MITMECHE

Introduction to Robot Operating System and Raspberry Pi

Department of Mechanical Engineering
Massachusetts Institute of Technology

MITMECHE

AsiaUS Business Connection

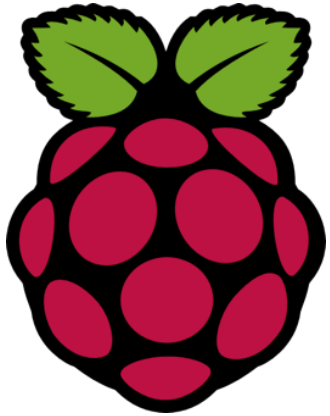
Workshop TiM THE BEAVER

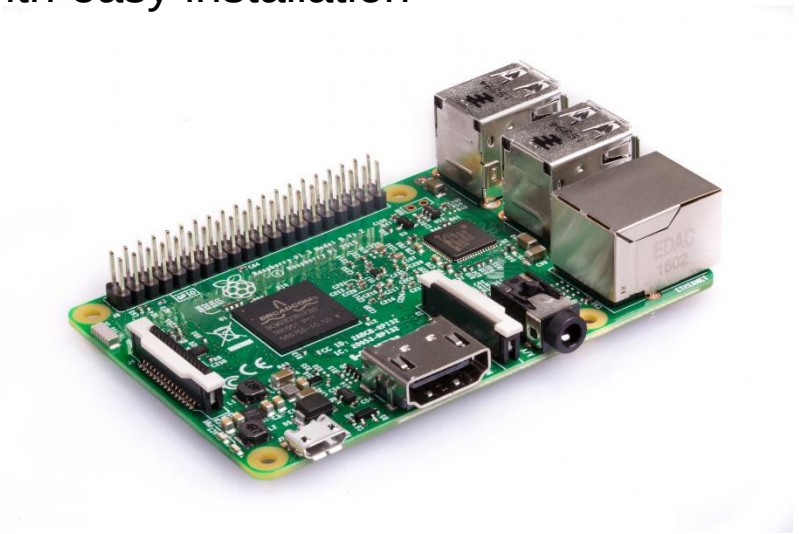
Outline

- Raspberry PI Overview
  - Project applications
  - Hardware resources
  - Quick start
- Introduction to Linux
  - Operating system and command line
  - Linux overview
- ROS
  - What is ROS
  - ROS topics, messages
  - ROS model of robotic arm with transformation package
  - ROS visualization
  - ROS robotic arm control

Raspberry PI

- Developed at University of Cambridge's computer lab for education
- Credit card sized PC with low cost at around $40
- Capability: programming, video playing, electronic projects
- SD card as storage for easy swapping of operating system
- Helpful for learning ROS with easy installation



