Interface Control Document for Q/W Calibration Load Motor Control Software

Gabriel Gómez-Molina, Fco. Javier Beltrán, Félix Tercero, Oscar García

IT-CDT 2021-9
Observatorio de Yebes
Apdo. 148, E-19080
Guadalajara, SPAIN

Contents

List of Figures

1 Introduction

This document describes the interface between the motor software and the radio-telescope control software (ACS). A general description of the software and the main functions to operate the motor are given in this document.

The software is developed in C++ language and a Makefile is given to compile it and to generate a test program (only for testing purposes for the developed classes). The user of this software is another software (ACS), therefore the interface between the two must be compatible and clear. Since C++ is one of the languages supported by CORBA, the interface is transparent.

The structure of the software is explained in section 2, the configuration file and the dictionaries with the motor commands and motor axis parameters are described in section 3, the internal operation is commented in section 4, and finally, the user interface methods are described in section 5.

2 Class diagram

A simplified class diagram with the main methods is shown in Figure 1.

The software is divided in three main classes plus a common functions file (commonFunctions.cc).

- **Comm:** This is the communication class. This class contains the method to initialize a TCP socket to establish a connection with the motor, and implements the methods to send commands to the motor and receive replies from it. The communication is not handled directly with the stepper motor, but with a Raspberry Pi 3B+ (RS232-Ethernet interface).
- **StepperCommand:** This class implements the data structures for the commands and replies in the format that the motor specifies. It also contains the methods to encode the commands to send to the motor and decode the replies received from the motor, as well as checking if the received status code is correct.
- **MotorFunctions:** Here is the most important class, that implements the functionality to operate the motor and uses the other two classes to support that operation. The methods to configure, move the load and get the position of the load, among others, are implemented in this class.

