to compile the CollSoft ExpertGUI is to open the Expert*GUI/ExpertGUI/**ExpertGUI.pro*** file in **Qt Creator** and build it **disabling the shadow building ("projects"-> "general" -> switch off "shadow build").**

In alternative, you could move in the ExpertGUI/ExpertGUI directory and execute:

```
qmake ExpertGUI.pro
make
```

commands.

N.b. Do not run qmake command without specifying *ExpertGUI.pro* file in order to avoid the overwriting of ExpertGUI.pro file!

The building process produces, in the both ways,

***ExpertGUI** in ExpertGUI/ExpertGUI/*

To compile the UserGUI you may follow the same steps of the compilation of the ExpertGUI.

Be careful! To profitably run the UserGUI you must set up an LDAP server with a database compatible with the UserGUI: see the section LDAP for further instruction.

If you want to use UserGUI without LDAP, you can insert username "admin" and password "admin".

3.1 First software beginning

Warning! CheckExistence and CollSoft must be executed with root privileges!

In order to use this system is important to not move the executable CheckExistence and the CollSoft server executable files from their locations: every executable have to remain in its place in order to avoid file/program path problems.

First step:

Move to *CheckExistence/bin/* using the shell.

Execute **CheckExistence with root privileges.**

Second step:

Launch the **ExpertGUI** or the **UserGUI** and follow the instructions.

N.b. When you modify *DefineGeneral.h* file you have to clean the previous compilation with *"make clean"* and execute another one with *"make"*.

N.b. If *CheckExistence* server creates an instance of *CollSoft* and *CheckExistence* dies, then also *CollSoft* dies.

4 Some consideration about flex

In Utils.h you can find these procedures:

//This function is generated by flex. Its purpose is to analyze SerialDrvLog.txt.
void Analizza1(vector<SerialCouple>& serial_list, int max_log);*

//This function is generated by flex. Its purpose is to analyze FileParLog.txt.
void AnalizzaFilePar(vector<ParameterStruct>& parameter_arg, int max_log);*

//This function is generated by flex. Its purpose is to analyze EncoderLog.txt.

void AnalizzaFileEncoder(vector<EncoderStruct>& encoder_arg, int max_log);*

They are, respectively, defined in *DrvList.c*, *FilePar.c* and *Encoder.c* .
These files are very huge.... but don't worry! They are automatically generated by flex!
In the make file you can find these lines:

```
flex -o./SourceCode/DrvList.c -PDrv1 ./LogFile/SerialDrvLog.flex
flex -o./SourceCode/FilePar.c -PFilePar ./LogFile/FileParLog.flex
flex -o./SourceCode/Encoder.c -PEncoder ./LogFile/EncoderLog.flex
```

Their purpose is to generated them. *-PDrv1* means that flex use *"Drv1"* prefix instead of the default one *"yy"*.
So, for example, the resulting functions will be *Drv1wrap* instead of *yywrap*. This choice is fundamental because in this way we can create three different scanners without name conflict.
Indeed, compiling the files without *-P* option, the result is three files with the same *yywrap* function. So, the compilation will fail!

N.b. These files uses "Utils.h" and "DefineGeneral.h". The include path is simply "Utils.h" and not, for example, "../SourceCode/SerialDrvLog.flex" because the file that will be compiled is DrvList.c generated by flex from SerialDrvLog.flex and moved in /SourceCode/DrvList.c.

5 Some consideration about modbus libraries

The library used to communicate with the programmer (BasicModbusLibrary) needs special attention.

This library is obtained simplifying the ***libmodbus v.3.0.6*** by **Stephane Raimbault** (see **http://libmodbus.org** for information).

In particular, I have converted the library modifying the dynamic linking to static linking.

I have to emphasize that my version does not not implement all the functions contained in the Stephane Raimbault one.

In particular, you have to modify the timout manually: I have moved its definition in ***DefineGeneral.h*** of the Server Program.

So, if you want to change it, you have to modify the definitions in **DefineGeneral.h**. Than, you have to recompile the program with *"make clean"* and *"make"*.

In **DefineGeneral.h:**

```
#define _RESPONSE_TIMEOUT    500000
#define _BYTE_TIMEOUT        500000
```

So, tested functions are:

modbus_new_rtu

modbus_free

modbus_strerror

modbus_set_slave

modbus_read_registers

modbus_write_registers

modbus_flush

And **modbus_t** data type.

6 Communication protocol

Every output has an header like, for example, *"Exp: "* or *"Connect: "*. This header is very useful for the client applications like **ExpertGUI** or **UserGUI** because they can recognize the output and manage it in a consistent way.

6.1 A communication example

Now, an example of communication between the ExpertGUI and the server is provided. In this scenario, the **ExpertGUI** tries to get the parameter of the driver 1, so it sends to the server the command *"get_par 1"*.

Then, the server sends to the **ExpertGUI**:

```
getpar 1 MaxVel 1000
getpar 1 Acceleration 50
getpar 1 Deceleration 50
getpar 1 ecc....
```

So, the **ExpertGUI** reads the output and put the **MaxVel** value in the corresponding field, the **Acceleration** value in the corresponding field and so on....

The function **FindPointer** is very useful and it skips one word.

7 Some consideration about the firmware interaction with the system

It's obvious that every functions that communicate with the drivers is designed knowing how the firmware works.

Be careful to the hard coded informations like this:

```
int count = 0;
while (status_state != 4 && status_state != 5 && status_state != 0 && count < LIMITSTATUS_STATE)
```

```
{
        ReadStatusState(...);
}
```

status_state = 4 or *status_state = 5* means that the previous operation is terminated.

count is a timeout: if the operation is not ultimated in the times specified by *LIMITSTATUS_STATE*, the homing function is aborted.

N.b. To execute a movimentation is required to set CountTargetPosition, than to set to STATEMOVEREL the request_state register.

8 General Purpose Libraries

This section will provide a description of the objects used to manage the Input/Ouput/LogFile of either the Server Program and the Check Existence program: ***ApplicationSetup, LogFile, OutputModule, TcpUser, Input, CommunicationObject.***

These objects are strictly coupled and they call each others. Their purpose is to provide a programmer friendly interface to manage the input and the output of the program. **CommunicationObject** listens to the incoming connection and records the commands sent via *TCP/IP.*

Input object records the input sent via stdin.

The programmer can choose the input modality modifying this instruction in the **DefineGeneral.h** file:

```
//INPUTMODALITY defines the method for fetching input.
//Available option are:
//tcp, that allows the input only via tcp/ip
```

```
//all, that allows the input either via tcp/ip and via stdin
//user, that allows the input only via stdin.
//If the option type is not recognized, all modality is actived.
#define INPUTMODALITY    "tcp"
```

The default communication way is *"tcp"*.

TcpUser is a struct used to record the command and his sender.

The **OutputModule** provides a very smart way to manage the output: the programmer only have to call the Output method to sent the output to the user who has request the command.

The **LogFile** object is useful to write some information in the LogFile.

The **ApplicationObject** record some useful informations like the path of the log files.
simplified schema:
ExpertGUI/UserGUI --> get_par 2 --> CommunicationObject --> TcpUser <-- Main.c --> OutputModule --> ExpertGUI/UserGUI.
LogFile remarkable methods:

LogFileSet
LogFile.

N.b. The CommunicationObject treats the "\n" like a command end.

So, if you send this command:

```
"get_par 2\nget_par 3\n"
```

the **CommunicationObject** cuts it in:

```
"get_par 2"
```

"get_par 3"

So, the server will execute first *"get_par 2"*, then *"get_par 3"*.

N.B. I recommend to end every command sent by the client with "\n".

Indeed, the **TCP/IP** protocol is stream oriented, so if you write a code like this:

```
Send("get_par 1");
Send("get_par 2");
```

you may expect that the server will receive *"get_par 1"* and *"get_par 2"* but there is no guarantee that this will be the real behavior.

The server may receive

"get_par 1get_par 2".

If you add **"\n"** at the end of each command like this:

```
Send("get_par 1\n");
Send("get_par 2\n");
```

the server may receive

"get_par 1\n"

and

"get_par 2\n"

but if it will receive *"get_par 1\nget_par 2\n"* that's okay because the **CommunicationObject** will divide it in

"get_par 1"

and

"get_par 2".

N.B. If you want a simple example to use this libraries you have to study the **CheckExistence** program. That program is very elementary because it uses the general purpose libraries and can execute only three commands: **check_process**, **kill_process** and **new_process**.

9 Check Existence Commands

Each command may be sent via **TCP/IP.**
This is done setting the **INPUTMODALITY** in the **DefineGeneral.h** file:

The commands that the CheckExistence server could execute are (they are reported in order to appearance in the **MainExistence.c** file):
check_process: checks if **CollSoft** program exists.
kill_process: checks if **CollSoft** program exists, if yes kills it.

new_process: checks if **CollSoft** program exists, if not creates an instance of it.

N.B. all these commands are CASE INSENSITIVE.

10 CollSoft Server Command

Each command may be sent via **TCP/IP** or via **stdin**.
This is done setting the **INPUTMODALITY** in the **DefineGeneral.h** file:

#define INPUTMODALITY "tcp"

***INPUTMODALITY** defines the method for fetching input.*
Available option are:
“tcp”, that allows the input only via tcp/ip
“all”, that allows the input either via tcp/ip and via stdin
“user”, that allows the input only via stdin.
If the option type is not recognized, all modality is activated.

The default way to compile the server is to set **INPUTMODALITY** to **“tcp”** because the server will be executed by the **CheckExistence** server and it will receive command by **TCP/IP** even if the user tries the system locally.
The command that the server could be execute are (they are reported in order to appearance in the Main.c file):

exit: stop the server. This command can be sent only via **stdin**. If you want to kill the server via **TCP/IP** you have to use the **CheckExistence** server.
read_serial_log: this command read the file *SerialDrvLog.txt*.

read_par_log: this command read the file *FileParLog.txt*.

read_log: this command read the file *GeneralLog.txt.*

read_encoder_log: this command read the file *EncoderLog.txt.*

check_drv_assoc: this command check the association between the drivers serial number found in the *SerialDrvLog.txt* file and the real situation.

check_par_assoc: this command check the association between the drivers parameters found in the *FileParLog.txt* file and the real situation.

check_encode_assoc: this command check the association between the encoder values contained in the *EncoderLog.txt* file and the real situation.

connect absoluteprogrammerpath: this command tries to connect the server with the programmer indicated by *"absoluteprogrammerpath"*.
help: prints the list of the commands that the server can execute.
get_par drvnum: prints the parameters of the driver indicated by **drvnum**. Parameters printed are: max_vel, velhome, acceleration, deceleration, phase_current, analog_output0.
check_position drvnum: check if the actual position of the driver indicated by drvnum correspond with the one indicated by the encoder (analog_input0).

set_par drvnum max_vel acceleration deceleration PhaseCurrent AnalogOutput0: set max_vel (Each unity of maxvel correspond to 0.25rpm), acceleration and deceleration (Each unity of acceleration and deceleration correspond to 1rpm/s), phasecurrent and AnalogOutput0 of the driver specified with drvnum.
homing drvnum: executes the homing procedure for the driver indicated by drvnum.

get_mov_par drvnum: this command obtains the actual position and the AnalogInput0 values of the driver indicated by drvnum.

encode drvnum: this command start the encoding procedure for the driver indicated by **drvnum**.

move_to drvnum targetposition: this command set to targetposition the target position of the driver indicated by drvnum.

get_all_parameter: this command is equivalent to execute get_move_par drvnum, get_par drvnum and check_position drvnum for the all drivers.

homing_mult drvnum1 drvnum2 drvnum3 drvnum....: this command execute the

homing procedure for the driver indicated by drvnum1, drvnum2, drvnum3, drvnum....

moveto_mult targetposition drvnum1 drvnum2 drvnum3 drvnum....: this command set the target position to targetposition of the drivers indicated by drvnum1, drvnum2, drvnum3, drvnum....

setmult_par max_vel acceleration deceleration PhaseCurrent AnalogOutput0 drvnum1 drvnum2 drvnum3 drvnum....:

set max_vel (Each unity of maxvel correspond to 0.25rpm), acceleration and deceleration (Each unity of acceleration and deceleration correspond to 1rpm/s), phasecurrent and AnalogOutput0 of the drivers specified with drvnum.

check_internal_status:

this command retrieves the content of the GeneralStatus struct.

