CMU - Robotics Institute **Manipulation Lab - Simple Hands Project**

Hand node - ROS - Introduction

<manipulationLab@gmail.edu> October 13, 2011

1 Hand

Hand_node is a ROS interface to control the gripper in the Simple Hands project implemented in C++. It provides abstraction to command the motor controller of the hand and read the finger encoders through an Arduino microcontroller.

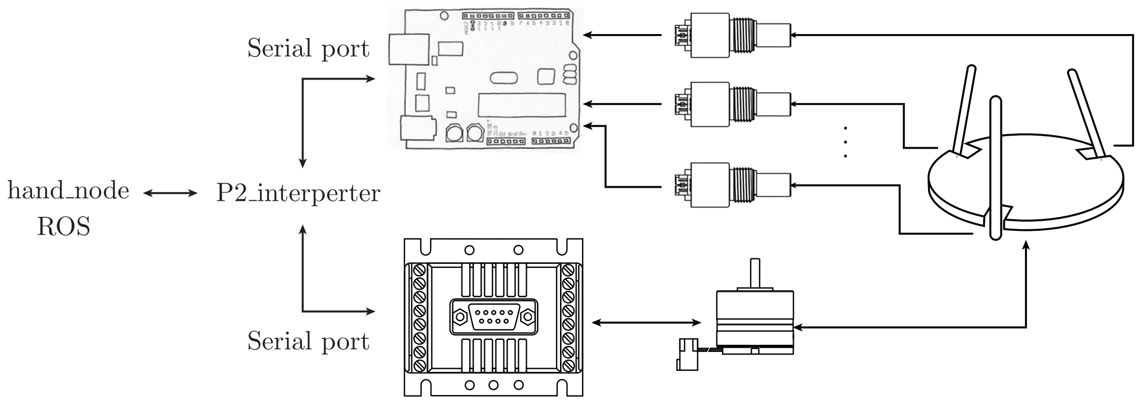

Figure 1: Schematic of the control of the hand.

It is composed of two ROS packages: hand_node, a standalone typical ROS package that provides topics and services, and hand_comm, including the message definitions and a C++ client class to simplify the invocation and communication with hand_node.

1.1 hand_node

(Located at `svnroot/code/nodes/hand/ROS/hand_node`)

Services:

- **hand_Calibrate**: Blocking service to calibrate the motor encoder. The hand opens until reaching a hard stop and homes the motor at that position (motor encoder set to 0). This service needs to be called every time that the hand is switched on, since the motor encoder is not absolute and loses the encoder count when disconnected.

```
int64 encMotor   # Motor encoder
int64[] enc      # List of finger encoders
int64 ret        # Set to 1 (success) or 0 (error)
string msg       # Error description
```

- **hand_SetForce**: Service to set the max force that the hand should be using when opening or closing. Internally, this service limits the max current that the motor should use. The input parameter force should be a real value between 0 and 1. That value will be mapped to a current value between system parameters minIntensity and maxIntensity.

```
float64 force # value from 0 to 1
---
int64 ret     # Set to 1 (success) or 0 (error)
string msg    # Error description
```

- **hand_SetSpeed**: Service to set the speed of the hand. Internally, this service sets the speed of the hand motor. the input parameter speed should be a real value between 0 and 1. That value is mapped to a speed value between system parameters minSpeed and maxSpeed.

```
float64 speed    # Value between 0 and 1
---
int64 ret        # Set to 1 (success) or 0 (error)
string msg       # Error description
```

- **hand_GetEncoders**: Service to get the value of the motor and finger encoders.

```
---
int64 encMotor      # Motor encoder
int64[] encFinger   # finger encoders
int64 ret           # Set to 1 (success) or 0 (error)
string msg          # Error description
```

- **hand_GetAngles**: If the hand is calibrated, this service returns the value of the angles of the motor and fingers.

```
---
float64 angleMotor   # Motor angle
float64[] angle      # Finger angles
int64 ret            # Set to 1 (success) or 0 (error)
string msg           # Error description
```

- **hand_SetEncoder**: Non-blocking service to set the command the position of the motor encoder.

```
int64 enc      # Position of the motor encoder
---
int64 ret      # Set to 1 (success) or 0 (error)
string msg     # Error description
```