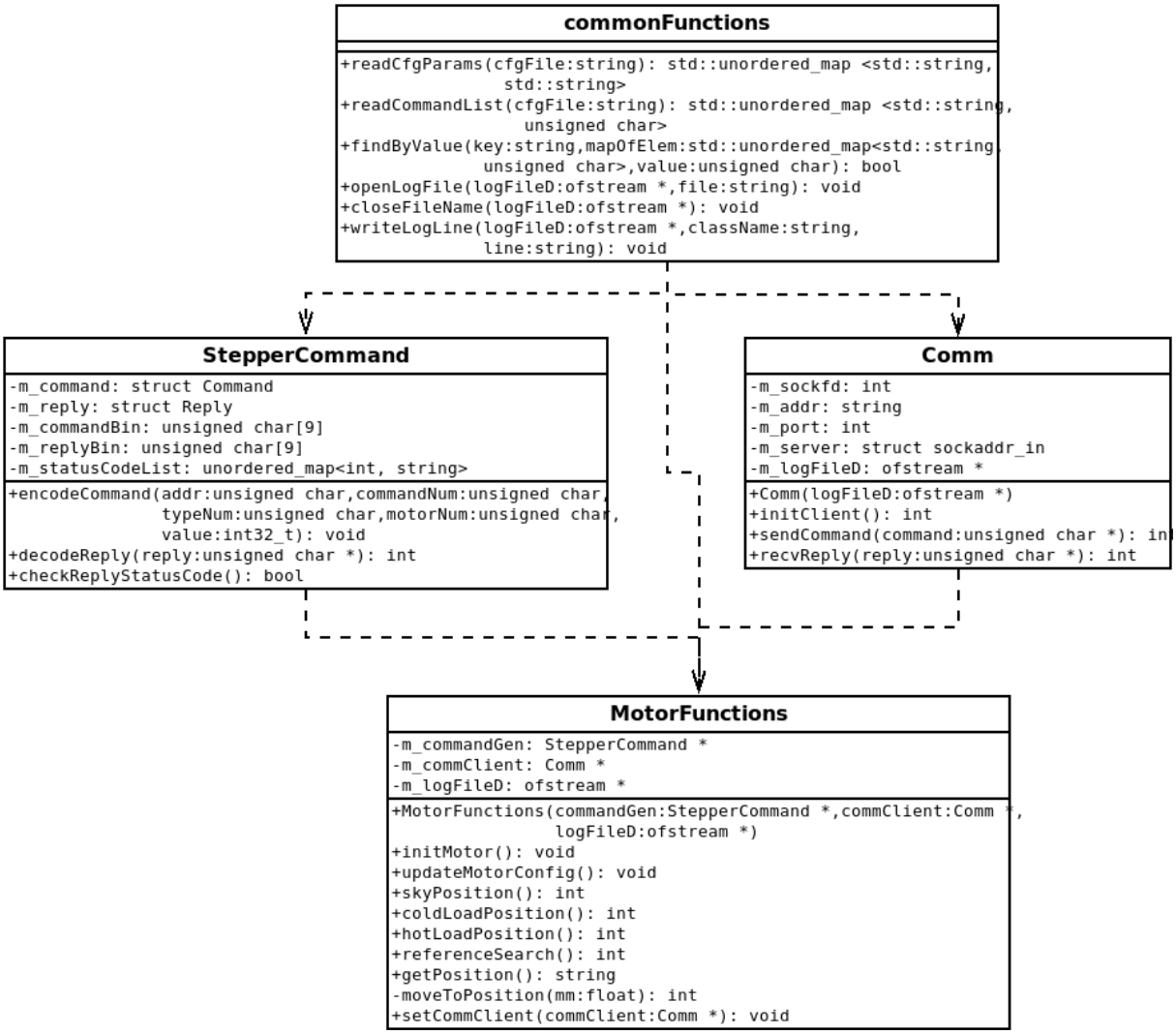

Figure 1: Class diagram

- **commonFunctions:** This file contains the functions to read the parameters.cfg file and the dictionaries of commands and motor axis parameters. It also implements the logger functions.

These classes will be described in depth in section 4.

3 Configuration files and Dictionaries

The stepper motor firmware (PD-1180 TMCL) has a list of commands to operate it, and a list of motor axis parameters that can be set or read. The commands have a mnemonic and a number associated with it. The same thing happens to the axis parameters. In order to work with parameter and command names instead of numbers, two dictionaries have

been created. The **commandList.txt** and **motorAxisParameters.txt** each one with a list of **command=number** and **axisParameter=number** respectively. These files are read by the program and the dictionaries are loaded in the program memory so the mnemonics and parameter names can be used to make the software more readable.

Regarding the configurable parameters of the system, there is another file, the **parameters.cfg** configuration file with the main parameters to set up to be able to connect to the Raspberry Pi 3B+ (RS232-Ethernet interface) and operate the motor. The file structure is the same as the other two described above: parameterName=value.

In this file, the Raspberry Pi 3B+ IP address, the motor module address, the microstep resolution of the motor, the ball screw lead, the speed and acceleration, the load positions, etc are configured.

4 Internal operation

The internal operation of the classes **Comm** and **StepperCommand** is transparent to the user, so, for more information besides the description given in section 2, the C++ code is provided.

Regarding the **MotorFunctions** class a more detailed description is given here.

The class methods can be divided in three parts: the motor configuration methods, the data conversion methods and the system operation methods. The first are all the methods programmed to set up the motor with an initial configuration, the second are the methods that take care of data conversion between real world units and motor units, and the third are the methods to operate the calibration system.

4.1 Motor configuration

The configuration methods are the following:

```
public:
  //Initialize the motor the first time it is connected to
  //the power supply.
  int initMotor();
  //Read again the parameters.cfg file and reconfigure the motor
  //calling configMotor().
  int updateMotorConfig();
```

```
  //To reset the communication client.
  void setCommClient(Comm * commClient);
  //Get axis parameter from the motor.
  int32_t getMotorParameter(string parameter, bool printOutput);
  //Set axis parameter.
  void setMotorParameter(string parameter, int32_t value);
  //Configure the speed, acceleration
  //and ramp mode for positioning.
  void configSpeedAndAcceleration();
  //Configure the reference search speed.
  void configReferenceSearchSpeed();
  //Reset all settings in the EEPROM to their factory defaults.
  void restoreFactorySettings();
  //Restart the CPU of the module (like a power cycle).
  void softwareReset();

private:
  //All basic configuration of the motor is done here.
  int configMotor();
  //Configure the encoder prescaler based on
  //the microsteps selected.
  void configEncoderPrescaler(int ustep);
  //Read the config parameters from the configuration file.
  void readConfigParamsFromFile();
  //Read every axis parameter from the motor
  //and print them on the screen.
  void getMotorConfiguration();
  //Read axis parameters from the motor and save them.
  void saveMotorConfigurationToFile(string fileName);
```

These methods take care of setting up the motor for the first use, reconfigure the motor reloading the configuration parameters from the parameters.cfg file, getting all axis parameters and printing them on the screen or saving them on a file, etc.

