

## 4 Основы ROS

# Установка и запуск ROS

Для продолжения нашего курса вам необходимо установить на ваш компьютер ROS.

---

## Установка ROS

Инструкция по установке ROS доступна в видео уроке

<https://www.youtube.com/watch?v=AG9u6eSvqb4>

---

## Информация из видео

Инструкция установки ROS для Ubuntu <http://wiki.ros.org/noetic/Installation/Ubuntu>

Настройка рабочего окружения

<http://wiki.ros.org/ROS/Tutorials/InstallingandConfiguringROSEnvironment>

Запуск ROS


```
roscore
```

Запуск программы turtlesim

```
roslaunch turtlesim turtlesim_node
```

Запуск программы ROS для управления черепашкой

```
roslaunch turtlesim turtlesim_teleop_key
```

 В будущем для выполнения любого задания с ROS необходимо, чтобы работала мастер-нода ROS. Для ее запуска в консоле необходимо запустить программу `roscore`

Если запустить мастер-ноду с командой `roscore &`, то процесс запустится в фоне и консоль останется "свободной" для дальнейшего использования

# Обмен сообщениями

# Краткое введение

Для управления роботом должна существовать некая программа. Чем больше и сложнее робот, тем "больше и сложнее" должна получиться эта программа.

Можно пойти "понятным" нам путем и написать одну очень большую программу, которая будет делать на роботе вообще все. Но отладить и поддерживать такую программу очень сложно. А при огромных требованиях к функционалу современных роботов почти невозможно.

Было принято решение, что на роботе одновременно должно работать множество небольших программ. Каждая такая программа должна выполнять одну или несколько простых функций. В формулировках ROS такая программа называется **нода (node)**. Писать и поддерживать такие мини-программы значительно проще, чем одну большую.

Если мы разделяем одну большую программу на много маленьких, то у нас появляется вопрос, каким образом данные из одной программы (ноды) должны попадать в другие программы. Без обмена данными такие программы будут абсолютно бесполезны.

Для этого в ROS используется несколько разных подходов. Один из самых часто используемых подходов — это протокол, построенный по так называемой модели **PubSub (Издатель и Подписчик)**

Рассмотрим сущности этого подхода.

**Издатель (Publisher)** - это процесс, который имеет какие-либо данные и хочет сообщить о них всему роботу. Например, программа получает данные с датчика температуры, переводим их в понятные нам градусы цельсия и отправляет эти данные в ROS.

**Подписчик (Subscriber)** - это процесс, который хочет получить существующие в системе данные для дальнейшей их обработки. Например, это может быть часть программа (ноды) которая получает данные о температуре, и если температура превышена, то включает вентилятор для охлаждения или выводит эти данные на экран.

Чтобы Издатель и Подписчик могли обмениваться данными, обе программы должны понимать, где "брать" и куда "складывать" данные. Для этого в модели PubSub вводится понятие **Topic** (Тема), которое идентифицируется простым текстовым названием (и служит некоторым адресом, где данные находятся).

Так Топик температуры может называется *temp*, тогда Подписчик может, зная имя топики, подписаться на него и получать данные. А Издатель знает, куда "отдавать" свои данные.

Само значение температуры (данные) в модели PubSub называется **Сообщение (message)**.

Важно понимать, что в рамках одного топики должны находиться сообщения только одного типа, например, температуры. Если нам необходимы данные о давлении, то необходимо работать с другим топиком, например, pressure и т.д.

Через топики могут передаваться абсолютно разные данные, например, данные с лидара или камеры.

Любая нода может быть одновременно как подписчиком, так и издателем. Такой подход позволяет строить очень сложные взаимодействия, используя очень простой подход для обмена данными.

# Сообщения

Сообщение представляет собой структуру данных, которая используется при обмене информацией между нодами.

Топики (Topic), службы (Services) и Действия (Actions) используют сообщения для взаимодействия между собой. Сообщения могут включать в себя как базовые типы (целое число, число с плавающей точкой, логические и тд), так и массивы сообщений.

Помимо этого, сообщения могут инкапсулировать в себе другие существующие типы сообщений и специальные заголовки.

Сообщения описываются в файлах `.msg` как пары значений: тип поля и имя поля.

```
1 fieldtype fieldname
2 fieldtype1 fieldname1
```

Для реального примера сообщение может выглядеть так:

```
1 int32 x
2 int32 y
```

Мы описали сообщение, содержащее две переменных типа `int32` с именами `x` и `y`

---

## Базовые типы ROS

Типы данных ROS не заимствованы напрямую ни из одного из языков. В момент "сборки" происходит преобразование типа ROS к типу используемого языка.