If GeneralStatus.assoc_file_status == 1 means the user has already executed the procedure to check

the association between the serial numbers contained in the SerialDrvLog.txt file and the real situation.

If GeneralStatus.par_file_status = 1 means the user has already executed the procedure to check

the association between the parameters contained in the FileParLog.txt file and the real situation.

If GeneralStatus.encoder_file_status = 1 means the user has already executed the procedure to check

the association between the encoder values contained in the EncoderLog.txt file and the real situation.

load_encoder_from_file: this command gets the encoding parameters for each drivers from the EncoderLog.txt file and use it to accomplished the check_position command.

read_actual_encoder_values: this command prints the actual encoding parameters that will be used to accomplished the check_position command.

device_list: this command prints the device contained in /dev
If the server not recognized the command, this message will be print:

"Unrecognized command. Digit 'help' to see the list of all commands available."
N.B. all these commands are CASE INSENSITIVE. Ex.: "exit", "Exit", "EXIT", "eXIT" are equivalent.

11 Expert GUI Message

Now, it will listed the messages that reacts the **ExpertGUI**.

For reacts it means all the messages relevant for **readTcpData** function that is connected with **readyRead** signal.

Socket: _pSocket
Signal: readyRead
Slot: readTcpData

Messages list (expressed in regular expression):

1. **"^Device list:"**
2. **"^Reading LogFile..."**
3. **"^Loading encoder values from file:"**
4. **"^Check Drv Assoc:"**
5. **"^Check Par Assoc:"**
6. **"^Check Encode Assoc:"**
7. **"^Connect:"**
8. **"^Welcome:"**
9. **"^Check position warning!"**

10. "^get_pos_status[ ][0-9]{1,2}[ ]-{0,1}[0-9]{1,5}"

11. "^InternalStatusSerial: [01]$"

12. "^InternalStatusParameter: [01]$"

13. "^InternalStatusEncoder: [01]$"

14. "^getpar[ ][0-9]{1,2}[ ]((MaxVel)|(VelHome)|(Acceleration)|(Deceleration)|(PhaseCurrent)|(AnalogOutput0))[ ]-{0,1}[0-9]{1,20}$"

15. "^get_mov_par[ ][0-9]{1,2}[ ]((CurrentPosition)|(AnalogInput0))[ ]-{0,1}[0-9]{1,20}$"

All other messages are printed in Expert Mode Message (QtextEdit ExpertModeMessage).

Socket: _pSocket_existence

Signal: readyRead

Slot: readTcpData_existence

For Check Existence server communication there is no problem: no regolar expression matching is performed by **readTcpData_existence**.

The only **ExpertGUI** reaction is to print the output in **CheckExistenceLog QtextEdit**.

12 User GUI message

Now, it will listed the messages that reacts the UserGUI.

For reacts it means all the messages relevant for **readTcpData** function that is connected with **readyRead** signal.

Socket: _pSocket

Signal: readyRead

Slot: readTcpData

Messages list (Expressed in regular expression):

1. "^get_pos_status[ ][0-9]{1,2}[ ]-{0,1}[0-9]{1,5}"

es.

get_pos_status 1 0

The first parameter, **"1"**, indicated the sled number. It must be between **1** and **14**.
When this message is received, **ReadTcpData** changes the color of the sled icon with an appropriate one.
-1: red
0: green
1: yellow.

13 Authomatic settings protocol

N.B. This socket has no signal connected. The reading is performed with **WaitForReadyRead** function.

Socket: _pSocket_1
Pseudo code algorithm:

Connection to COLLSOFT_IP, COLLSOFT_PORT (v. DefineGeneral.h)
if connection okay
read message
send "connect /dev/ttyUSB0"
read message
if reading okay
if command success (it means the server has sent this regular exp: "(CONNECTION SUCCESS)|(CONNECTION done)|(CONNECTION start)")
status_1 = true

disconnect from host

if status1 = true

Connection to COLLSOFT_IP, COLLSOFT_PORT (v. DefineGeneral.h)

if connection okay

read message

send "load_encoder_from_file"

read message

if reading okay

if command success (it means the server has sent this regular exp: "Loading encoder values from file: okay")

status_2 = true

disconnect from host

N.B. status_1 and status_2 ARE NOT global variables.

14 Command sent by UserGUI

get_all_parameter

homing

homing_mult

move_to

The server output to these command is very articulated but the only message relevant for the UserGUI is **"^get_pos_status[ ][0-9]{1,2}[ ]-{0,1}[0-9]{1,5}".**

15 Server Input/Output

Now, it will be explain the relevant output of the server for each command. For relevant output it means the output that is captured by **UserGUI** or **ExpertGUI**.

Command name: a mnemonic name to simply refer to the command.

Command syntax: the syntax you can use to execute the command. All the commands are case-insensitive.

15.1.1 exit

Name:
exit

Syntax:
exit

Meaning:
stop the server

Server regular expression:
"^[Ee][Xx][iI][tT][\t]*$"

The output of this command is not relevant because it could only be sent by stdin.

15.1.2 read_serial_log

Name:
read_serial_log

Syntax:
read_serial_log

Meaning:
this command read the file SerialDrvLog.txt.

Server regular expression:

"^[Rr][Ee][Aa][Dd]_[Ss][Ee][Rr][Ii][Aa][Ll]_[Ll][Oo][Gg][\t]*$"

This command is relevant for **ExpertGUI**. The server add this prefix to every burst: "Reading LogFile...\n".
So, when **ExpertGUI** in **ReadTcpData** slot recognize "Reading LogFile...", prints the burst received in the **Expert Mode QTextEdit** and in **GeneralLog QTextEdit**.

15.1.3 read_par_log

Name:
read_par_log

Syntax:
read_par_log
Meaning:
this command read the file FileParLog.txt.

Server regular expression: "^[Rr][Ee][Aa][Dd]_[Pp][Aa][Rr]_[Ll][Oo][Gg][\t]*$"

This command is relevant for **ExpertGUI**. The server add this prefix to every burst: "Reading LogFile...\n".
So, when **ExpertGUI** in **ReadTcpData** slot recognize "Reading LogFile...", prints the burst received in the **Expert Mode QTextEdit** and in **GeneralLog QTextEdit**.

15.1.4 read_log

Name:
read_log

Syntax:
read_log

Meaning:
this command read the file GeneralLog.txt

Server regular expression: "^[Rr][Ee][Aa][Dd]_[Ll][Oo][Gg][\t]*$"

This command is relevant for **ExpertGUI**. The server add this prefix to every burst: "Reading LogFile...\n".
So, when **ExpertGUI** in **ReadTcpData** slot recognize "Reading LogFile...", prints the burst received in the **Expert Mode QTextEdit** and in **GeneralLog QTextEdit**.

15.1.5 read_encoder_log

Name: read_encoder_log
Syntax: read_encoder_log

Meaning: this command read the file EncoderLog.txt

Server regular expression: "^[Rr][Ee][Aa][Dd]_[Ee][Nn][Cc][Oo][Dd][Ee][Rr]_[Ll][Oo][Gg][\t]*$"

This command is relevant for **ExpertGUI**. The server add this prefix to every burst: "Reading LogFile...\n".
So, when **ExpertGUI** in **ReadTcpData** slot recognize "Reading LogFile...", prints the burst received in the **Expert Mode QTextEdit** and in **GeneralLog QTextEdit**.

15.1.6 check_drv_assoc

Name:
check_drv_assoc

Syntax:
check_drv_assoc

Meaning:
this command check the association between the drivers serial number found in the SerialDrvLog.txt file and the real situation.

Server regular expression: "^[Cc][Hh][Ee][Cc][Kk]_[Dd][Rr][Vv]_[Aa][Ss][Ss][Oo][Cc][\t]*$"

This command is relevant for ExpertGUI. Is very important because the server expects a reaction by the ExpertGUI and the command is blocking for the ExpertGUI.
The relevant output is: "^Check Drv Assoc:".

15.1.7 check_par_assoc

Name:
check_par_assoc

Syntax:
check_par_assoc

Meaning:
this command check the association between the drivers parameters found in the FileParLog.txt file and the real situation.

Server regular expression: "^[Cc][Hh][Ee][Cc][Kk]_[Pp][Aa][Rr]_[Aa][Ss][Ss][Oo][Cc][\t]*$"

This command is relevant for ExpertGUI. Is very important because the server expects a reaction by the ExpertGUI and the command is blocking for the ExpertGUI.

The relevant output is: "^Check Par Assoc:".

15.1.8 check_encode_assoc

Name:
check_encode_assoc

Syntax:
check_encode_assoc

Meaning:
this command check the association between the encoder values contained in the EncoderLog.txt file and the real situation.

Server regular expression: "^[Cc][Hh][Ee][Cc][Kk]_[Ee][Nn][Cc][Oo][Dd][Ee]_[Aa][Ss][Ss][Oo][Cc][\t]*$"

This command is relevant for **ExpertGUI**. Is very important because the server expects a reaction by the **ExpertGUI** and the command is blocking for the **ExpertGUI**.
The relevant output is: "^Check Encode Assoc:".

15.1.9 connect

Name:
connect

Syntax:
connect absoluteprogrammerpath

Meaning:
this command tries to connect the server with the programmer indicated by

"absoluteprogrammerpath".

Server regular expression:
"^[Cc][Oo][Nn][Nn][Ee][Cc][Tt][\t][A-z0-9/\\]{1,100}[\t]*$"

This command is relevant either for **ExpertGUI** (**ReadTcpData** slot) either for **UserGUI** (**AuthomaticSettings** function).

Relevant output:
"^Connect:"

15.1.10 help

Name:
help

Syntax:
help

Meaning:
prints the list of the commands that the server can execute.