4.2 Data conversion

The data conversion methods are necessary to change the real world units of position (mm), speed (mm/s), acceleration (mm/s^2), etc, to the stepper motor internal units. The list of methods is shown below:

```
public:
  //Convert velocity in mm/s to motor value.
  int speedMms2Val(int mms);
  //Converts acceleration from mm/s^2 to motor value.
  int accelerationMms2Val(int mms);

private:
  //Convert the microstep resolution value into
  //the microstep resolution byte.
  int uStepRes2Val(int uStepRes);
  //Convert velocity in mm/s to rps.
  int speedMms2Rps(int mms);
  //Converts distance from millimeters to motor value (microsteps).
  int positionMM2Val(float mm);
  //Converts distance from motor value (microsteps) to millimeters.
  float positionVal2MM(int val);
  //Converts distance from motor value (microsteps) to micrometers.
  int positionVal2UM(int val);
```

4.2.1 Speed and Acceleration conversion

The speed conversion is performed from mm/s to the motor value (integer 0..2047) with the following equations:

$$\begin{cases} v_{pps} = \frac{16 \cdot 10^6 \cdot val}{2^{pulse\_div} \cdot 2048 \cdot 32} & \text{(ustep/s)} \\ v_{rps} = \frac{v_{pps}}{F \cdot uF} & \text{(rev/s)} \\ v_{mms} = v_{rps} \cdot L & \text{(mm/s)} \end{cases}$$

Where val is the speed value in motor units, $pulse\_div$ is the pulse divisor axis parameter, F the fullstep resolution (200 steps/rev), uF microstep resolution (ustep/step), and L is the ball screw lead (5mm/rev for this system).

The conversion from mm/s to motor value is given by Equation 1.

$$v_{mms} = \frac{16 \cdot 10^6 \cdot val \cdot L}{2^{pulse\_div} \cdot 2048 \cdot 32 \cdot F \cdot uF} \quad \text{(mm/s)} \tag{1}$$

The conversion from the motor value to mm/s is given by Equation 2.

$$val = \frac{v_{mms} \cdot 2^{pulse\_div} \cdot 2048 \cdot 32 \cdot F \cdot uF}{16 \cdot 10^6 \cdot L} \tag{2}$$

Regarding the acceleration, the conversion is from mm/s^2 to stepper motor value (integer 0..2047). The equations are equivalent.

$$\begin{cases} a_{pps} = \frac{(16 \cdot 10^6)^2 \cdot val}{2^{pulse\_div+ramp\_div+29}} & \\ a_{rps} = \frac{a_{pps}}{F \cdot uF} & (\text{rev}/s^2) \\ a_{mms} = a_{rps} \cdot L & (\text{mm}/s^2) \end{cases}$$

The conversion from mm/s^2 to motor value is given by Equation 3.

$$a_{mms} = \frac{(16 \cdot 10^6)^2 \cdot val \cdot L}{2^{pulse\_div+ramp\_div+29} \cdot F \cdot uF} \quad (\text{mm}/s^2) \tag{3}$$

The conversion from the motor value to mm/s^2 is given by Equation 4.

$$val = \frac{a_{mms} \cdot 2^{pulse\_div+ramp\_div+29} \cdot F \cdot uF}{(16 \cdot 10^6)^2 \cdot L} \tag{4}$$

4.2.2 Distance conversion

The distance or position conversion from mm to microstep and vice versa is computed with the following equations.

$$\begin{cases} p_{mm} = \frac{p_{ustep} \cdot L}{F \cdot uF} & (\text{mm}) \\ p_{ustep} = \frac{p_{mm} \cdot F \cdot uF}{L} & (\text{microstep}) \end{cases}$$

Where p_{ustep} is the position value in microstep and p_{mm} is the position value in millimeters.