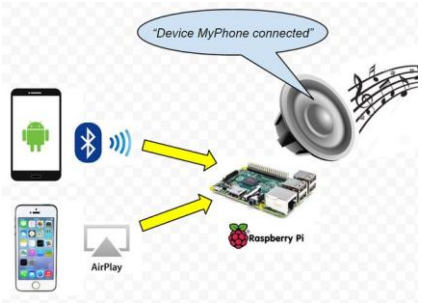


Project Applications

Motion sensor alarm system

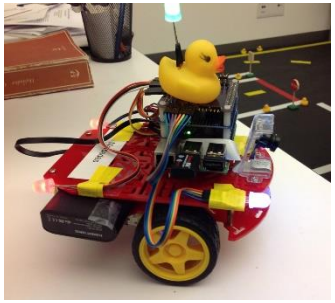
Game system

Mini tank



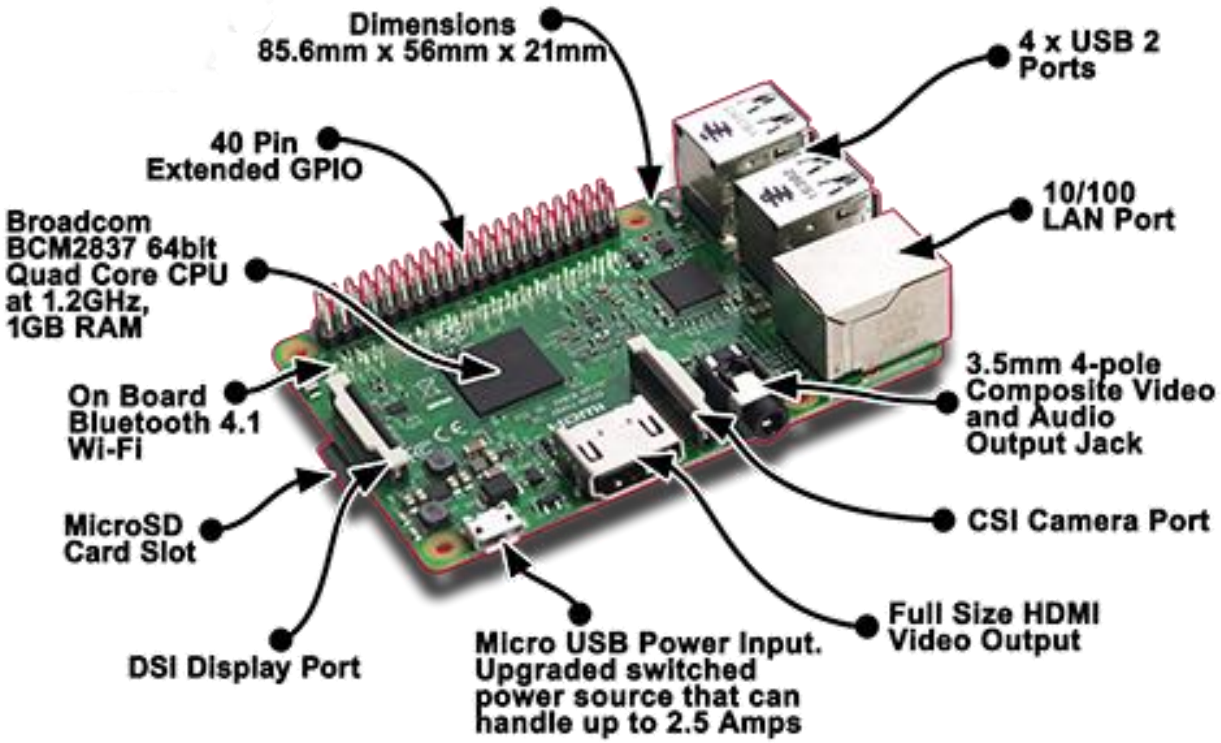
Bluetooth audio player

Autonomous boat

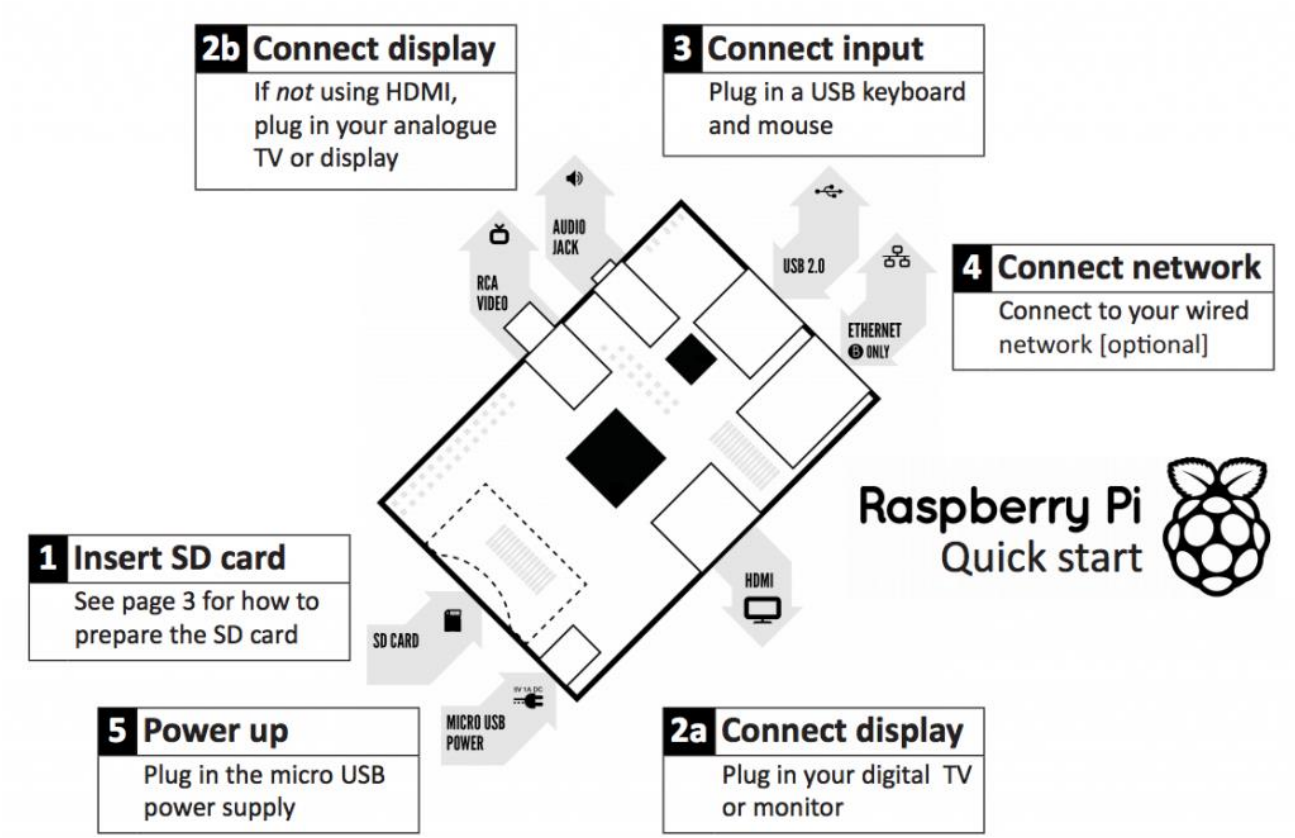


MIT Duckie Town

Raspberry Pi Resources



Raspberry Pi Quick Start

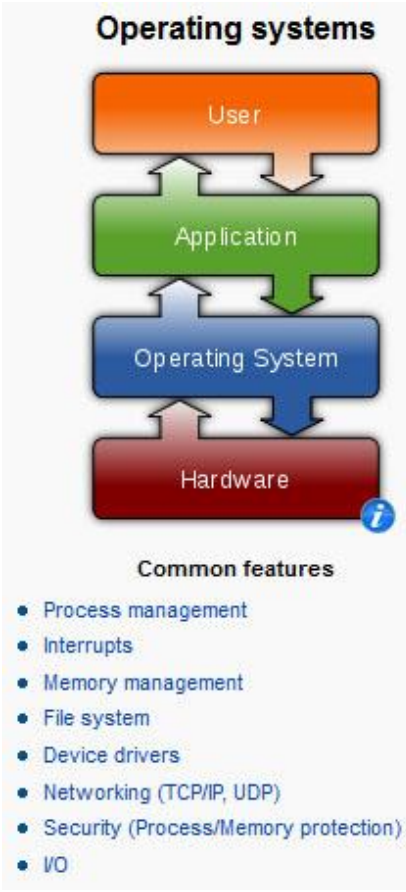


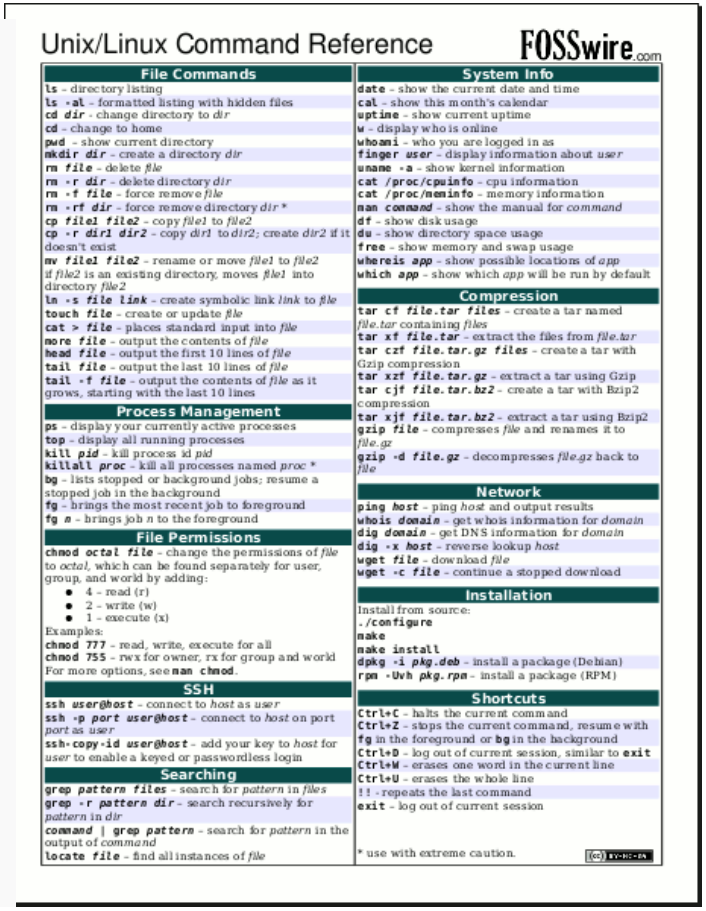
MITMECHE

AsiaUS Business Connection

Workshop TiM THE BEAVER

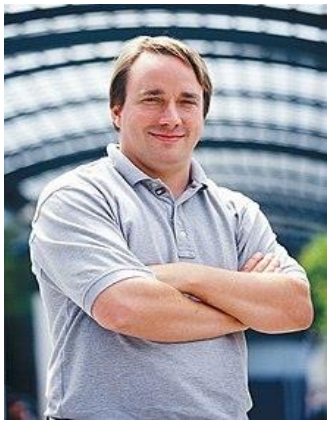
Operating System and Command Line





Linux Overview

- Linux is a Unix clone written from scratch by Linus Torvalds with assistance from a loosely-knit team of hackers across the Net