Server regular expression:
"^[Hh][Ee][Ll][Pp][\t]*$"

The output is relevant for ExpertGUI but not recognition by ReadTcpData is performed. The output is simply printed in the Expert Mode QTextEdit.

15.1.11 get_par

Name:

get_par

Syntax:

get_par drvnum

Meaning:

prints the parameters of the driver indicated by drvnum. Parameters are: max_vel, velhome, acceleration, deceleration, phase_current, analog_output0.

Server regular expression:

"^[Gg][Ee][Tt]_[Pp][Aa][Rr][\t][0-9]{1,2}[\t]*$"

The output is relavant for **ExpertGUI**. This is a fundamental output because the **ExpertGUI** use it for many purposes.

Function called by the server: **GetPar** or **SendFailedGetPar**.

Relevant output (recognized by ReadTcpData slot):

"^getpar[ ][0-9]{1,2}[ ]((MaxVel)|(VelHome)|(Acceleration)|(Deceleration)|(PhaseCurrent)|(AnalogOutput0))[ ]-{0,1}[0-9]{1,20}$"

15.1.12 check_position

Name:

check_position

Syntax:

check_position drvnum

Meaning:

check if the actual position of the driver indicated by drvnum correspond with the one indicated by the encoder (analog_input0).

Server regular expression:

"^[Cc][Hh][Ee][Cc][Kk]_[Pp][Oo][Ss][Ii][Tt][Ii][Oo][Nn][\t][0-9]{1,2}[\t]*$"

The output is relevant either for ExpertGUI, either for UserGUI.

Relevant output:

"^get_pos_status[ ][0-9]{1,2}[ ]-{0,1}[0-9]{1,5}"

15.1.13 set_par

Name:

set_par

Syntax:

set_par drvnum max_vel acceleration deceleration PhaseCurrent AnalogOutput0

Meaning:

set max_vel (Each unity of maxvel correspond to 0.25rpm), acceleration and deceleration (Each unity of acceleration and deceleration correspond to 1rpm/s), phasecurrent and AnalogOutput0 of the driver specified with drvnum.

Server regular expression:

"^[Ss][Ee][Tt]_[Pp][Aa][Rr]([\t]+[0-9]{1,5})([\t]+[0-9]{1,5})([\t]+-{0,1}[0-9]{1,5})([\t]+[0-9]{1,5})([\t]+[0-9]{1,5})([\t]+[0-9]{1,5})([\t]+[0-9]{1,5})[\t]*$"

If no error occurred, this command called a GetPar (or SendFailedGetPar) function, so the relavant output is the same of the "^[Gg][Ee][Tt]_[Pp][Aa][Rr][\t][0-9]{1,2}[\t]*$" command.

15.1.14 get_mov_par

Name:

get_mov_par

Syntax:

get_mov_par drvnum

Meaning:

this command obtains the actual position and the AnalogInput0 values of the driver indicated by drvnum.

Server regular expression:

"^[Gg][Ee][Tt]_[Mm][Oo][Vv]_[Pp][Aa][Rr][\t]+[0-9]{1,2}[\t]*$"

The output is relevant for **ExpertGUI**. For executing the command, the server launches **GetMovePar** function, so the relevant output is:

"^get_mov_par[ ][0-9]{1,2}[ ]((CurrentPosition)|(AnalogInput0))[ ]-{0,1}[0-9]{1,20}$"

15.1.15 homing

Name:

homing

Syntax:

homing drvnum

Meaning:

executes the homing procedure for the driver indicated by drvnum.

Server regular expression:

"^[Hh][Oo][Mm][Ii][Nn][Gg][\t]+[0-9]{1,2}[\t]*$"

The output is relevant for ExpertGUI. At the end of the command the server executes GetMovePar function, so the relevant output is:

"^get_mov_par[ ][0-9]{1,2}[ ]((CurrentPosition)|(AnalogInput0))[ ]-{0,1}[0-9]{1,20}$"

15.1.16 encode

Name:
encode

Syntax:
encode drvnum

Meaning: this command start the encoding procedure for the driver indicated by drvnum.
Server regular expression: "^[Ee][Nn][Cc][Oo][Dd][Ee][\t]+[0-9]{1,2}[\t]*$"
The output is relevant only for ExpertGUI. This command can be sent only from expert mode. No relevant output is sent from server to client.

15.1.17 move_to

Name: move_to

Syntax: move_to drvnum targetposition

Meaning: this command set to targetposition the target position of the driver indicated by **drvnum**.

Server regular expression: "^[Mm][Oo][Vv][Ee]_[Tt][Oo][\t]+[0-9]{1,2}([\t]+-{0,1}[0-9]{1,10})[\t]*$"

The output is relevant only for **ExpertGUI**. At the end of the command the server executes **GetMovePar** function, so the relevant output is:
"^get_mov_par[ ][0-9]{1,2}[ ]((CurrentPosition)|(AnalogInput0))[ ]-{0,1}[0-9]{1,20}$"

15.1.18 get_all_parameter

Name:

get_all_parameter

Syntax:

get_all_parameter

Meaning:

this command is equivalent to execute get_move_par drvnum, get_par drvnum and check_position drvnum for the all drivers.

Server regular expression: "^[Gg][Ee][Tt]_[Aa][Ll][Ll]_[Pp][Aa][Rr][Aa][Mm][Ee][Tt][Ee][Rr][\t]*$

The output is relevant either for **ExpertGUI** and for **UserGUI**. In order to perform the command, the server execute **GetMovePar**, **GetPar** and **CheckPositionEncoderSingle** function.

Relevant output for ExpertGUI:

"^get_mov_par[ ][0-9]{1,2}[ ]((CurrentPosition)|(AnalogInput0))[ ]-{0,1}[0-9]{1,20}$"

"^getpar[ ][0-9]{1,2}[ ]((MaxVel)|(VelHome)|(Acceleration)|(Deceleration)|(PhaseCurrent)|(AnalogOutput0))[ ]-{0,1}[0-9]{1,20}$"

"^get_pos_status[ ][0-9]{1,2}[ ]-{0,1}[0-9]{1,5}"

Relevant output for UserGUI:

"^get_pos_status[ ][0-9]{1,2}[ ]-{0,1}[0-9]{1,5}"

15.1.19 homing_mult

Name:

homing_mult

Syntax:

homing_mult drvnum1 drvnum2 drvnum3 drvnum....

Meaning:

this command execute the homing procedure for the driver indicated by drvnum1, drvnum2, drvnum3, drvnum....

Server regular expression:

"^[Hh][Oo][Mm][Ii][Nn][Gg]_[Mm][Uu][Ll][Tt]([\t]+[0-9]{1,2}){1,14}[\t]*$"

The output is relevant for ExpertGUI. At the end of the command the server executes GetMovePar function, so the relevant output is:

"^get_mov_par[ ][0-9]{1,2}[ ]((CurrentPosition)|(AnalogInput0))[ ]-{0,1}[0-9]{1,20}$"

15.1.20 moveto_mult

Name:

moveto_mult

Syntax:

moveto_mult targetposition drvnum1 drvnum2 drvnum3 drvnum....

Meaning:

this command set the target position to targetposition of the drivers indicated by drvnum1, drvnum2, drvnum3, drvnum....

Server regular expression:

"^[Mm][Oo][Vv][Ee][Tt][Oo]_[Mm][Uu][Ll][Tt]([\t]+-{0,1}[0-9]{1,12})([\t]+[0-9]{1,2}){1,14}[\t]*$"

The output is relevant only for ExpertGUI. At the end of the command the server

executes GetMovePar function, so the relevant output is:
"^get_mov_par[ ][0-9]{1,2}[ ]((CurrentPosition)|(AnalogInput0))[ ]-{0,1}[0-9]{1,20}$"

15.1.21 setmult_par

Name:
setmult_par

Syntax:
setmult_par max_vel acceleration deceleration PhaseCurrent AnalogOutput0 drvnum1 drvnum2 drvnum3 drvnum....
Meaning:
set max_vel (Each unity of maxvel correspond to 0.25rpm), acceleration and deceleration (Each unity of acceleration and deceleration correspond to 1rpm/s), phasecurrent and AnalogOutput0 of the drivers specified with drvnum.

Server regular expression:
"^[Ss][Ee][Tt][Mm][Uu][Ll][Tt]_[Pp][Aa][Rr]([\t]+[0-9]{1,5})([\t]+-{0,1}[0-9]{1,5})([\t]+[0-9]{1,5})([\t]+[0-9]{1,5})([\t]+[0-9]{1,5})([\t]+[0-9]{1,5})([\t]+[0-9]{1,5}){1,20}[\t]*$"
If no error occurred, this command called a GetPar (or SendFailedGetPar) function, so the relavant output is the same of the "^[Gg][Ee][Tt]_[Pp][Aa][Rr][\t][0-9]{1,2}[\t]*$" command.

15.1.22 check_internal_status

Name:
check_internal_status

Syntax:
check_internal_status

Meaning:

this command retrieves the content of the GeneralStatus struct.

If GeneralStatus.assoc_file_status == 1 means the user has already executed the procedure to check

the association between the serial numbers contained in the SerialDrvLog.txt file and the real situation.

If GeneralStatus.par_file_status = 1 means the user has already executed the procedure to check

the association between the parameters contained in the FileParLog.txt file and the real situation.

If GeneralStatus.encoder_file_status = 1 means the user has already executed the procedure to check

the association between the encoder values contained in the EncoderLog.txt file and the real situation.

Server regular expression:

"^[Cc][Hh][Ee][Cc][Kk]_[Ii][Nn][Tt][Ee][Rr][Nn][Aa][Ll]_[Ss][Tt][Aa][Tt][Uu][Ss]
[\t]*$"

The output is relevant only for ExpertGUI.

Relevant output:

"^InternalStatusSerial: [01]$"

"^InternalStatusParameter: [01]$"

"^InternalStatusEncoder: [01]$"

15.1.23 load_encoder_from_file

Name:

load_encoder_from_file

Syntax:

load_encoder_from_file

Meaning:
this command gets the encoding parameters for each drivers from the EncoderLog.txt file and use it to accomplished the check_position command.

Server regular expression: "^[Ll][Oo][Aa][Dd]_[Ee][Nn][Cc][Oo][Dd][Ee][Rr]_[Ff][Rr][Oo][Mm]_[Ff][Ii][Ll][Ee][\t]*$"
The output is relevant only for ExpertGUI.

Relevant output:
"^Loading encoder values from file:"

15.1.24 read_actual_encoder_values

Name:
read_actual_encoder_values

Syntax:
read_actual_encoder_values

Meaning:
this command prints the actual encoding parameters that will be used to accomplished the check_position command.

Server regular expression: "^[Rr][Ee][Aa][Dd]_[Aa][Cc][Tt][Uu][Aa][Ll]_[Ee][Nn][Cc][Oo][Dd][Ee][Rr]_[Vv][Aa][Ll][Uu][Ee][Ss][\t]*$"
The output is relevant only for **ExpertGUI**.