В таблице ниже описаны базовые типы ROS и их представление в языках C++ и Python

в ROS	для C++	для Python
bool	uint8_t	bool
int8	int8_t	int
uint8	uint8_t	int
int16	int16_t	int
uint16	uint16_t	int
int32	int32_t	int
uint32	uint32_t	int
int64	int64_t	long
uint64	uint64_t	long
float32	float	float
float64	double	float
string	std::string	str
time	ros::Time	rospy.Time
duration	ros::Duration	rospy.Duration

Отдельно стоит отметить типы данных

в ROS	для C++	для Python
fixed-length	boost::array, std::vector	tuple
variable-length	std::vector	tuple
uint8[]	std::vector	bytes



---

`bool[]``std::vector<uint8_t>``list of bool`

---

## Наследование типов сообщений

Сообщения могут содержать не только базовые типы, но и переиспользовать уже созданные типы сообщений. Например, сообщение типа `geometry_msgs/Pose` описывается конфигурацией

```
1 Point position
2 Quaternion orientation
```

Где `Point` и `Quaternion` - это существующие типы сообщений.

Если рассмотреть сообщение `geometry_msgs/Pose` в развернутом виде, то мы увидим структуру:

```
1 geometry_msgs/Point position
2
3   float64 x
4   float64 y
5
6   float64 z
7
8 geometry_msgs/Quaternion orientation
9
10  float64 x
11
12  float64 y
13  float64 z
14  float64 w
```

---

## Консольная утилита `rosmmsg`

`rosmmsg` - это удобный инструмент командной строки, который предоставляют справочную информацию для разработчиков, а также служит мощным средством анализа для получения дополнительной информации о данных, передаваемых в ROS.

Например, если вы используете сообщение в своем коде, вы можете выполнить `rosmmsg show` в командной строке для получения списка полей, используемых в сообщении:

```
1 $ rosmmsg show sensor_msgs/CameraInfo
2
3 Header header
4   uint32 seq
5   time stamp
6   string frame_id
7 uint32 height
8 uint32 width
9 RegionOfInterest roi
10  uint32 x_offset
11  uint32 y_offset
12  uint32 height
13  uint32 width
14 float64[5] D
15 float64[9] K
16 float64[9] R
17 float64[12] P
```

## Команды консольной утилиты `rosmmsg`

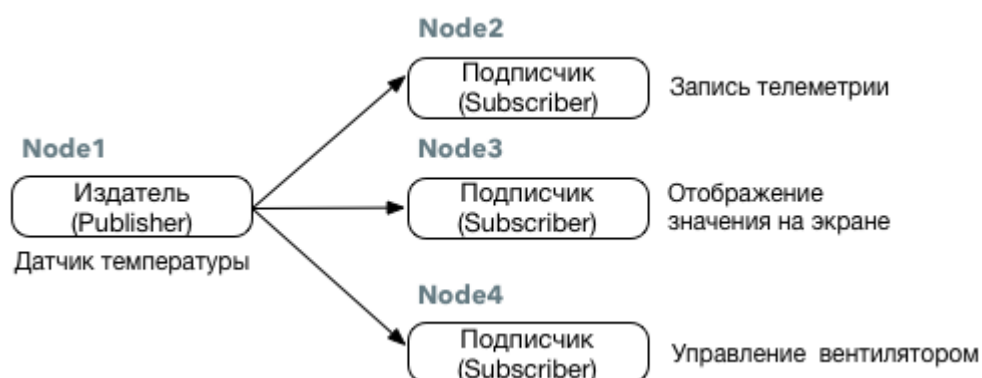
```
1 Commands:
2   rosmmsg show    Показать информацию о сообщении
3   rosmmsg info    Алиас для команды rosmmsg show
4   rosmmsg list    Вывести все существующие типы сообщений
5   rosmmsg md5     Отобразить md5sum сообщения
6   rosmmsg package Список всех сообщений в пакете
7   rosmmsg packages Список пакетов, использующих сообщение
```

# Topic

Модель работы в режиме Topic подразумевает использование одного типа сообщения для Издателя (Publisher) и Подписчика (Subscriber).

Модель Topic являются однонаправленной и подразумевает непрерывную отправку или получение сообщений. Такой способ коммуникации подходит для датчиков, которым требуются периодическая передача данных. Несколько подписчиков могут получать сообщения от одного издателя и наоборот (возможна работа несколько издателей).