- Unix is a multitasking, multi-user computer operating system originally developed in 1969 by a group of AT&T employees at Bell Labs.
- 64% of the world's servers run some variant of Unix or Linux. The Android phone and the Kindle run Linux.



MITMECHE

AsiaUS Business Connection

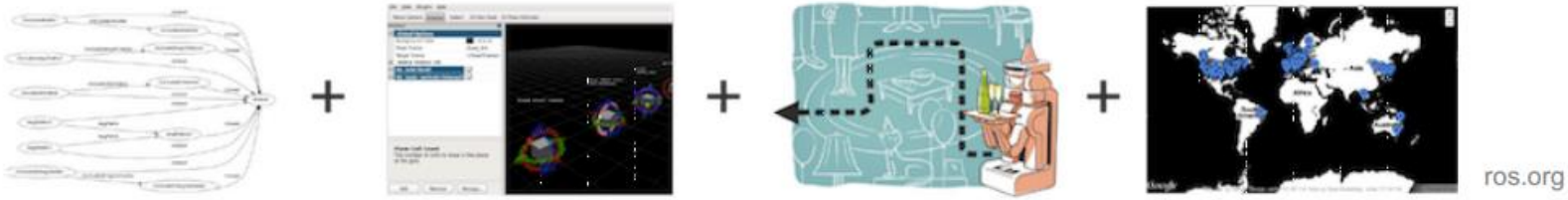
Workshop TIM THE BEAVER

Introduction to ROS

- ROS Basic
  - What is ROS
  - ROS architecture: core, nodes, topics, messages
  - ROS model of robotic arm with transformation package
  - ROS visualization
  - ROS robotic arm control
- Additional Tools
  - ROS services
  - ROS action
  - ROS time
  - ROS bags
- Many tutorials available and ROS official website

What is ROS?