Relevant output for ExpertGUI:
"^Loading encoder values from file:"

N.B. The same relevant output of load_encoder_from_file.

15.1.25 device_list

Name:

device_list

Syntax:

device_list

Meaning:

this command prints the device contained in /dev

Server regular expression:

"^[Dd][Ee][Vv][Ii][Cc][Ee]_[Ll][Ii][Ss][Tt][\t]*$"

The output is relevant only for ExpertGUI.

Relevant output for ExpertGUI:

"^Device list:"

15.1.26 Others significant Server Program output:

"^Check position warning!": relevant only for ExpertGUI. This output is sent when the server receives a check_position command but the user has not sent load_encoder_from_file yet.

"^Welcome:": relevant only for ExpertGUI. The server sends "^Welcome:" when it accepts a connection with a client.

16 LDAP

The point is to defined the requirements to implement an **LDAP** server to be used with the **UserGUI**.
Why **LDAP**? Because is a simple and cheap way to manage user. There a lot of wonderful examples of it. Here, i'll report the procedure to queries the database.

The **UserGUI** may have a system to identified the users. And the administrator have to manage it in a simple and safe way.

It is not important the **LDAP** implementation or the **LDAP** tree structure. You can change the values defined below and the procedure will work the same.
The procedure search a **posixAccount** with the uid indicated by user and retrieve the password. No encryption protocol are required so the procedure expects a clear password.
Then, it compare the password received with the password used by pass and return the match or the mismatch.
Is very important to underline that this way to manage the users is totally unsafe respect the expert users. Indeed, the procedure queries the server with the administrator password and retrieves the user password. But his purpose is to manage the user account in a totally safe environment. So, the programmer or the administrator con add a **posixAccount** remotely, then the user can enter the **UserGUI** expert mode or the movimentation simply.

I used the **phpldapadmin** to graphically manage LDAP. You can find useful guides in **www.digitalocean.com**: you can googling writing **"digitalocean LDAP"** or with every keywords you want.

If you want to use UserGUI without LDAP, you can insert username “admin” and password “admin”.
This is the main procedure:

```
#define LDAP_DEPRECATED 1

#include <ldap.h>

#define LDAP_HOST "127.0.0.1"
#define LDAP_PORT 389
#define LDAP_DN "cn=admin, dc=elinp, dc=com"
#define LDAP_PW "fantinodivaren"
#define LDAP_BASE_DN "ou=wp09,dc=elinp,dc=com"

//return status.
//status = -1 : unable to query the LDAP server.
//status = 0: password mismath/the user does not exist.
//status = 1: okay.
//Input argument: the username (char* user) and the password (char* pass)
//to check.
int EnablePasswordExpertMode::ldap_authentication(char* user, char* pass)
{
   int status = -1;

   LDAP *ldap;
   LDAPMessage *answer, *entry;
   BerElement *ber;

   int  result;
   int  auth_method    = LDAP_AUTH_SIMPLE;
   int  ldap_version   = LDAP_VERSION3;
   char *ldap_host     = LDAP_HOST;
   int   ldap_port     = LDAP_PORT;
   char *ldap_dn       = LDAP_DN;
   char *ldap_pw       = LDAP_PW;
```

```
  char *base_dn       = LDAP_BASE_DN;

        // The search scope must be either LDAP_SCOPE_SUBTREE or
LDAP_SCOPE_ONELEVEL
  int  scope          = LDAP_SCOPE_SUBTREE;
  // The search filter, "(objectClass=*)" returns everything. Windows can return
  // 1000 objects in one search. Otherwise, "Size limit exceeded" is returned.
  //~ char *filter        = "(&(objectClass=user)(sAMAccountName=frank4dd))";
  QString user_tmp(user);

  QString filter_tmp = "(&(objectClass=posixAccount)(uid=" + user_tmp + "))";

  QByteArray tmp = filter_tmp.toLatin1();
  char *filter       = tmp.data();

  // The attribute list to be returned, use {NULL} for getting all attributes
  //char *attrs[]       = {"memberOf", NULL};
  char *attrs[]       = {"userPassword"};
  // Specify if only attribute types (1) or both type and value (0) are returned
  int  attrsonly      = 0;
  // entries_found holds the number of objects found for the LDAP search
  int  entries_found  = 0;
  // dn holds the DN name string of the object(s) returned by the search
  char *dn            = "";
  // attribute holds the name of the object(s) attributes returned
  char *attribute     = "";
  // values is  array to hold the attribute values of the object(s) attributes
  char **values;
  // i is the for loop variable to cycle through the values[i]
  int  i              = 0;
```

```
  /* First, we print out an informational message. */
  //printf( "Connecting to host %s at port %d...\n\n", ldap_host, ldap_port );

  /* STEP 1: Get a LDAP connection handle and set any session preferences. */
  /* For ldaps we must call ldap_sslinit(char *host, int port, int secure) */
  if ( (ldap = ldap_init(ldap_host, ldap_port)) == NULL ) {
   //perror( "ldap_init failed" );
   return status;
  } else {
   //printf("Generated LDAP handle.\n");
  }

  /* The LDAP_OPT_PROTOCOL_VERSION session preference specifies the client */
  /* is an LDAPv3 client. */
            result = ldap_set_option(ldap, LDAP_OPT_PROTOCOL_VERSION,
&ldap_version);

  if ( result != LDAP_OPT_SUCCESS ) {
     ldap_perror(ldap, "ldap_set_option failed!");
     return status;
  } else {
   //printf("Set LDAPv3 client version.\n");
  }

  /* STEP 2: Bind to the server. */
  // If no DN or credentials are specified, we bind anonymously to the server */
  // result = ldap_simple_bind_s( ldap, NULL, NULL );
  result = ldap_simple_bind_s(ldap, ldap_dn, ldap_pw );

  if ( result != LDAP_SUCCESS ) {
   //fprintf(stderr, "ldap_simple_bind_s: %s\n", ldap_err2string(result));
```

```
    return status;
  } else {
    //printf("LDAP connection successful.\n");
  }

  /* STEP 3: Do the LDAP search. */
  result = ldap_search_s(ldap, base_dn, scope, filter,
                         attrs, attrsonly, &answer);

  if ( result != LDAP_SUCCESS ) {
    //fprintf(stderr, "ldap_search_s: %s\n", ldap_err2string(result));
    return status;
  } else {
    //printf("LDAP search successful.\n");
  }

  /* Return the number of objects found during the search */
  entries_found = ldap_count_entries(ldap, answer);
  if ( entries_found == 0 ) {
    //fprintf(stderr, "LDAP search did not return any data.\n");

    ////////////////////
    //Very important to return 0 (incorrect username and/or password) in case of invalid
username
    status = 0;
    ////////////////////

    return status;
  } else {
    //printf("LDAP search returned %d objects.\n", entries_found);
  }
```

```
/* cycle through all objects returned with our search */
for ( entry = ldap_first_entry(ldap, answer);
      entry != NULL;
      entry = ldap_next_entry(ldap, entry)) {

  /* Print the DN string of the object */
  dn = ldap_get_dn(ldap, entry);
  //printf("Found Object: %s\n", dn);

  // cycle through all returned attributes
  for ( attribute = ldap_first_attribute(ldap, entry, &ber);
        attribute != NULL;
        attribute = ldap_next_attribute(ldap, entry, ber)) {

    /* Print the attribute name */
    //printf("Found Attribute: %s\n", attribute);
    if ((values = ldap_get_values(ldap, entry, attribute)) != NULL) {

      /* cycle through all values returned for this attribute */
      for (i = 0; values[i] != NULL; i++) {

        QString passtocheck_tmp(pass);
        QString passtocheckLDAP_tmp(values[i]);

        if (passtocheck_tmp == passtocheckLDAP_tmp)
        {
           status = 1;
           return status;
        }
        else
```

```
        {
            status = 0;
            return status;
        }
        /* print each value of a attribute here */
        //printf("%s: %s\n", attribute, values[i] );
       }
       ldap_value_free(values);
     }
    }
    ldap_memfree(dn);
  }

  ldap_msgfree(answer);
  ldap_unbind(ldap);

  return status;

}
```

17 Server Global Variables

Now, it will be listed the **Server Global Variables**. All these variables are defined in **CommandExecutor.c** and declared in **CommandExecutor.h** or **Utils.h**

17.1 bool loading_encoder_from_file_okay

Declaration: extern bool loading_encoder_from_file_okay; (CommandExecutor.h)

Definition: bool loading_encoder_from_file_okay; (CommandExecutor.c)

Inizialization: loading_encoder_from_file_okay = 0; (Main.c at the beginning of the main function).

Meaning: Records if the user has already loaded the encoding parameters from file EncoderLog.txt: when it is equal to 0 the user have not loaded the encoder parameters yet. When it is equal to 1 the user
have done it.

Uses:

Main.c, main function

```
[...]
if (loading_encoder_from_file_okay == 1 && STATE_CONNECT == 1)
[...]
```

CommandExecutor.c, CheckPositionEncoderSingleWarning function

```
[...]
if (loading_encoder_from_file_okay == 0)
        output_module->Output("Check position warning! You have to press the button Load Encoder From File in General tab or you have to digit load_encoder_from_file command in order to accomplished the check position procedure in a consistent way!\n");
[...]
```

CommandExecutor.c, LoadEncoderFromFile function

```
[...]
        loading_encoder_from_file_okay = 1;
[...]

[...]
```

loading_encoder_from_file_okay = 0;

[...]

17.2 file_check_status GeneralStatus

Declaration: typedef struct{
bool assoc_file_status;
bool par_file_status;
bool encoder_file_status;
} file_check_status; (CommandExecutor.h)

extern file_check_status GeneralStatus; (CommandExecutor.h)

Definition: file_check_status GeneralStatus; (CommandExecutor.c)

Inizialitation:

GeneralStatus.assoc_file_status = 0;

GeneralStatus.par_file_status = 0;

GeneralStatus.encoder_file_status = 0;

Meaning:

Set to 0 the struct GeneralStatus.

If GeneralStatus.assoc_file_status == 1 means the user has already executed the procedure to check the association between the serial numbers contained in the **SerialDrvLog.txt** file and the real situation.

If GeneralStatus.par_file_status = 1 means the user has already executed the procedure to check the association between the parameters contained in the FileParLog.txt file and the real situation.

If GeneralStatus.encoder_file_status = 1 means the user has already executed the

procedure to check the association between the encoder values contained in the **EncoderLog.txt** file and the real situation.

Uses:
CommandExecutor.c, CheckDrvAssoc function.
CommandExecutor.c, CheckParAssoc function.
CommandExecutor.c, CheckEncodeAssoc function.
Main.c, main function, check_internal_status command.