На изображении ниже показана модель работы датчика температуры, когда его данные получают различные ноды.



## Консольная утилита rostopic

`rostopic` - это специальная консольная утилита, предназначенная для отображения отладочной информации о топиках в ROS. С ее помощью удобно искать нужные топики и выводить сообщения в консоль для отладки.

Список основных используемых команд:

- |   |                            |                                    |
|---|----------------------------|------------------------------------|
| 1 | <code>rostopic bw</code>   | Показать занимаемый сетевой канал  |
| 2 | <code>rostopic echo</code> | Вывести сообщения на экран         |
| 3 | <code>rostopic find</code> | Поиск топика по типу               |
| 4 | <code>rostopic hz</code>   | Показать частоту обновления топика |

5	<code>rostopic info</code>	Показать информацию о топике
6	<code>rostopic list</code>	Показать список существующий топиков
7	<code>rostopic pub</code>	Опубликовать данные в топик
8	<code>rostopic type</code>	Показать тип сообщения для топика

## Примеры использования

Вывести список существующих топиков:

```
rostopic list
```

Вывести сообщения из топика `topic_name` :

```
rostopic echo /topic_name
```

## rostopic pub

Отправить текстовое сообщение в топик:

```
rostopic pub my_topic std_msgs/String "hello there"
```

Отправить сообщение типа `geometry_msgs/Twist` в топик `/cmd_vel` с частотой 10hz:

```
rostopic pub -r 10 /cmd_vel geometry_msgs/Twist '{linear: {x: 0.1, y: 0.0,
```

Удобно при вызове функций использовать `Tab` для поиска и подстановки необходимых данных в командную строку.

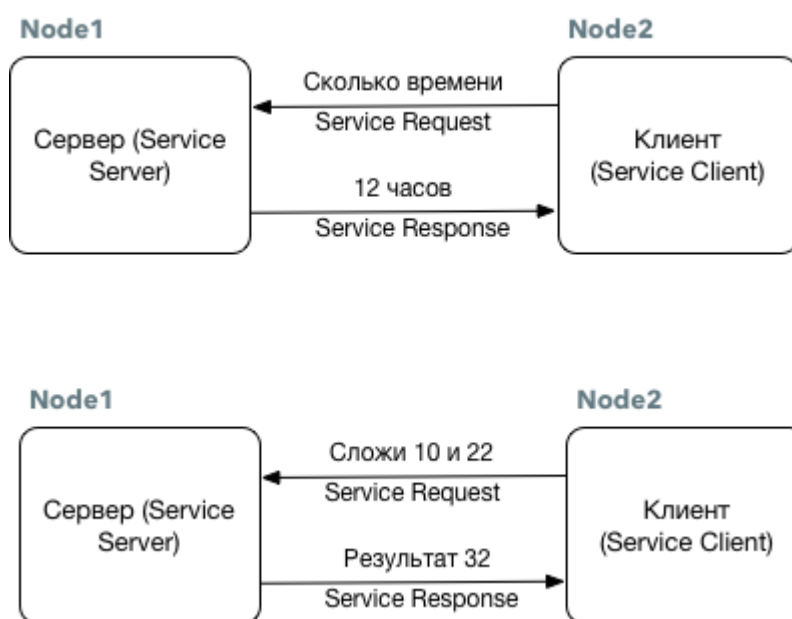
Например

```
1 rostopic pub /c+Tab -> rostopic pub /cmd_vel (подставится адрес существующей темы)
2 rostopic pub /cmd_vel +Tab -> rostopic pub /cmd_vel geometry_msgs/Twist (п
```

Более подробная информация доступна на Wiki странице <http://wiki.ros.org/rostopic>

# Service

Модель коммуникации в режиме Сервис представляет собой двунаправленную синхронную связь между клиентом (Service Client), создающим запрос, и сервером (Service Server), отвечающим на запрос. Синхронность в данном случае означает, что если наша программа отправит запрос, то до получения ответа от сервера программа находится в этой точке и ожидает результат.



Данный способ удобно использовать для периодической передачи данных или когда существует потребность в синхронной связи (режим запрос-ответ).

Сервер отвечает только тогда, когда есть запрос (Service Request) и клиент, который может получить ответ (Service Response). В отличие от модели работы с топиками, модель сервис работает с "одноразовыми" соединениями. Поэтому, когда цикл запрос и ответ завершен, соединение между двумя нодами будет прервано.