ROS = Robot Operating System



Plumbing	Tools	Capabilities	Ecosystem
▪ Process management ▪ Inter-process communication ▪ Device drivers	▪ Simulation ▪ Visualization ▪ Graphical user interface ▪ Data logging	▪ Control ▪ Planning ▪ Perception ▪ Mapping ▪ Manipulation	▪ Package organization ▪ Software distribution ▪ Documentation ▪ Tutorials

History of ROS

- Originally developed in 2007 at the Stanford Artificial Intelligence Laboratory
- Since 2013 managed by OSRF
- Today used by many robots, universities and companies
- De facto standard for robot programming

ROS Philosophy

- **Peer to peer**
  Individual programs communicate over defined API (ROS *messages*, *services*, etc.).
- **Distributed**
  Programs can be run on multiple computers and communicate over the network.
- **Multi-lingual**
  ROS modules can be written in any language for which a client library exists (C++, Python, MATLAB, Java, etc.).
- **Light-weight**
  Stand-alone libraries are wrapped around with a thin ROS layer.
- **Free and open-source**
  Most ROS software is open-source and free to use.

ROS Workspace

- Defines context for the current workspace
- Default workspace loaded with

```
> source /opt/ros/indigo/setup.bash
```

depends on version
“kinetic” installed now

Overlay your *catkin workspace* with

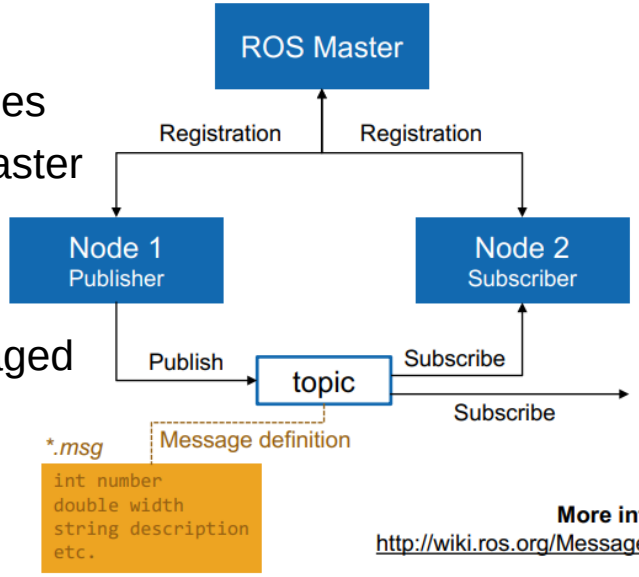
```
> cd ~/catkin_ws
> source devel/setup.bash
```

Check your workspace with

```
> echo $ROS_PACKAGE_PATH
```

ROS Architecture

- ROS Master
  - Manages the communication between nodes
  - Every node registers at startup with the master
- ROS Nodes (Packages)
  - Single-purpose, executable program
  - Individually compiled, executed, and managed
- ROS Topics
  - can publish or subscribe to a topic
  - Often 1 publisher and n subscribers
  - Topic is a name for a stream of messages
- ROS Messages (*.msg files)
  - Data structure defining the type of a topic
  - Compromised of a nested structure of data



ROS Basic Commands

Start a master with

```
> roscore
```

Run a node with

```
> rosrun package_name node_name
```

See active nodes with

```
> rosnode list
```

Retrieve information about a node with

```
> rosnode info node_name
```

See the type of a topic

```
> rostopic type /topic
```

Publish a message to a topic

```
> rostopic pub /topic type args
```

List active topics with

```
> rostopic list
```

Subscribe and print the contents of a topic with

```
> rostopic echo /topic
```

Show information about a topic with

```
> rostopic info /topic
```

geometry_msgs/Point.msg

```
float64 x
float64 y
float64 z
```

sensor_msgs/Image.msg

```
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
uint32 height
uint32 width
string encoding
uint8 is_bigendian
uint32 step
uint8[] data
```

geometry_msgs/PoseStamped.msg

```
std_msgs/Header header
 uint32 seq
 time stamp
 string frame_id
geometry_msgs/Pose pose
  geometry_msgs/Point position
     float64 x
     float64 y
     float64 z
  geometry_msgs/Quaternion
orientation
     float64 x
     float64 y
     float64 z
     float64 w
```

ROS Messages

ROS Workspace

- *catkin* is the ROS build system to generate executables, libraries, and interfaces
- We suggest to use the *Catkin Command Line Tools*

→ Use **`catkin build`** instead of `catkin_make`

Navigate to your catkin workspace with

```
> cd ~/catkin_ws
```

Build a package with

```
> catkin build package_name
```

! Whenever you build a **new** package, update your environment

```
> source devel/setup.bash
```

If necessary, clean the entire build and devel space with

```
> catkin clean
```

Work here

src

The *source space* contains the source code. This is where you can clone, create, and edit source code for the packages you want to build.

Don't touch

devel

The *development (devel) space* is where built targets are placed (prior to being installed).

Don't touch

build

The *build space* is where CMake is invoked to build the packages in the source space. Cache information and other intermediate files are kept here.