17.3 EncoderStruct EncoderArrayValue[MAXIMUM_DRIVER];

Declaration: typedef struct {
int drv_num;
double slope;
double intercept;
double coefficient;
} EncoderStruct; (Utils.h)

extern EncoderStruct EncoderArrayValue[MAXIMUM_DRIVER]; (CommandExecutor.h)

Definition: EncoderStruct EncoderArrayValue[MAXIMUM_DRIVER]; (CommandExecutor.c)

Inizialization:

```
for (int enc = 0; enc < MAXIMUM_DRIVER; enc++)
{
        EncoderArrayValue[enc].drv_num = -1;
        EncoderArrayValue[enc].slope = -1;
        EncoderArrayValue[enc].intercept = -1;
        EncoderArrayValue[enc].coefficient = -1;
```

} (At the beginning of the main function of Main.c)

Meaning: this array records the values used to accomplished the check_position procedure for each driver.

Uses:
CommandExecutor.c in CheckEncodeAssoc, CheckPositionEncoderSingle, CheckPositionEncoderSingleWarning, CheckPositionEncoderToAll, LoadEncoderFromFile, ReadActualEncoderValue function.

17.4 Parameters arrays

ParameterStruct ParameterArray[MAXIMUM_DRIVER];
ParameterStruct ParameterArrayParagorn[MAXIMUM_DRIVER];
ParameterStruct ParameterArrayTmp[MAXIMUM_DRIVER];

Declaration: typedef struct {
int drv_num;
int max_vel;
int16_t vel_home;
int acceleration;
int deceleration;
int phase_current;
int analog_output0;
} ParameterStruct; (Utils.h)

extern ParameterStruct ParameterArray[MAXIMUM_DRIVER];
extern ParameterStruct ParameterArrayParagorn[MAXIMUM_DRIVER];
extern ParameterStruct ParameterArrayTmp[MAXIMUM_DRIVER]; (CommandExecutor.h).

Definition:

ParameterStruct ParameterArray[MAXIMUM_DRIVER];

ParameterStruct ParameterArrayParagorn[MAXIMUM_DRIVER];

ParameterStruct ParameterArrayTmp[MAXIMUM_DRIVER]; (CommandExecutor.c)

Initialization:

```
for (int par = 0; par < MAXIMUM_DRIVER; par ++)
{
        ParameterArray[par].drv_num = -1;
        ParameterArray[par].max_vel = -1;
        ParameterArray[par].vel_home = -1;
        ParameterArray[par].acceleration = -1;
        ParameterArray[par].deceleration = -1;
        ParameterArray[par].phase_current = -1;
        ParameterArray[par].analog_output0 = -1;

        ParameterArrayParagorn[par].drv_num = -1;
        ParameterArrayParagorn[par].max_vel = -1;
        ParameterArrayParagorn[par].vel_home = -1;
        ParameterArrayParagorn[par].acceleration = -1;
        ParameterArrayParagorn[par].deceleration = -1;
        ParameterArrayParagorn[par].phase_current = -1;
        ParameterArrayParagorn[par].analog_output0 = -1;

        ParameterArrayTmp[par].drv_num = -1;
        ParameterArrayTmp[par].max_vel = -1;
        ParameterArrayTmp[par].vel_home = -1;
        ParameterArrayTmp[par].acceleration = -1;
        ParameterArrayTmp[par].deceleration = -1;
        ParameterArrayTmp[par].phase_current = -1;
        ParameterArrayTmp[par].analog_output0 = -1;
```

} (At the beginning of the main function of Main.c)

Meaning: These arrays are useful to perform several operations relatively to the drivers parameters collected from FileParLog.txt or from the drivers. In particular, they are used to compare the
parameters obtained by the drivers with the parameters read from FileParLog.txt .

Uses: CheckParAssoc function in CommandExecutor.c .

17.5 Serial Numbers arrays

unsigned int SerialNumberArray[MAXIMUM_DRIVER];
unsigned int SerialNumberArrayParagorn[MAXIMUM_DRIVER];
unsigned int SerialNumberArrayTmp[MAXIMUM_DRIVER];

Declaration:
extern unsigned int SerialNumberArray[MAXIMUM_DRIVER];
extern unsigned int SerialNumberArrayParagorn[MAXIMUM_DRIVER];
extern unsigned int SerialNumberArrayTmp[MAXIMUM_DRIVER]; (CommandExecutor.h)

Definition:
unsigned int SerialNumberArray[MAXIMUM_DRIVER];
unsigned int SerialNumberArrayParagorn[MAXIMUM_DRIVER];
unsigned int SerialNumberArrayTmp[MAXIMUM_DRIVER]; (CommandExecutor.c)

Initialization:
//Set to zero the arrays that will contain the serial number of the drivers
bzero (SerialNumberArray, MAXIMUM_DRIVER);
bzero (SerialNumberArrayParagorn, MAXIMUM_DRIVER);
bzero (SerialNumberArrayTmp, MAXIMUM_DRIVER); (At the beginning of the main

function of Main.c)

Meaning: These arrays are useful to perform several operations relatively to the drivers serial numbers collected from SerialDrvLog.txt or from the drivers. In particular, they are used to compare the
serial numbers obtained by the drivers with the serial numbers read from FileParLog.txt .

Uses: CheckDrvAssoc function in CommandExecutor.c .

18 Description of the most relevant functions

Registers description in **DefineGeneral.h** .

18.1 DefineGeneral.h

//Definition of the addresses used in the program.
//The address of the registers could be checked using the manual of the drivers,
//the address of the variables could be checked at the beginning of the software firmware of the drivers.
//N.B. The address used for the modbus communication must be the physical address minus one.
//Ex. If the physical address of AnalogInput(0) is 0xA203 you must use 0x202 for the modbus communication.

#define	*StopAddr*	*0xA000*
#define	*StatusAddr*	*0xA102*
#define	*ControlModeAddress*	*0xA104*
#define	*CurrentPositionAddress*	*0xA10B*
#define	*TargetPositionAddress*	*0xA301*

#define	*AddressCounterA*	*0xA10F*
#define	*AddressMaxVel*	*0xA107*
#define	*AddressVelHome*	*0xA00A*
#define	*AddressAcceleration*	*0xA109*
#define	*AddressDeceleration*	*0xA10A*
#define	*AddressRefVal*	*0xA300*
#define	*AddressAnalogInput0*	*0xA202*
#define	*AddressAnalogOutput0*	*0xA204*
#define	*SerialNumberAddress*	*0x9D05*
#define	*PhaseCurrentAddress*	*0xA103*
#define	*StatusStateAddress*	*0xA005*
#define	*RequestStateAddress*	*0xA008*
#define	*count_TargetPosAddress*	*0xA003*

18.2 Low level function

All these function are declared in **DriverFunction.h** file and defined in **DriverFunction.c** .
The following functions interact with the drivers using the functions defined in **BasicModbusLibrary.c**, a simplified version of **libmodbus v.3.0.6** by **Stephane Raimbault**.
These functions does not set the slave number, so you have to set it before calling them.

Usually, the following functions call only modbus_read_register or modbus_write_register and modbus_strerror.

N.b. The return value and the argument of the functions depend to the dimension of the driver register being written.

N.b. Register address' are defined in DefineGeneral.h .

N.b. In each function is very important usleep(SLEEPMODBUS) in order to give the programmer the time to send the data.
N.b. Setting functions return the error status.

General information: all the function defined in DriverFunctio.c are unblocking. So, if the function failed, an error status is setted.
Blocking procedures are present in CommandExecutor.c functions.

N.b. In register used I reported the address of the register used in modbus communication. To obtain the phycal address you have to sum one to it.

Precondition: all these function (excluding the Connect) are designed assuming that ctx is a valid modbus_t resource. If this condition is not respected the result could be a segmentation fault.
The program calls these functions only if S**TATE_CONNECT == 1.**
It is relevant to point out that the segmentation fault does not happened if the **modbus_new_rtu** function is correctly executed. So, if you connected the PC to the driver but at a later time you switch off the system, the server does not crash because **ctx** is a valid resource yet. The communication with the programmer will be impossible but the resource is still valid.

Function list:

1. modbus_t **Connect**(modbus_t *ctx, bool* STATE_CONNECT, char* path);*

*2. unsigned int **ReadSerialNumber**(modbus_t *ctx, int* rc_arg);*

*3. int **SetMaxVel** (modbus_t *ctx, uint16_t max_vel, string header);*

*4. int **SetVelHome** (modbus_t *ctx, int16_t vel_home, string header);*

*5. int **SetAcceleration** (modbus_t *ctx, uint16_t acceleration, string header);*

*6. int **SetDeceleration** (modbus_t *ctx, uint16_t deceleration, string header);*

*7. int **SetPhaseCurrent** (modbus_t *ctx, uint16_t phase_current, string header);*

*8. int **SetAnalogOutput0** (modbus_t *ctx, uint16_t analog_output0, string header);*

*9. int **SetStatusState**(modbus_t *ctx, uint16_t status_state, string header);*

*10. int **SetRequestState**(modbus_t *ctx, uint16_t request_state, string header);*

*11. int **SetTargetPosition**(modbus_t *ctx, unsigned int moveto_val, string header);*

*12. int **SetCountTargetPosition**(modbus_t *ctx, unsigned int moveto_val, string header);*

*13. uint16_t **ReadMaxVel**(modbus_t *ctx, int* rc_arg, string header);*

*14. int16_t **ReadVelHome**(modbus_t *ctx, int* rc_arg, string header);*

*15. uint16_t **ReadAcceleration**(modbus_t *ctx, int* rc_arg, string header);*

*16. uint16_t **ReadDeceleration**(modbus_t *ctx, int* rc_arg, string header);*

*17. uint16_t **ReadPhaseCurrent**(modbus_t *ctx, int* rc_arg, string header);*

*18. uint16_t **ReadAnalogOutput0**(modbus_t *ctx, int* rc_arg, string header);*

*19. uint16_t **ReadStatusState**(modbus_t *ctx, int* rc_arg, string header);*

*20. uint16_t **ReadAnalogInput0**(modbus_t *ctx, int* rc_arg, string header);*

*21. int **ReadCurrentPosition**(modbus_t *ctx, int* rc_arg, string header);*

18.2.1 Connect

modbus_t Connect(modbus_t *ctx, bool* STATE_CONNECT, char* path)*

Meaning:

Function used to connect the server with the programmer. STATE_CONNECT is a global_variable that records if the operation is successfully done.
path is the absolute path of the device (ex. "/dev/ttyUSB0").

Driver interaction:

ctx = modbus_new_rtu(path, 9600, 'N', 8, 1);
string tmp_string(modbus_strerror(errno));
modbus_free(ctx);
modbus_set_slave(ctx, DEFAULT_SLAVE); //See DefineGeneral.h for DEFAULT_SLAVE values.

Relevant address':

-

18.2.2 ReadSerialNumber

*unsigned int ReadSerialNumber(modbus_t *ctx, int* rc_arg)*

Meaning:

Function used to read the serial number of the driver.
rc_arg will contain the status of the operation.

Driver interaction:

rc = modbus_read_registers(ctx, SerialNumberAddress, 2, &data[0]);

tmp_errno = modbus_strerror(errno);

Relevant address':

SerialNumberAddress (0x9D05)

18.2.3 SetMaxVel

*int SetMaxVel (modbus_t *ctx, uint16_t max_vel, string header)*

Meaning:

Function used to set the max_vel parameter.

header is the prefix of every output printed by the function.