4.3 System operation

There are the methods to operate the system to perform the calibration of the receivers.

```
public:
  //Move to Sky Position
  float skyPosition();
  //Move to Cold Load Position
  float coldLoadPosition();
  //Move to Hot Load Position
  float hotLoadPosition();
  //Get the position with a code: sky, hot, cold, unknown.
  string getPosition();
  //Reference the motor to the left limit switch,
  //and go to the reference position
  //(refPosition()) X millimeter to the right of
  //the left limit switch.
  float referenceSearch();
  //Move the reference position. After reference search.
  float refPosition(float mm);
  //Move the load to an absolute position in millimeters.
  //The function does the conversion
  //from millimeters to microstep.
  float moveToPosition(float mm);

private:
  //Send a command and get a reply from the stepper motor.
  void sendCommandAndCheckReply(unsigned char addr,
        unsigned char commandNum,
        unsigned char typeNum,
        unsigned char motorNum,
        int32_t value);
```

5 User interface

In this section the usage of the classes is described. First of all, it is necessary to create ofstream object for the logger. Then a communication client (instance of Comm class), a command generator (instance of StepperCommand class) and a motor object (instance of MotorFunctions class). The StepperCommand, Comm and ofstream objects are passed as a parameter to the motor object. Then, the system operation is commanded with the motor instance.

After creating the motor object, it is necessary to initialize it with initMotor() method, then, the system is ready to start switching between positions. A short example is shown below.

```
//Create log file and open it:
string logFileName = "CL.log";
ofstream logFileD;
openLogFile(&logFileD, logFileName);
writeLogLine(&logFileD, "main.cc", "Start Main Program.");
//Create the communicationClient object:
Comm commClient = Comm(&logFileD);
//Create commandGenerator object:
StepperCommand commandGen = StepperCommand();
//Create motor object:
MotorFunctions motor = MotorFunctions(&commandGen,
                                      &commClient,
                                      &logFileD);

motor.initMotor(); //Initialize the motor.

motor.coldLoadPosition(); //Move to Cold Load Position
position = motor.getPosition(); //Ask the position
printf("Position:%s\n", position.c_str());

//It is possible to change a parameter in
//parameters.cfg and then reload the motor
//configuration from parameters.cfg file.
motor.updateMotorConfig();

motor.skyPosition(); //Move to Sky position
position = motor.getPosition(); //Ask the position
printf("Position:%s\n", position.c_str());

//Do a reference search to position the motor in the
//default position = Sky
motor.referenceSearch();
```

The user interface functions to operate the system (MotorFunctions class) are:

- **int initMotor(): parameters:** none, **return value:** 0 if no error
- **float skyPosition(): parameters:** none, **return value:** position deviation (mm)

- **float coldLoadPosition(): parameters:** none, **return value:** position deviation (mm)
- **float hotLoadPosition(): parameters:** none, **return value:** position deviation (mm)
- **string getPosition(): parameters:** none, **return value:** sky|hot|cold|unknown
- **float referenceSearch(): parameters:** none, **return value:** position deviation (mm)
- **void setCommClient(Comm * commClient): parameters:** instance pointer of Comm, **return value:** none
- **int updateMotorConfig(): parameters:** none, **return value:** 0 if no error

6 ACS Component

The control system of the radio-telescopes at Yebes Observatory uses the ALMA Common Software (ACS). One of its main features is the posibility to use three different programming languages (C++, Java, and Python) with full compatibility between them. This software uses a component-container where the developed tools are components and they are managed with containers. Then, to integrate the calibration load in the ACS control system a new C++ component has been developed. This component uses the functions mentioned in section 5 so other parts of the control system are able to use the calibration load automatically. The functions implemented in this component are:

- **setLoadPosition(pos)**: Tells the calibration load to move to the selected position. Options: "SKY", "COLD" y "HOT".
- **getLoadPosition()**: Asks for the calibration load position. The possible responses are "SKY", "COLD" y "HOT".
- **referenceSearch()**: Tells the calibration load system to perform a reference search to the left limit switch.
- **reloadSocket()**: Closes the actual TCP connection with the calibration load system and opens a new one.

7 Software version

The latest version of the motor software and the ACS components are listed below.

- **Motor software:**
Author: Gabriel Gómez Molina Version: R1.8.0

- **ACS component:**
Author: Fco. Javier Beltrán Martínez Version: 1.6.2