ROS Launch

- *launch* is a tool for launching multiple nodes (as well as setting parameters)
- Are written in XML as **.launch* files
- If not yet running, launch automatically starts a roscore

Browse to the folder and start a launch file with

```
> roslaunch file_name.launch
```

Start a launch file from a package with

```
> roslaunch package_name file_name.launch
```

range_world.launch (simplified)

```
<?xml version="1.0"?>
<launch>
  <arg name="use_sim_time" default="true"/>
  <arg name="world" default="gazebo_ros_range"/>
  <arg name="debug" default="false"/>
  <arg name="physics" default="ode"/>

  <group if="$(arg use_sim_time)">
    <param name="/use_sim_time" value="true" />
  </group>

  <include file="$(find gazebo_ros)
                              /launch/empty_world.launch">
    <arg name="world_name" value="$(find gazebo_plugins)/
                    test/test_worlds/$(arg world).world"/>
    <arg name="debug" value="$(arg debug)"/>
    <arg name="physics" value="$(arg physics)"/>
  </include>
</launch>
```

Create re-usable launch files with tag, which works like a parameter (default optional)

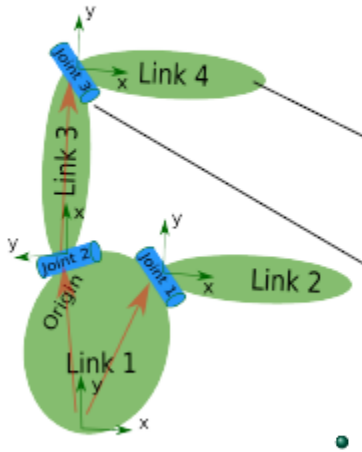
When launching, arguments can be set with

```
> roslaunch launch_file.launch arg_name:=value
```

- **launch**: Root element of the launch file
- **node**: Each *<node>* tag specifies a node to be launched
- **name**: Name of the node (free to choose)
- **pkg**: Package containing the node
- **type**: Type of the node, there must be a corresponding executable with the same name
- **output**: Specifies where to output log messages (`screen`: console, `log`: log file)

Unified Robot Description Format Model

- Description consists of a set of *link* elements and a set of *joint* elements
- Joints connect the links together



robot.urdf

```
<robot name="robot">
  <link> ... </link>
  <link> ... </link>
  <link> ... </link>

  <joint> .... </joint>
  <joint> .... </joint>
  <joint> .... </joint>
</robot>
```

```
<link name="link_name">
  <visual>
    <geometry>
      <mesh filename="mesh.dae"/>
    </geometry>
  </visual>
  <collision>
    <geometry>
      <cylinder length="0.6" radius="0.2"/>
    </geometry>
  </collision>
  <inertial>
    <mass value="10"/>
    <inertia ixx="0.4" ixy="0.0" …/>
  </inertial>
</link>
```

```
<joint name="joint_name" type="revolute">
  <axis xyz="0 0 1"/>
  <limit effort="1000.0" upper="0.548" … />
  <origin rpy="0 0 0" xyz="0.2 0.01 0"/>
  <parent link="parent_link_name"/>
  <child link="child_link_name"/>
</joint>
```

More info
http://wiki.ros.org/urdf/XML/model

- XML format
- Tags: link, joint, transmission, ...
- Kinematic tree structure
- Order in the file does not matter
  - One file per hand type, per transmission type and per fingertip model
  - Every link and joint is described explicitly

$\Longrightarrow$ a lot of redundancy, very long files, hard to read and hard to maintain, etc...

XML Macro language

- Increase modularity
- Reduce redundancy
- Permit Parametrization
- Generate URDF on-the-fly

- Inclusion
- Macros
- Properties
- Expansion of all xacro statements
- Command line and output to stdout

- Reduce redundant code
- Parametrized entities
- Modularity

- Properties:
  - definition
  - instantiation
  - string concatenation
- Simple math
  - in variables
  - nested variables
  - no function

example

```
<xacro:property name="width" value=".2"/>
<cylinder radius="${width}" length=".1"/>

<link name="${robotname}s_leg" />

<cylinder radius="${diam/2}" length=".1"/>
```

- Simple macro:
  - definition
  - instantiation
- Parametrized macro:
  - definition
  - instantiation
- Nested macros

example

```
<xacro:macro name="default_origin">
 <origin xyz="0 0 0" rpy="0 0 0"/>
</xacro:macro>
<xacro:default_origin />
<xacro:macro name="default_inertial" params="mass">
 <inertial>
  <xacro:default_origin />
  <mass value="${mass}" />
   <inertia ixx="0.4" ixy="0.0" ixz="0.0"
          iyy="0.4" iyz="0.0" izz="0.2"/>
 </inertial>
</xacro:macro>
<xacro:default_inertial mass="10"/>
```

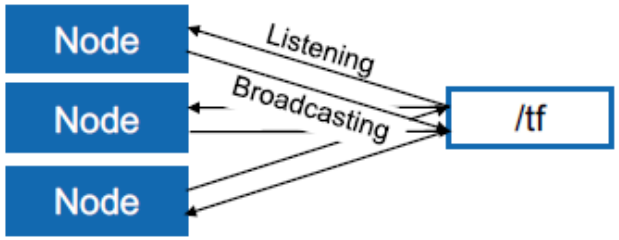
- Default values:
  - Provides default values for optional or repeated parameters
- Conditional statement:
  - Only tests true or false 0 and 1
- Command line argument:
  - xacro.py file.xacro rad:=3