The return value is the error status.

Driver interaction:

rc = modbus_write_registers(ctx, AddressMaxVel, 1, &data[0]);

tmp_errno = modbus_strerror(errno);

Relevant address':

AddressMaxVel (0xA107)

18.2.4 SetVelHome

*int SetVelHome (modbus_t *ctx, int16_t vel_home, string header)*

Meaning:

Function used to set the vel_home parameter.

header is the prefix of every output printed by the function.

The return value is the error status.

Driver interaction:

rc = modbus_write_registers(ctx, AddressVelHome, 1, &data[0]);

tmp_errno = modbus_strerror(errno);

Relevant address':

AddressVelHome (0xA00A)

18.2.5 SetAcceleration

*int SetAcceleration (modbus_t *ctx, uint16_t acceleration, string header)*

Meaning:

Function used to set the acceleration parameter.

header is the prefix of every output printed by the function.

The return value is the error status.

Driver interaction:

rc = modbus_write_registers(ctx, AddressAcceleration, 1, &data[0]);

tmp_errno = modbus_strerror(errno);

Relevant address':

AddressAcceleration (0xA109)

18.2.6 SetDeceleration

*int SetDeceleration (modbus_t *ctx, uint16_t deceleration, string header)*

Meaning:

Function used to set the acceleration parameter.

header is the prefix of every output printed by the function.

The return value is the error status.

Driver Interaction:

rc = modbus_write_registers(ctx, AddressDeceleration, 1, &data[0]);

tmp_errno = modbus_strerror(errno);

Relevant address':

AddressDeceleration (0xA10A)

18.2.7 SetPhaseCurrent

*int SetPhaseCurrent (modbus_t *ctx, uint16_t phase_current, string header)*

Meaning:

Function used to set the phase_current parameter.

header is the prefix of every output printed by the function.

The return value is the error status.

Driver interaction:

rc = modbus_write_registers(ctx, PhaseCurrentAddress, 1, &data[0]);

tmp_errno = modbus_strerror(errno);

Relevant address':

PhaseCurrentAddress (0xA103)

18.2.8 SetAnalogOutput0

*int SetAnalogOutput0 (modbus_t *ctx, uint16_t analog_output0, string header)*

Meaning:

Function used to set the analog_output0 parameter.

header is the prefix of every output printed by the function.

The return value is the error status.

Driver interaction:

rc = modbus_write_registers(ctx, AddressAnalogOutput0, 1, &data[0]);

tmp_errno = modbus_strerror(errno);

Relevant address':

AddressAnalogOutput0 (0xA204)

18.2.9 SetStatusState

*int SetStatusState (modbus_t *ctx, uint16_t status_state, string header)*

Meaning:

Function used to set the status_state register.

header is the prefix of every output printed by the function.

The return value is the error status.

Driver interaction:

rc = modbus_write_registers(ctx, StatusStateAddress, 1, &data[0]);

tmp_errno = modbus_strerror(errno);

Relevant address':

StatusStateAddress (0xA005)

18.2.10 SetRequestState

*int SetRequestState (modbus_t *ctx, uint16_t request_state, string header)*

Meaning:

Function used to set the request_state register.

header is the prefix of every output printed by the function.

The return value is the error status.

Driver interaction:

rc = modbus_write_registers(ctx, RequestStateAddress, 1, &data[0]);

tmp_errno = modbus_strerror(errno);

Relevant address':

RequestStateAddress (0xA008)

18.2.11 SetTargetPosition

*int SetTargetPosition(modbus_t *ctx, unsigned int moveto_val, string header)*

Meaning:

Function used to set the target_position register.

header is the prefix of every output printed by the function.

The return value is the error status.

Driver interaction:

rc = modbus_write_registers(ctx, TargetPositionAddress, 2, data);

tmp_errno = modbus_strerror(errno);

Relevant address':

TargetPositionAddress (0xA301)

18.2.12 SetCountTargetPosition

*int SetCountTargetPosition(modbus_t *ctx, unsigned int moveto_val, string header)*

Meaning:

Function used to set the target_position register.

header is the prefix of every output printed by the function.

The return value is the error status.

Driver interaction:

rc = modbus_write_registers(ctx, count_TargetPosAddress, 2, data);

tmp_errno = modbus_strerror(errno);

Relevant address':

count_TargetPosAddress (0xA003)

18.2.13 ReadMaxVel

*uint16_t ReadMaxVel(modbus_t *ctx, int* rc_arg, string header)*

Meaning:

Function used to read the max_vel of the driver.

rc_arg will contain the status of the operation.

Driver interaction:

rc = modbus_read_registers(ctx, AddressMaxVel, 1, &data[0]);

tmp_errno = modbus_strerror(errno);

Relevant address':

AddressMaxVel (0xA107)

18.2.14 ReadVelHome

*int16_t ReadVelHome(modbus_t *ctx, int* rc_arg, string header)*

Meaning:

Function used to read the vel_home of the driver.

rc_arg will contain the status of the operation.

Driver interaction:

rc = modbus_read_registers(ctx, AddressVelHome, 1, &data[0]);

tmp_errno = modbus_strerror(errno);

Relevant address':

AddressVelHome (0xA00A)

18.2.15 ReadAcceleration

*uint16_t ReadAcceleration(modbus_t *ctx, int* rc_arg, string header)*

Meaning:

Function used to read the acceleration of the driver.

rc_arg will contain the status of the operation.

Driver interaction:

rc = modbus_read_registers(ctx, AddressAcceleration, 1, &data[0]);

tmp_errno = modbus_strerror(errno);

Relevant address':

AddressAcceleration (0xA109)

18.2.16 ReadDeceleration

*uint16_t ReadDeceleration(modbus_t *ctx, int* rc_arg, string header)*

Meaning:

Function used to read the deceleration of the driver.

rc_arg will contain the status of the operation.

Driver interaction:

rc = modbus_read_registers(ctx, AddressDeceleration, 1, &data[0]);

tmp_errno = modbus_strerror(errno);

Relevant address':

AddressDeceleration (0xA10A)

18.2.17 ReadPhaseCurrent

*uint16_t ReadPhaseCurrent(modbus_t *ctx, int* rc_arg, string header)*

Meaning:

Function used to read the phase_current of the driver.

rc_arg will contain the status of the operation.

Driver interaction:

rc = modbus_read_registers(ctx, PhaseCurrentAddress, 1, &data[0]);

tmp_errno = modbus_strerror(errno);

Relevant address':

PhaseCurrentAddress (0xA103)

18.2.18 ReadAnalogOutput0

*uint16_t ReadAnalogOutput0(modbus_t *ctx, int* rc_arg, string header)*

Meaning:

Function used to read the analog_output0 of the driver.

rc_arg will contain the status of the operation.

Driver interaction:

rc = modbus_read_registers(ctx, AddressAnalogOutput0, 1, &data[0]);

tmp_errno = modbus_strerror(errno);

Relevant address':

AddressAnalogOutput0 (0xA204)

18.2.19 ReadStatusState

*uint16_t ReadStatusState(modbus_t *ctx, int* rc_arg, string header)*

Meaning:

Function used to read the status_state register of the driver.

rc_arg will contain the status of the operation.

Driver interaction:

rc = modbus_read_registers(ctx, StatusStateAddress, 1, &data[0]);

tmp_errno = modbus_strerror(errno);

Relevant address':

StatusStateAddress (0xA005)

18.2.20 ReadAnalogInput0

*uint16_t ReadAnalogInput0(modbus_t *ctx, int* rc_arg, string header)*

Meaning:

Function used to read the analog_input0 of the driver.

rc_arg will contain the status of the operation.

Driver interaction:

rc = modbus_read_registers(ctx, AddressAnalogInput0, 1, &data[0]);

Relevant address':

AddressAnalogInput0 (0xA202)

18.2.21 ReadCurrentPosition

*int ReadCurrentPosition(modbus_t *ctx, int* rc_arg, string header)*

Meaning:

Function used to read the current_position of the driver.

rc_arg will contain the status of the operation.

Driver interaction:

rc = modbus_read_registers(ctx, CurrentPositionAddress, 2, &data[0]);

Relevant address':

CurrentPositionAddress (0xA10B)

18.3 Mid level function.

General information:

These functions generally call the low level ones. They are called by Main.c when the server want execute a command.

N.B. Some of these functions are blocking.

N.B. SendFailedGetPar, SendFailedGetMovPar, SendFailedGetStatusPos, LoadEncoderFromFile, ReadActualEncoderValue and HelpCommand don't interact with the drivers so they are not reported in the description
below.

Precondition:
The same of low level functions. These function are designed assuming that ctx is a valid modbus_t resource. If this condition is not respected the result could be a segmentation fault.
The program calls these functions only if *STATE_CONNECT == 1*.
It is relevant to point out that the segmentation fault does not happened if the modbus_new_rtu function is correctly executed. So, if you connected the PC to the driver but at a later time you switch
off the system, the server does not crash because ctx is a valid resource yet. The communication with the programmer will be impossible but the resource is still valid.

18.3.1 CheckDrvAssoc

void CheckDrvAssoc (CommunicationObject& mioTCP, Input mioinput, modbus_t* ctx)*
Meaning:
This function tries to read **SerialDrvLog.txt** file, then compares the serial numbers
read in that file with the serial numbers obtained querying the drivers.

Low level interaction:

modbus_flush(ctx);

modbus_set_slave(ctx, j+1);

SerialNumber = ReadSerialNumber(ctx, &function_status);

Special attention: this function is a blocking one. When it reads the serial numbers from the drivers and from the **SerialDrvLog.txt**, it asks the user to accept the situation. The timeout is fixed to *LIMITCHECKLOG*SLEEPCHECK*seconds* (see **DefineGeneral.h** file for *LIMITCHECKLOG* and *SLEEPCHECK* values).

It is called Analizza1 procedure that is created by flex (see Makefile in ServerProgram directory and **SerialDrvLog.flex** file).

18.3.2 CheckParAssoc

void CheckParAssoc (CommunicationObject& mioTCP, Input mioinput, modbus_t* ctx)*

Meaning:

This function tries to read **FileParLog.txt** file, then compares the parameters read in that file with the parameters obtained querying the drivers.

Low level interaction:

modbus_flush(ctx);

modbus_set_slave(ctx, j+1);

tmp_parameter_struct.max_vel = ReadMaxVel(ctx, &function_status, "check_drv_assoc_exp: ");

tmp_parameter_struct.vel_home = ReadVelHome(ctx, &function_status, "check_drv_assoc_exp: ");

tmp_parameter_struct.acceleration = ReadAcceleration(ctx, &function_status, "check_drv_assoc_exp: ");

tmp_parameter_struct.deceleration = ReadDeceleration(ctx, &function_status, "check_drv_assoc_exp: ");
tmp_parameter_struct.phase_current = ReadPhaseCurrent(ctx, &function_status, "check_drv_assoc_exp: ");
tmp_parameter_struct.analog_output0 = ReadAnalogOutput0(ctx, &function_status, "check_drv_assoc_exp: ");
Special attention: this function is a blocking one. When it reads the parameters from the drivers and from the **FileParLog.txt**, it asks the user to accept the situation. The timeout is fixed to
*LIMITCHECKLOG*SLEEPCHECK*seconds* (see **DefineGeneral.h** file for *LIMITCHECKLOG* and *SLEEPCHECK* values).
It is called AnalizzaFilePar procedure that is created by flex (see Makefile in ServerProgram directory and **FileParLog.flex** file).