- **hand_SetAngle**: Non-blocking service to set the command the position of the motor angle, if the hand is calibrated.

```
float64 angle   # Nominal position of the motor.
                # 0.0 sets the fingers perpendicular to the palm.
---
int64 ret       # Set to 1 (success) or 0 (error)
string msg      # Error description
```

- **hand_Ping**: Service to ping the hand and check if motor and encoders are available.

```
---
int64 ret      # Set to 1 (success) or 0 (error)
string msg     # Error description
```

- **hand_SetRest**: Service to stop the hand motor. Even if the hand is not moving, the motor controller is actively controlling the motor positions to recover form any external deviations. It is good practice to set the motor to rest when we no longer care, in order to avoid overheating of the motor.

```
---
int64 ret     # Set to 1 (success) or 0 (error)
string msg    # Error description
```

- **hand_WaitRest**: Blocking service to wait until the hand stops. The hand stops when it gets to the commanded motor position, or when it gets stuck before getting there for more than half a second. Delay is an input parameter that specifies how long the service should wait until checking for the hand to be stopped. This is useful, for example when we issue a hand_WaitRest right after a hand_SetEncoder, since it takes a while for the hand to start moving.

```
float64 delay    # Seconds to wait before begin to check for rest
---
int64 ret        # Set to 1 (success) or 0 (error)
string msg       # Error description
```

- **hand_IsMoving**: Service to check whether the hand is moving or not.

```
---
int64 moving      # 1 (moving) 0 (not)
int64 ret         # Set to 1 (success) or 0 (error)
string msg        # Error description
```

Topics:

- **hand_EncodersLog**: Topic broadcasted at 40Hz with the motor and finger encoder values.

  ```
  float64 timeStamp    # ROS time-stamp
  int64 encMotor       # Motor encoder
  int64[] encFinger    # Finger encoders
  ```

- **hand_AnglesLog**: Topic broadcasted at 40Hz with the motor and finger angles. Only published when the hand is calibrated.

  ```
  float64 timeStamp      # ROS time-stamp
  float64 angleMotor     # Motor angle
  float64[] angle        # Finger angles
  ```

1.2 hand_comm

(Located at svnroot/code/nodes/hand/ROS/hand_comm)
Hand_comm is a ROS wrapped C++ class meant to simplify the communication with hand_node. It contains:

1. Message and Service definitions. When creating a ROS application that uses hand_node, the application only needs to be dependent on hand_comm. As a consequence, the application does not need to link to all the drivers and libraries needed to control the hand.
2. C++ class HandComm that handles all configuration and initialization required to use hand_node. As an example, if we want to call the service hand_Ping, we only need instantiate a HandComm object and call to the member routine handPing. Otherwise we would have to include all required header files, subscribe to the hand_Ping service and format the required message to be sent to the service.

Usage example:

```
...
ros::NodeHandle node;

HandComm hand;            //Create HandComm client
hand.subscribe(&node);    //Subscribe to all services of hand_node
while(!hand.Ping());      //Wait until hand is ready to move
hand.Calibrate();         //Calibrate the hand.

...
```

Member routines:

```
class HandComm
{
  public:

    HandComm();
    HandComm(ros::NodeHandle* np);
    ~HandComm();

    // Subscribe Function
    void subscribe(ros::NodeHandle* np);

    // Subscribe to Topics
    void subscribeAngles(ros::NodeHandle* np, int q_len,
        void (*funcPtr)(const hand_comm::hand_AnglesLogConstPtr&));
    void subscribeEncoders(ros::NodeHandle* np, int q_len,
        void (*funcPtr)(const hand_comm::hand_EncodersLogConstPtr&));

    // Shutdown service clients
    void shutdown();

    bool Ping();
    bool Calibrate();
    bool GetAngles(double &mot_ang, double ang[NUM_FINGERS]);
    bool GetAngles(double &mot_ang, Vec &ang);
    bool IsMoving(bool &moving);
    bool SetEncoder(int enc);
    bool SetRest();
    bool WaitRest(int delay);
    bool GetEncoders(int &mot_enc, int enc[NUM_FINGERS]);
    bool GetEncoders(int &mot_enc, Vec &enc);
    bool SetAngle(double angle);
    bool SetForce(double force);
    bool SetSpeed(double speed);

  private:
...
};
```