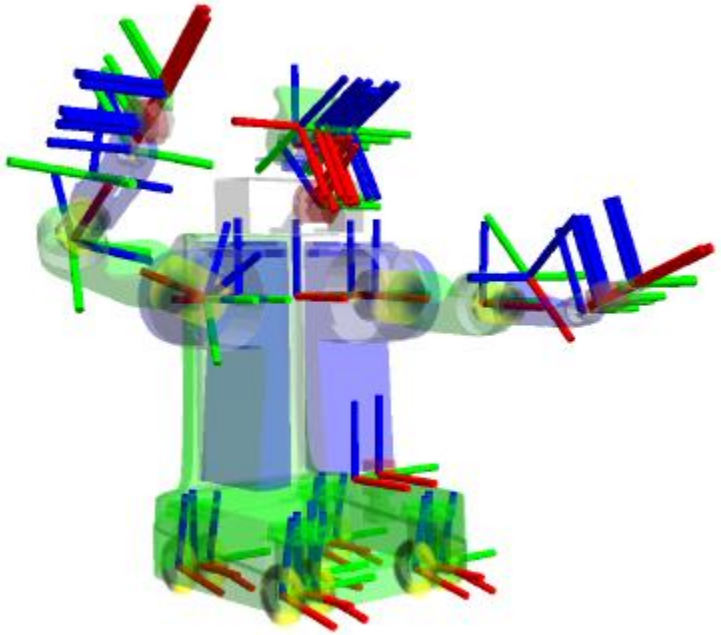
example

```
<xacro:macro name="pos" params="x y:=0"/>
<xacro:pos x="1"/>

<xacro:if value="<expression>">
<xacro:unless value="<expression>">

<xacro:arg name="rad" default="2"/>
<cylinder radius="$(arg rad)" length=".1"/>
```

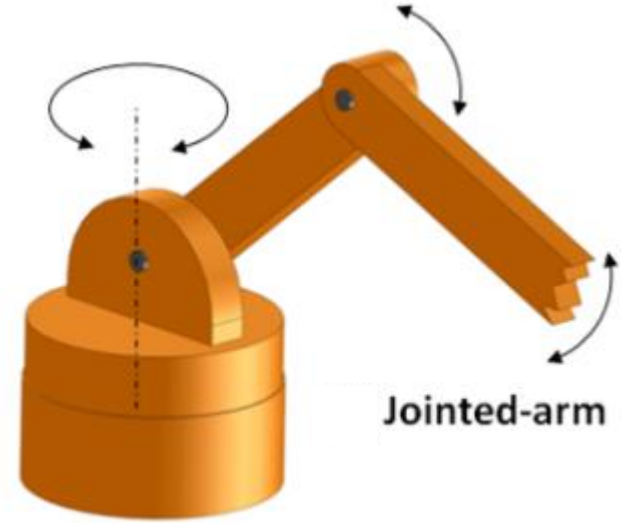
TF Transformation

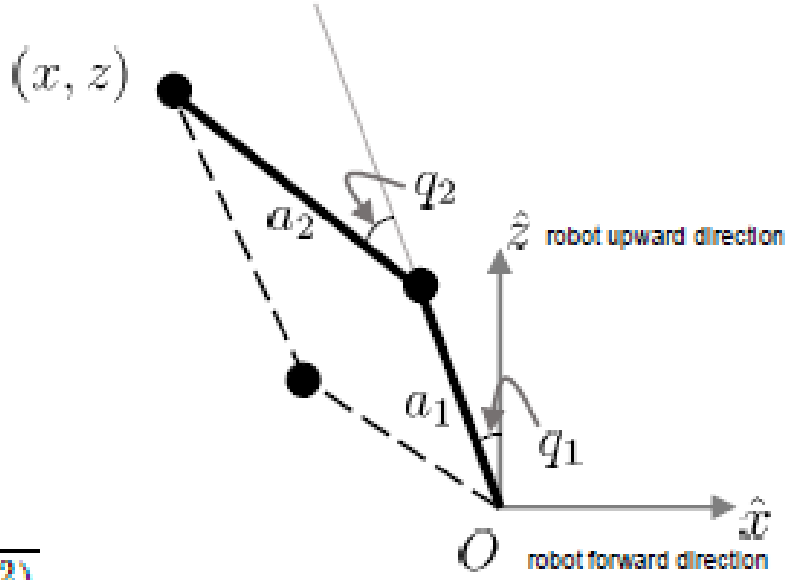
- Tool for keeping track of coordinate frames over time
- Maintains relationship between coordinate frames in a tree structure buffered in time
- Lets the user transform points, vectors, etc. between coordinate frames at desired time
- Implemented as publisher/subscriber model on the topics `/tf` and `/tf_static`