18.3.3 CheckEncodeAssoc

void CheckEncodeAssoc (CommunicationObject& mioTCP, Input mioinput, modbus_t* ctx)*

Meaning:
This function tries to read EncoderLog.txt file, then compares the parameters
read in that file with the parameters obtained querying the drivers.

Low level interaction:

```
modbus_flush(ctx);

Encode(ctx, j+1, tmp_encoder_struct);

if (ENCODINGHOME == 1)
{
```

```
        Homing(ctx, j+1);
}
```

Special attention: this function is a blocking one. When it reads the parameters from the drivers and from the **FileParLog.txt**, it asks the user to accept the situation. The timeout is fixed to *LIMITCHECKLOG*SLEEPCHECK*seconds* (see **DefineGeneral.h** file for *LIMITCHECKLOG* and *SLEEPCHECK* values).
It is called AnalizzaFileEncoder procedure that is created by flex (see Makefile in ServerProgram directory and **FileParLog.flex** file).
Encode and Homing functions are defined in **CommandExecutor.c** file too.

18.3.4 GetPar

void GetPar (modbus_t ctx, int get_par_value)*

Meaning:
This function collects the parameters from the driver indicated by get_par_value.
The parameters collected are: max_vel, vel_home, acceleration, deceleration, phase_current,
AnalogOutput0.

Low level interaction:
modbus_flush(ctx);
function_status = modbus_set_slave(ctx, get_par_value);
tmp_parameter_struct.max_vel = ReadMaxVel(ctx, &function_status, "Exp: ");
tmp_parameter_struct.vel_home = ReadVelHome(ctx, &function_status, "Exp: ");
tmp_parameter_struct.acceleration = ReadAcceleration(ctx, &function_status, "Exp: ");
tmp_parameter_struct.deceleration = ReadDeceleration(ctx, &function_status, "Exp: ");

tmp_parameter_struct.phase_current = ReadPhaseCurrent(ctx, &function_status, "Exp: ");
tmp_parameter_struct.analog_output0 = ReadAnalogOutput0(ctx, &function_status, "Exp: ");

Special attention: this function is not blocking. It tries to obtained some information from the driver and if the attempt fails it sends "-1" instead of the parameters.

18.3.5 SetPar

void SetPar (modbus_t ctx, int set_par_value, char* buffer)*

Meaning:
This function sets the parameters of driver indicated by set_par_value to the values contained in buffer. The values are: max_vel, vel_home, acceleration, deceleration, phase_current, AnalogOutput0.

Low level interaction:
modbus_flush(ctx);
function_status = SetMaxVel(ctx, max_vel, "Exp: ");
function_status = SetVelHome(ctx, vel_home, "Exp: ");
function_status = SetAcceleration(ctx, acceleration, "Exp: ");
function_status = SetDeceleration(ctx, deceleration, "Exp: ");
function_status = SetPhaseCurrent(ctx, phase_current, "Exp: ");
function_status = SetAnalogOutput0(ctx, analog_output0, "Exp: ");
GetPar(ctx, set_par_value);
Special attention: The precondition to use this function is that in buffer is stored a consistent set_par command. This is guaranteed by the check of the command in Main.c . The correct syntax of the command is: set_par drvnum max_vel acceleration deceleration PhaseCurrent AnalogOutput0.
This function calls GetPar that is defined in CommandExecutor.c .

18.3.6 SetParMult

void SetParMult (modbus_t ctx, int set_par_value, char* buffer)*

Meaning:

This function sets the parameters of driver indicated by set_par_value to the values contained in buffer. The values are: max_vel, vel_home, acceleration, deceleration, phase_current, AnalogOutput0.

Low level interaction:

modbus_flush(ctx);
function_status = SetMaxVel(ctx, max_vel, "Exp: ");
function_status = SetVelHome(ctx, vel_home, "Exp: ");
function_status = SetAcceleration(ctx, acceleration, "Exp: ");
function_status = SetDeceleration(ctx, deceleration, "Exp: ");
function_status = SetPhaseCurrent(ctx, phase_current, "Exp: ");
function_status = SetAnalogOutput0(ctx, analog_output0, "Exp: ");
GetPar(ctx, set_par_value);

Special attention:

This function is the same of the SetPar one in except of the syntax of the command stored in buffer.

The precondition to use this function is that in buffer is stored a consistent set_par command. This is guaranteed by the check of the command in Main.c . The correct syntax of the command is:

set_par drvnum max_vel acceleration deceleration PhaseCurrent AnalogOutput0.

This function calls GetPar that is defined in CommandExecutor.c .

18.3.7 Homing

void Homing(modbus_t ctx, int homing_value)*

Meaning: This function orders the driver indicated by homing_value to execute the homing procedure.
See firmware documentation for more information about the procedure.

Low level interaction:
modbus_flush(ctx);
function_status = modbus_set_slave(ctx, homing_value);
status_state = ReadStatusState(ctx, &rc, "Exp: ");
function_status = SetStatusState(ctx, 0, "Exp: ");
function_status = SetRequestState(ctx, STATEHOMING, "Exp :");

Special attention: this function is blocking. When the function begins, it is checked that the drivers has terminated the previous operation checking the StatusState register. The timeout is equal to
*LIMITSTATUS_STATE*SLEEPSTATUS_STATE*microseconds* (see **DefineGeneral.h** file for *LIMITSTATUS_STATE* and *SLEEPSTATUS_STATE* values).

18.3.8 GetMovePar

void GetMovePar(modbus_t ctx, int mov_par_value)*

Meaning: this function collects the movimentation parameters from the driver indicated by mov_par_value. The movimentation parameters are CurrentPosition and AnalogInput0. CorrentPosition is the actual
position of the driver, AnalogInput0 is the values retrieved by encoder to the driver and it can be used for checking the position of the engine mastered by the driver. Since the operation requires the
driver to have already accomplished the previous operation, a check to the status of the driver is performed.

Low level interaction:

modbus_flush(ctx);

function_status = modbus_set_slave(ctx, mov_par_value);

status_state = ReadStatusState(ctx, &rc, "Exp: ");

current_position = ReadCurrentPosition(ctx, &function_status, "Exp: ");

analog_input0 = ReadAnalogInput0(ctx, &function_status, "Exp: ");

Special attention: this function is blocking. When the function begins, it is checked that the drivers has terminated the previous operation checking the StatusState register. The timeout is equal to *LIMITSTATUS_STATE*SLEEPSTATUS_STATE*microseconds* (see **DefineGeneral.h** file for *LIMITSTATUS_STATE* and *SLEEPSTATUS_STATE* values).

18.3.9 MoveTo

void MoveTo(modbus_t ctx, int moveto_drv_num, char* buffer)*

Meaning: this function set the CountTargetPosition of the driver indicated by moveto_drv_num to the values found in buffer.

Low level interaction:

modbus_flush(ctx);

function_status = modbus_set_slave(ctx, moveto_drv_num);

status_state = ReadStatusState(ctx, &rc, "Exp: ");

function_status = SetStatusState(ctx, 0, "Exp: ");

function_status = SetCountTargetPosition(ctx, moveto_value, "Exp :");

function_status = SetRequestState(ctx, STATEMOVEREL, "Exp :");

Special attention: The precondition to execute the function is that in buffer is stored a valid move_to command. This is guaranteed by the check performed in **Main.c**.
The correct syntax of the command is: move_to drvnum val . In order to accomplished

the movimentation is performed a check to the status of the driver: it has to have already terminated the previous operation.
This function is blocking. When the function begins, it is checked that the drivers has terminated the previous operation checking the StatusState register. The timeout is equal to
*LIMITSTATUS_STATE*SLEEPSTATUS_STATE*microseconds* (see **DefineGeneral.h** file for *LIMITSTATUS_STATE* and *SLEEPSTATUS_STATE* values).

18.3.10 MoveToMult

void MoveToMult(modbus_t ctx, int moveto_drv_num, char* buffer)*
Meaning: This function set the CountTargetPosition of the driver indicated by moveto_drv_num to the values found in buffer.

Low level interaction:
modbus_flush(ctx);
function_status = modbus_set_slave(ctx, moveto_drv_num);
status_state = ReadStatusState(ctx, &rc, "Exp: ");
function_status = SetStatusState(ctx, 0, "Exp: ");
function_status = SetCountTargetPosition(ctx, moveto_value, "Exp :");
function_status = SetRequestState(ctx, STATEMOVEREL, "Exp :");

Special attention: this function is the of the MoveTo one in except of the syntax of the command stored in buffer. The precondition to execute the function is that in buffer is stored a valid move_to command. This is guaranteed by the check performed in Main.c. The correct syntax of the command is: move_to drvnum val . In order to accomplished the movimentation is performed a check to the status of the
driver: it has to have already terminated the previous operation.
This function is blocking. When the function begins, it is checked that the drivers has terminated the previous operation checking the StatusState register. The timeout is equal to

*LIMITSTATUS_STATE*SLEEPSTATUS_STATE*microseconds* (see **DefineGeneral.h** file for *LIMITSTATUS_STATE* and *SLEEPSTATUS_STATE* values).

18.3.11 Encode

void Encode(modbus_t ctx, int encode_drv_num, EncoderStruct& drv_parameters)*

Meaning: This function executes an encoding procedure for the driver indicated in encode_drv_num and save in drv_parameters struct the values obtained from the linear regression.

Low level interaction:

modbus_flush(ctx);

function_status = modbus_set_slave(ctx, encode_drv_num);

status_state = ReadStatusState(ctx, &rc, "Exp: ");

MoveTo(ctx, encode_drv_num, (char) tmp_buffer.c_str());*

Special attention: Warning: in order to execute this function the user must have already done the homing procedure. If this has not happened, this function returns inconsistent values. In **DefineGeneral.h** there is a definition called *ENCODINGHOME*: if *ENCODINGHOME == 1*, **Main.c** execute an homing procedure before calling the encoding procedure.

The encoding procedure consists to reach the final position indicated by *MAXEXTENSION* by steps indicated by *ENCODINGSTEP*.

Be careful: it is assumed that the path begins from 0 and ends to a negative value. You have to check the driver polarity before calling this procedure.

