Quick Start Guide


Contact: robots@cognitionfactory.de

This document, the robot & containing software contains confidential and proprietary information some or all of which may be legally privileged. If you have received a copy of this document in error, please notify the author immediately by sending an email to the email address provided above. You must not use, disclose, distribute, copy, or print this document.

How to use the robot

Power on

The only thing you have to do is to switch on the power on the backside of the robot and plug in a LAN cable. The robot uses DHCP to get an IP address from your router. Please look up the robot's IP address in your router's web interface.

Additionally, the robot has the fixed IP address 10.10.10.2.

SSH

You can connect to the robot using SSH. On Windows you can use the PuTTY http://www.putty.org client, on Linux or OSX open a new terminal window and type:

```
$ ssh remote@<ROBOT_IP>
Linux a10Lime 3.4.67+ #4 PREEMPT Thu May 15 09:05:44 CEST 2014 armv7l
remote@robot:~$
```

In the home directory of the remote user you will find the example program client.py and this README.

Power off

Connect to the robot using SSH and execute halt:

```
$ ssh remote@<ROBOT_IP>
remote@robot:~$ sudo halt
```

After that, you can switch off the robot.

**Caution:** The robot has no breaks. If you switch off the power, the robot will collapse which may damage the linear motor. Please make sure you have parked the robot in a safe position before you power off the machine.
Coordinate system of the robot

The robot's effectors can be moved in space. The world frame originates in the robot's base with z pointing upwards, x to the front and y to the right when looking from the front at the robot.

Units

- X, Y and Z coordinates have to be given in **meters**. Use floats here.
- Joints: **radians**. Use floats here. Settings q0, q1 and q2 to 0.0 lets the robot stand in an upright position.

Programming the robot

Using Python

Python Client

You can use an example program `client.py` file (you can find it in the home directory on the robot).

```
$ ssh remote@<ROBOT_IP>
remote@robot:~$ python client.py
```

You can use this client directly on the robot or on a connected computer. The client requires ZMQ to be installed on your system. Please look up the installation instructions on the web: [http://zeromq.org/bindings:python](http://zeromq.org/bindings:python).

API Overview

The python client communicates with the robot using JSON. Each command will return a python dictionary. The dictionary always contains a key `status`. On success, this value will be 'OK', on error 'NOK' plus an description of the error. Please see the example below:

```
>>> from client import RobotClient
>>> client = RobotClient('localhost')
>>> client.robot_start()
{u'status': u'OK'}
>>> client.robot_move_to_xyz(1.0, 0.25, 0.3, 'wrist', speed=1.0)
{u'error': u'Inverse position failed', u'status': u'NOK'}
>>> client.robot_get_position('wrist')
{u'y': -0.0007404142525047064, u'x': 0.24527621269226074, u'z': 0.2944899797439575, u'status': u'OK'}
```

The following list gives an overview of the available commands:
robot_start(): Give power to the robot's motors.
robot_stop(): Remove power from the robot's motors.

robot_move_to_xyz(x, y, z, effector, speed=1.0, acceleration=1.0):
Move one of the robot's end effectors (wrist or pipet) to a point in space.

robot_set_joints(q0, q1, q2): Set robot joints in radians.
robot_get_joints(): Get the current joint configuration.

robot_get_position(effector): Get the position of the robot's effector. Returns a dictionary with x, y and z as keys.

C++ Development
All libraries and headers are installed on the robot in /opt/cf/library. In your home folder you can find some samples (Source Code and CMake files) that show how to use the library and compile programs that link to the libraries.

API Overview
The robot can be controlled through the class cf::r0::Robot:

```cpp
Robot(cf::kin::Kinematic* kinematic,
    const std::string& controllerdevice,
    const double& updateRate,
    const cf::hal::DynamixelDevice::BaudRate& baudRate,
    const bool debug = false);

enum MoveToResult
{
    GoalReached,
    Aborted,
    Error
};

// load a configuration file with kinematic parameters
void loadConfigurationFile(const std::string& filename);
virtual void loadConfiguration(const boost::property_tree::ptree& tree);

//set correction values to the joint angles if required
void setCalibrationError(const cf::math::Vector& error);

// start the robot
bool start();

// stop the robot
bool stop();

// move the robot to a cartesian position (pose) with a given effector in a give mode (in configuration or operational space)
MoveToResult moveTo(const cf::math::Pose& pose,
    const cf::math::Vector& error);
```
const ::std::string& effector,
const Mode& mode);

// move the robot to a joint configuration
MoveToResult moveToConfigurationSpace(const ::cf::math::Vector&
q);

// move the robot to a cartesian position (pose) with a given
// effector
MoveToResult moveToOperationalSpace(const ::cf::math::Pose&
pose,
const ::std::string&
effector);

// set speed of the robot motions from 0.0 to 1.0
void setRobotSpeed(const double& speed);

// set the acceleration of the motors
void setRobotAcceleration(const double& acceleration);

// returns the current position of the wrist
::cf::math::Pose getCurrentWristPose();

// returns the current position of the selected effector
::cf::math::Pose getCurrentEffectorPose(const ::std::string&
effector);

// returns the current joint position (angles)
::cf::math::Vector getJointPosition() const;

// check, if a position is reachable with the robot
bool isReachable(const ::cf::math::Pose& pose,
const ::std::string& effector);

// compute the error between actual and expected pose.
void calculateError(const ::cf::math::Pose& pose,
const ::std::string& effector,
::cf::math::Vector& jointConfigurationError,
::cf::math::Pose& currentPose);

// returns the number of registered effectors
const ::std::size_t getNrEffectors() const;

// returns the names of registered effectors
const ::std::vector< ::std::string > getEffectorNames() const;

// register new effectors
void setEffectors(const EffectorMap& effectors);

// check, if effector exists
bool hasEffector(const ::std::string& effector) const;
// enable / disable the motor torque
bool enableTorque(const bool & enabled);

You can also use the ZMQ messages (see Python) with C++ to send commands to the robot.

Additionally, low level access to the motors is available.

**Autostart**

There are programs that are used to control the robot and start automatically while booting the system using inittab.

- `r0-node`: The robot control program (daemon).
- `robot.py`: A small python wrapper to control the robot (i.e. the server for `client.py`).

You can disable the autostart by commenting the corresponding lines in `/etc/inittab`.

**Configuration files**

All configuration reside inside `/opt/cf/library/share/config` and should be self-explanatory.

**Serial devices**

- `/dev/ttyS1`: Dynamixel Device (RS-485)
- `/dev/ttyS2`: RS-232 1
- `/dev/ttyS2`: RS-232 2