Inverse Kinematics



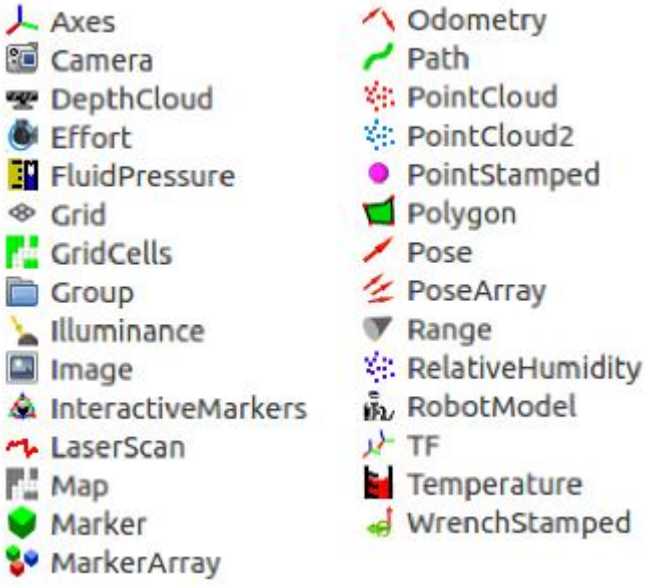


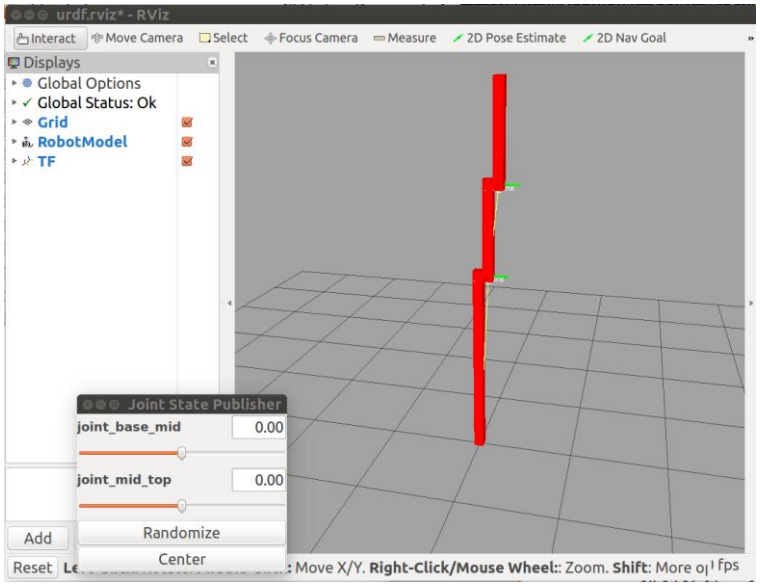
$$q_2 = \pm 2 \tan^{-1} \sqrt{\frac{(a_1 + a_2)^2 - (x^2 + z^2)}{(x^2 + z^2) - (a_1 - a_2)^2}},$$

$$q_1 = \operatorname{atan2}(z, x) - \operatorname{atan2}(a_2 \sin q_2, a_1 + a_2 \cos q_2) - \pi/2,$$

Robot Arm Visualization

- RViz
  - Tool for visualization
  - Many plugins available





ROS Services

- Request/response communication between nodes is realized with *services*
  - The *service server* advertises the service
  - The *service client* accesses this service
- Similar in structure to messages, services are defined in **.srv* files

List available services with

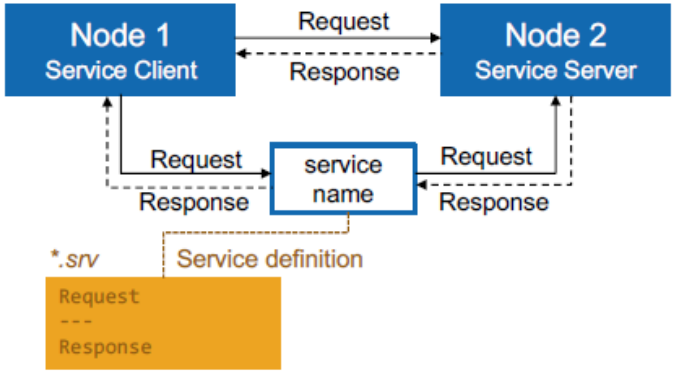
```
> rosservice list
```

Show the type of a service

```
> rosservice type /service_name
```

Call a service with the request contents

```
> rosservice call /service_name args
```



ROS Services

- Request/response communication between nodes is realized with *services*
  - The *service server* advertises the service
  - The *service client* accesses this service
- Similar in structure to messages, services are defined in **.srv* files

List available services with

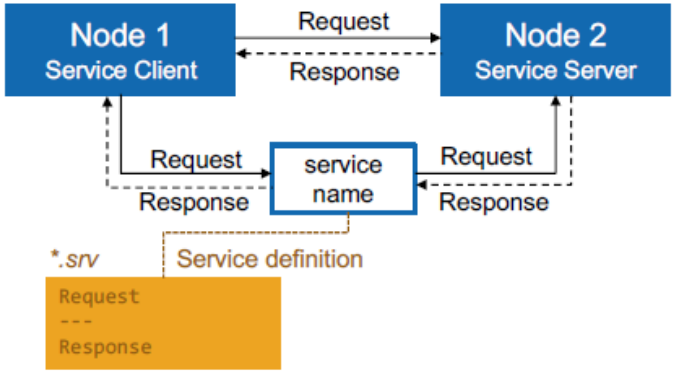
```
> rosservice list
```

Show the type of a service

```
> rosservice type /service_name
```

Call a service with the request contents

```
> rosservice call /service_name args
```