18.3.12 CheckPositionEncoderSingle

int CheckPositionEncoderSingle (modbus_t ctx, int position_encoder_drv_num)*

Meaning: this function performs the comparison between the position declared by the

driver (the value of the register Position) and the position retrieved by the encoder (the value of the register
AnalogInput0).
The driver is indicated by position_encoder_drv_num.
Return values:
0 all okay
-1 real position mismatch with estimated position
-2 problem communicating with the driver
everything > 0 the driver is blocked in an invalid state

Low level interaction:
modbus_flush(ctx);
status_state = ReadStatusState(ctx, &rc, "Exp: ");
current_position = ReadCurrentPosition(ctx, &function_status, "Exp: ");
analog_input0 = ReadAnalogInput0(ctx, &function_status, "Exp: ");

Special attention:
This function is blocking. When the function begins, it is checked that the drivers has terminated the previous operation checking the StatusState register. The timeout is equal to
*LIMITSTATUS_STATE*SLEEPSTATUS_STATE*microseconds* (see **DefineGeneral.h** file for *LIMITSTATUS_STATE* and *SLEEPSTATUS_STATE* values).

18.3.13 CheckPositionEncoderSingleWarning

int CheckPositionEncoderSingleWarning (modbus_t ctx, int position_encoder_drv_num)*

Meaning: this function performs the comparison between the position declared by the driver (the value of the register Position) and the position retrieved by the encoder (the value of the register

AnalogInput0).

The driver is indicated by position_encoder_drv_num.

Return values:

0 all okay

-1 real position mismatch with estimated position

-2 problem communicating with the driver

everything > 0 the driver is blocked in an invalid state

Low level interaction:

modbus_flush(ctx);

status_state = ReadStatusState(ctx, &rc, "Exp: ");

current_position = ReadCurrentPosition(ctx, &function_status, "Exp: ");

analog_input0 = ReadAnalogInput0(ctx, &function_status, "Exp: ");

Special attention:

This function is the same of CheckPositionEncoderSingle but CheckPositionEncoderSingleWarning sends a warning message to the client when loading_encoder_from_file_okay is equal to 0 (it means that the user has not already loaded the encoding values in EncoderArrayValue using the command load_encoder_from_file).

This function is blocking. When the function begins, it is checked that the drivers has terminated the previous operation checking the StatusState register. The timeout is equal to

*LIMITSTATUS_STATE*SLEEPSTATUS_STATE*microseconds* (see **DefineGeneral.h** file for *LIMITSTATUS_STATE* and *SLEEPSTATUS_STATE* values).

18.3.14 CheckPositionEncoderToAll

int CheckPositionEncoderToAll (modbus_t ctx, int position_encoder_drv_num)*

Meaning: this function is the same of CheckPositionEncoderSingle but

CheckPositionEncoderToAll sends the response to all the clients connected to the server. It is used by Main.c in order to perform a
periodical check and send the response to the clients.
This function performs the comparison between the position declared by the driver (the value of the register Position) and the position retrieved by the encoder (the value of the register AnalogInput0).
The driver is indicated by position_encoder_drv_num.

Return values:
0 all okay
-1 real position mismatch with estimated position
-2 problem communicating with the driver
everything > 0 the driver is blocked in an invalid state

Low level interaction:
modbus_flush(ctx);
status_state = ReadStatusState(ctx, &rc, "Exp: ");
current_position = ReadCurrentPosition(ctx, &function_status, "Exp: ");
analog_input0 = ReadAnalogInput0(ctx, &function_status, "Exp: ");

Special attention: This function is blocking. When the function begins, it is checked that the drivers has terminated the previous operation checking the StatusState register. The timeout is equal to
*LIMITSTATUS_STATE*SLEEPSTATUS_STATE*microseconds* (see **DefineGeneral.h** file for *LIMITSTATUS_STATE* and *SLEEPSTATUS_STATE* values).

18.4 High level functions

In this section it will reported the command syntax and the mid/low level functions called for each command (in other words, the functions called by Main.c) .

N.B. The list below reports only the commands that call at least one mid/low level function!

1.

Command: check_drv_assoc

Function called: CheckDrvAssoc

2.

Command: check_par_assoc

Function called: CheckParAssoc

3.

Command: check_encode_assoc

Function called: CheckEncodeAssoc

4.

Command: connect absoluteprogrammerpath

Function called: Connect

5.

Command: get_par drvnum

Function called: GetPar

6.

Command: check_position drvnum

Function called: CheckPositionEncoderSingleWarning

7.

Command: set_par drvnum max_vel acceleration deceleration PhaseCurrent AnalogOutput0

Function called: SetPar

8.

Command: homing drvnum

Functions called: Homing, GetMovePar

9.

Command: get_mov_par drvnum

Function called: GetMovePar

10.

Command: encode drvnum

Functions called: Homing (only if ENCODINGHOME == 1. See DefineGeneral.h file.), Encode

11.

Command: move_to drvnum targetposition

Functions called: MoveTo, GetMovePar

12.

Command: get_all_parameter

Functions called: GetMovePar, GetPar, CheckPositionEncoderSingle

13.

Command: homing_mult drvnum1 drvnum2 drvnum3 drvnum....

Functions called: Homing, GetMovePar

14.

Command: moveto_mult targetposition drvnum1 drvnum2 drvnum3 drvnum....

Functions called: MoveToMult, GetMovePar

15.
Command: setmult_par max_vel acceleration deceleration PhaseCurrent AnalogOutput0 drvnum1 drvnum2 drvnum3 drvnum....

Function called: SetParMult

Name	Called Functions	Definition file	Blocking	References
Connect	modbus_new_rtu modbus_strerror modbus_free modbus_set_slave	DriverFunction.c	NO	18.2.1
ReadSerialNumber	modbus_read_registers modbus_strerror	DriverFunction.c	NO	18.2.2
SetMaxVel	modbus_write_registers modbus_strerror	DriverFunction.c	NO	18.2.3
SetVelHome	modbus_write_registers modbus_strerror	DriverFunction.c	NO	18.2.4
SetAcceleration	modbus_write_registers modbus_strerror	DriverFunction.c	NO	18.2.5
SetDeceleration	modbus_write_registers modbus_strerror	DriverFunction.c	NO	18.2.6
SetPhaseCurrent	modbus_write_registers modbus_strerror	DriverFunction.c	NO	18.2.7
SetAnalogOutput0	modbus_write_registers modbus_strerror	DriverFunction.c	NO	18.2.8
SetStatusState	modbus_write_registers modbus_strerror	DriverFunction.c	NO	18.2.9
SetRequestState	modbus_write_registers modbus_strerror	DriverFunction.c	NO	18.2.10
SetTargetPosition	modbus_write_registers modbus_strerror	DriverFunction.c	NO	18.2.11
SetCountTargetPosition	modbus_write_registers modbus_strerror	DriverFunction.c	NO	18.2.12
ReadMaxVel	modbus_read_registers modbus_strerror	DriverFunction.c	NO	18.2.13
ReadVelHome	modbus_read_registers modbus_strerror	DriverFunction.c	NO	18.2.14
ReadAcceleration	modbus_read_registers	DriverFunction.c	NO	18.2.15

	modbus_strerror			
ReadDeceleration	modbus_read_registers modbus_strerror	DriverFunction.c	NO	18.2.16
ReadPhaseCurrent	modbus_read_registers modbus_strerror	DriverFunction.c	NO	18.2.17
ReadAnalogOutput0	modbus_read_registers modbus_strerror	DriverFunction.c	NO	18.2.18
ReadStatusState	modbus_read_registers modbus_strerror	DriverFunction.c	NO	18.2.19
ReadAnalogInput0	modbus_read_registers	DriverFunction.c	NO	18.2.20
ReadCurrentPositio n	modbus_read_registers	DriverFunction.c	NO	18.2.21
CheckDrvAssoc	modbus_flush modbus_set_slave ReadSerialNumber	CommandExecutor.c	YES	18.3.1
CheckParAssoc	modbus_flush modbus_set_slave ReadMaxVel ReadVelHome ReadAcceleration ReadDeceleration ReadPhaseCurrent ReadAnalogOutput0	CommandExecutor.c	YES	18.3.2
CheckEncodeAssoc	modbus_flush Encode Homing	CommandExecutor.c	YES	18.3.3
GetPar	modbus_flush modbus_set_slave ReadMaxVel ReadVelHome ReadAcceleration	CommandExecutor.c	NO	18.3.4

	ReadDeceleration ReadPhaseCurrent ReadAnalogOutput0			
SetPar	modbus_flush SetMaxVel SetVelHome SetAcceleration SetDeceleration SetPhaseCurrent SetAnalogOutput0 GetPar	CommandExecutor.c	YES	18.3.5
SetParMult	modbus_flush SetMaxVel SetVelHome SetAcceleration SetDeceleration SetPhaseCurrent SetAnalogOutput0 GetPar	CommandExecutor.c	YES	18.3.6
Homing	modbus_flush modbus_set_slave ReadStatusState SetStatusState SetRequestState	CommandExecutor.c	YES	18.3.7
GetMovePar	modbus_flush modbus_set_slave ReadStatusState ReadCurrentPosition ReadAnalogInput0	CommandExecutor.c	YES	18.3.8
MoveTo	modbus_flush modbus_set_slave	CommandExecutor.c	YES	18.3.9

	ReadStatusState SetStatusState SetCountTargetPosition SetRequestState			
MoveToMult	modbus_flush modbus_set_slave ReadStatusState SetStatusState SetCountTargetPosition SetRequestState	CommandExecutor.c	YES	18.3.10
Encode	modbus_flush modbus_set_slave ReadStatusState MoveTo	CommandExecutor.c	YES	18.3.11
CheckPositionEncoderSingle	modbus_flush ReadStatusState ReadCurrentPosition ReadAnalogInput0	CommandExecutor.c	YES	18.3.12
CheckPositionEncoderSingleWarning	modbus_flush ReadStatusState ReadCurrentPosition ReadAnalogInput0	CommandExecutor.c	YES	18.3.13
CheckPositionEncoderToAll	modbus_flush ReadStatusState ReadCurrentPosition ReadAnalogInput0	CommandExecutor.c	YES	18.3.14

Table 1: Summary of the system functions. In the field called "blocking" is indicated if the function launches an opertion that makes the server not available until the end of the operation.

Command name	Functions called	Blocking	References
check_drv_assoc	CheckDrvAssoc	YES	15.1.6
check_par_assoc	CheckParAssoc	YES	15.1.7
check_encode_assoc	CheckEncodeAssoc	YES	15.1.8
connect	Connect	NO	15.1.9
get_par	GetPar	NO	15.1.11
check_position	CheckPositionEncoderSingle Warning	YES	15.1.12
set_par	SetPar	YES	15.1.13
homing	Homing GetMovePar	YES	15.1.15
get_mov_par	GetMovePar	YES	15.1.14
encode	Homing Encode	YES	15.1.16
move_to	MoveTo GetMovePar	YES	15.1.17
get_all_parameter	GetMovePar GetPar CheckPositionEncoderSingle	YES	15.1.18
homing_mult	Homing GetMovePar	YES	15.1.19
moveto_mult	MoveToMult GetMovePar	YES	15.1.20
setmult_par	SetParMult	YES	15.1.21

Table 2: List of the relevant functions called by the CollSoft server. Using the adjective relevant, I mean every function that interact with the drivers.