Driver explanation

Thibault Barbié

April 27, 2016

1 Introduction

The servomotor of D-Hand was made to be first used on Windows, the company provided a software to control the hand but it does not work on Linux. Hence we decided to write our own driver to control the servomotor through Linux. The protocol used to communicate between the hand and the computer is **Modbus**.

Modbus is a serial communication protocol heavily used around the world. It is the standard protocol to communicate with electrical devices. Modbus has different protocol versions but in our case we used **Modbus RTU** and the servomotor was connected to the computer with the USB port. Modbus works by sending information (using bits) to some specific registers (defined differently in each device). To know which register we should communicate with we used the servomotor company manual **ACON C/CG operation manual** and the **ACON serial communication operation manual**.

Through this document we will use two ways to define numbers, the decimal base and the hexadecimal base. When a number is in the decimal base it will appear normaly (as in 9900). When a number is in hexadecimal base it will appear with a H under the number (as in 9900_H).

2 Servomotor path initialisation

The first thing to do is to verify that the computer detects the servomotor. To do this we could simply check the file **/dev** with the following command line.

```
ls /dev
```

The servomotor should normaly appear in /dev/ttyUSB0. To allow communication between the servomotor and the computer we then give rights to /dev/ttyUSB0 by writing the following command line.

```
sudo chmod 777 /dev/ttyUSB0
```

Now we should be able to send and receive information from the computer to the servomotor.

3 Modbus protocol explanation

With the Modbus protocol, the computer sends a request to the device and the device answers to the computer. There are multiple ways to communicate with a device but we will just speak about the RTU mode. With RTU mode we send a chain of information. It begins with the address of the device, in our case the servomotor is put at address 1 (it can be changed in the IAI physical controller). Then it continues with the function code which indicate if the computer wants to write information in the device or read information from the device. Then there is the data. And finally there is the CRC check which is a safety against transmission errors.

A typical chain send by the computer is then : 010390000002E90B. We should note that this chain is in hexadecimal. Hence : 01_H is the device address, 03_H is the function code (indicating that the computer wants to read information from the device), 9000_H is the register address, 0002_H is the data sent (in this case indicating the number of register to be read) and $E90B_H$ is the CRC check.

To not worry with the modbus protocol we used the python module **minimalmodbus**. Minimalmodbus is a simple module made to be easy to use. We used mostly four functions : *write_bit*, *write_register*, *write_registers* and *read_register*. With these functions we just have to put the register and the data we want to send the rest is automaticaly handled by the minimalmodbus.

4 Code explanation

In the driver python file the first thing to do was to set the baud rate to 38400. If the baud rate was different then the communication between the computer and the servomotor would be impossible. Then we created a minimalmodbus *Instrument* by putting the device path and the address of the device (in our case "ttyUSB0" and 1). We also initialised the debug mode. In debug mode the user will be able to see the answer of the servomotor when sending requests.

```
def __init__(self,name,debug):
        minimalmodbus.BAUDRATE=38400
        self.s=minimalmodbus.Instrument("/dev/"+name,1)
        self.s.debug=debug
```

Listing 1: driver.py constructor

To initialise the servomotor we created an initialisation python file. In the initialisation file we proceeded in multiple steps.

```
# Author : Thibault Barbie

# This file initialise the servomotor. It removes the alarm and do a home return.

from driver import *
import time

b=Servo("ttyUSB0",False)

b.alarm_reset()
print "Alarm"
time.sleep(1.5)

b.modbus_on()
print "Modbus on"
time.sleep(1.5)

b.servo_on()
print "Servo On"
time.sleep(1.5)

b.home_return()
print "Home"
time.sleep(1.5)

b.servo_off()
print "Servo Off"
```

Listing 2: initialisation.py

The first one was to reset any alarm that could have occured. To do this we used the method *alarm_reset* of the driver.

```
def alarm_reset(self):
        # Alarm reset
        register=int('0407',16)
        self.s.write_bit(register,1,functioncode=5)

        # Restore normal status
        register=int('0407',16)
        self.s.write_bit(register,0,functioncode=5)
```

Listing 3: driver.py alarm_reset method

To reset the alarm we just put **ALRS** to 1 in the register 0407_H (ACON-C Modbus p129) and then put it again to 0 (ACON-C Modbus p36). We then activated the modbus mode which allow us to change everything with the modbus protocol. To do this we put **PMSL** to 1 in the register 0427_H(ACON-C Modbus p149).

```
def modbus_on(self):
        # Modbus ON
        register=int('0427',16)
        self.s.write_bit(register,1,functioncode=5)
```

Listing 4: driver.py modbus_on method

We then activated the servomotor by putting **SON** at 1 in the register 0403_H (ACON-C Modbus p123).

```
def servo_on(self):
        # Servo ON
        register=int('0403',16)
        self.s.write_bit(register,1,functioncode=5)
```

Listing 5: driver.py servo_on method

Before beginning to move the servomotor a home return must be done. To do it we need to create a rising edge in **HOME** by writing bits in register $040B_H$ (ACON-C Modbus p131).

```
def home_return(self):
        # Home return
        register=int('040B',16)
        self.s.write_bit(register,0,functioncode=5)
        self.s.write_bit(register,1,functioncode=5)
```

Listing 6: driver.py home_return method

We finally ended the initialisation by putting **SON** to 0. If we did not do this the servomotor would still work against any external forces (it would be impossible to manually move the position).

```
def servo_on(self):
        # Servo OFF
        register=int('0403',16)
        self.s.write_bit(register,0,functioncode=5)
```

Listing 7: driver.py servo_off method

To move the servomotor to a specific position we used 9 registers, beginning with the register 9900_H (ACON-C Modbus p158, p159 and p171). We first selected the position we wanted to attain (in mm). Then the positioning band was set with a small value ($0.1mm$). Then the speed (in $mm.s^{-1}$). Then the acceleration (in G). The push mode was used to hold object (ACON-C manual p70). It was set by putting a percentage of the current (ACON-C Modbus p159). The method set in the driver allows the user to change the position, the speed, the acceleration and the push. It contains protection to not break the servomotor.

```
def move_absolute_position(self,position,speed,acceleration,push):
        # Move the servomotor to the position except if an obstacle is detected
        # position in mm, speed in mm/s, acceleration in G, push in percentage (0.2-0.7)
        register=int('9900',16)

        if (position<0):
            position=0
        elif (position>13):
            position=13 # Maximum value before touching the palm
        if (speed<0):
            speed=0
        elif (speed>100):
            speed=100 # Maximum value (empirical)
        if (acceleration<0):
            acceleration=0
        elif (acceleration>0.209):
            acceleration=0.209 # Maximum value (empirical)
        if (push<0.2):
            push=0.2
        elif (push>0.7):
            push=0.7 # Maximum value said in manual

        l=[]
        l.append(0000)
        l.append(int(position*100))

        l.append(0000)
        l.append(10)

        l.append(0000)
        l.append(int(speed*100))

        l.append(int(acceleration*100))

        l.append(int(255*push))

        self.s.write_registers(register,l)
```