std_srvs/Trigger.srv

```
---
bool success
string message
```

Request

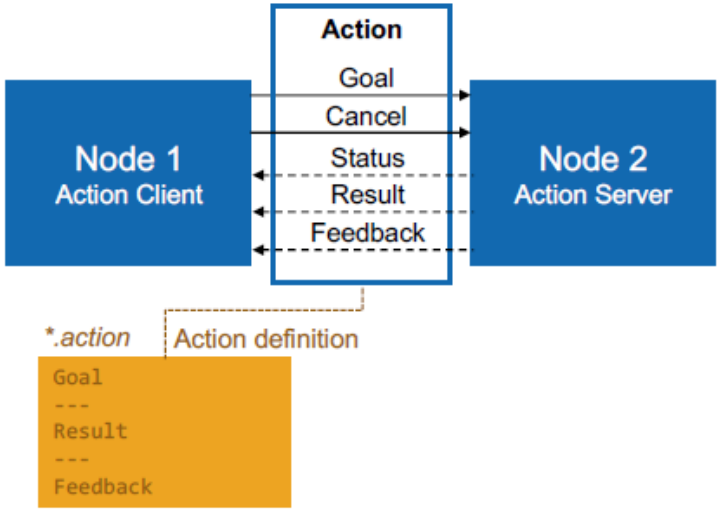
Response

nav_msgs/GetPlan.srv

```
geometry_msgs/PoseStamped start
geometry_msgs/PoseStamped goal
float32 tolerance
---
nav_msgs/Path plan
```

ROS Actions

- Similar to service calls, but provide possibility to
  - Cancel the task (preempt)
  - Receive feedback on the progress
- Best way to implement interfaces to time-extended, goal-oriented behaviors
- Similar in structure to services, action are defined in **.action* files
- Internally, actions are implemented with a set of topics



Averaging.action

```
int32 samples
---
float32 mean
float32 std_dev
---
int32 sample
float32 data
float32 mean
float32 std_dev
```

Goal
Result
Feedback

FollowPath.action

```
navigation_msgs/Path path
---
bool success
---
float32 remaining_distance
float32 initial_distance
```

MITMECHE

AsiaUS Business Connection

Workshop TIM THE BEAVER

ROS Comparison

	Parameters	Dynamic Reconfigure	Topics	Services	Actions
Description	Global constant parameters	Local, changeable parameters	Continuous data streams	Blocking call for processing a request	Non-blocking, preemptable goal oriented tasks
Application	Constant settings	Tuning parameters	One-way continuous data flow	Short triggers or calculations	Task executions and robot actions
Examples	Topic names, camera settings, calibration data, robot setup	Controller parameters	Sensor data, robot state	Trigger change, request state, compute quantity	Navigation, grasping, motion execution

ROS Time

- Normally, ROS uses the PC's system clock as time source (*wall time*)
- For simulations or playback of logged data, it is convenient to work with a simulated time (pause, slow-down etc.)
- To work with a simulated clock:
  - Set the `/use_sim_time` parameter

```
> rosparam set use_sim_time true
```

  - Publish the time on the topic `/clock` from
    - Gazebo (enabled by default)
    - ROS bag (use option `--clock`)

- To take advantage of the simulated time, you should always use the ROS Time APIs:
  - **`ros::Time`**

```
ros::Time begin = ros::Time::now();
double secs = begin.toSec();
```

  - **`ros::Duration`**

```
ros::Duration duration(0.5); // 0.5s
```

  - **`ros::Rate`**

```
ros::Rate rate(10); // 10Hz
```

- If wall time is required, use `ros::WallTime`, `ros::WallDuration`,

ROS Bags

- A *bag* is a format for storing message data
- Binary format with file extension *.bag
- Suited for logging and recording datasets for later visualization and analysis

Record all topics in a bag

```
> rosbag record --all
```

Record given topics

```
> rosbag record topic_1 topic_2 topic_3
```

Stop recording with `Ctrl + C`
Bags are saved with start date and time as file name in the current folder (e.g. `2017-02-07-01-27-13.bag`)

Show information about a bag

```
> rosbag info bag_name.bag
```

Read a bag and publish its contents

```
> rosbag play bag_name.bag
```

Playback options can be defined e.g.

```
> rosbag play --rate=0.5 bag_name.bag
```

`--rate=`*`factor`*	Publish rate factor
`--clock`	Publish the clock time (set param `use_sim_time` to `true`)
`--loop`	Loop playback
	etc.

Exercise

- Walk through ROS publisher and subscriber tutorial at this link

```python
#!/usr/bin/env python
# license removed for brevity
import rospy
from std_msgs.msg import String

def talker():
    pub = rospy.Publisher('chatter', String, queue_size=10)
    rospy.init_node('talker', anonymous=True)
    rate = rospy.Rate(10) # 10hz
    while not rospy.is_shutdown():
        hello_str = "hello world %s" % rospy.get_time()
        rospy.loginfo(hello_str)
        pub.publish(hello_str)
        rate.sleep()

if __name__ == '__main__':
    try:
        talker()
    except rospy.ROSInterruptException:
        pass
```

```python
#!/usr/bin/env python
import rospy
from std_msgs.msg import String

def callback(data):
    rospy.loginfo(rospy.get_caller_id() + "I heard %s", data.data)

def listener():

    # In ROS, nodes are uniquely named. If two nodes with the same
    # node are launched, the previous one is kicked off. The
    # anonymous=True flag means that rospy will choose a unique
    # name for our 'listener' node so that multiple listeners can
    # run simultaneously.
    rospy.init_node('listener', anonymous=True)

    rospy.Subscriber("chatter", String, callback)

    # spin() simply keeps python from exiting until this node is stopped
    rospy.spin()

if __name__ == '__main__':
    listener()
```

Thank You!