---

## Описание формата сервиса

Формат запроса и ответа задается специальным парным Сообщением (Message), в котором есть два сообщения: первое для запроса (Service Request), второе для ответа (Service Response). Файлы с описанием сервисов хранятся в директории `srv`

и имеют расширение `.srv` Подробное описание файла доступно на странице wiki <http://wiki.ros.org/rosbuild/srv>

В файле описания сервиса первая часть (до разделителя `—`) - это описание сообщения запроса, далее описание сообщения ответа.

Например `srv/AddToInts.srv`

```
1 uint32 x
2
3 uint32 y
4
5 ---
6
7 uint32 sum
```

При этом, имя файла `AddToInts.srv` соответствует имени Сервиса `AddToInts`.

---

## Консольная утилита `rosservice`

Для отладки и тестирования сервисов ROS существует специальная консольная утилита `rosservice` :

```
1 rosservice call Выполнение запроса к серверу
2 rosservice find Поиск сервиса по типу
3 rosservice info Выводит информацию о сервисе
4 rosservice list Выводит список запущенных сервисов
5 rosservice type Выводит тип сообщений используемый сервисом
6 rosservice uri Выводит RPC URL сервиса
```

Примеры использования `rosservice` :

### `rosservice call`

Вызов сервиса `service_name` с аргументами `service-args` :

```
rosservice call /service_name service-args
```

Вызов сервиса `my_service` с аргументами 1 и 2:

```
rosservice call /my_service 1 2
```

Для сложных сообщений аргументы с параметрами возможно писать в YAML синтаксисе, например:

```
rosservice call /my_service "{x: 1, y: 2}"
```

Подробнее об использовании YAML <http://wiki.ros.org/ROS/YAMLCommandLine>.

## **rosservice list**

```
rosservice list
```

Выводит список активных сервисов.

## **rosservice type**

```
rosservice type /service_name
```

Выводит тип обрабатываемого сообщения.

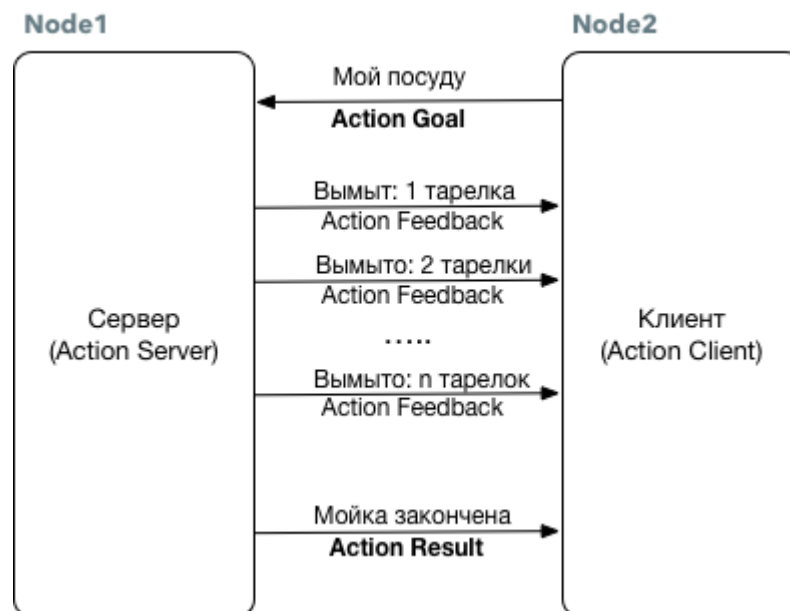


```
1 $ rosservice type my_service | rossrv show
2
3 result:
4 int64 x
5 int64 y
6 ---
7 int64 sum
```

Пример выводит описание типа сообщения для сервиса.

# Action

Модель коммуникации в режиме Действие (Action) используется, когда выполнение запрошенной команды занимает продолжительное время и необходима обратная связь с процессом. Это очень похоже на модель Service: (Service Request) используется как Задача (Action Goal), а ответ (Service Response) используется как Результат (Action Result). Также есть дополнительная сущность Обратная Связь (Action Feedback) для передачи промежуточных результатов выполнения процесса для клиента. После отправки запроса (Action Goal) программа может продолжать работать, "отвлекаясь" только на обработку или Результата (Action Result), или промежуточного состояния (Action Feedback). Такой метод работы будет асинхронным.



Например, как показано на рисунке, если клиент устанавливает цель – мыть посуду, сервер начинает мыть посуду и в процессе информирует клиента о ходе мытья посуды в форме обратной связи. После окончания обработки, сервер отправляет клиенту сообщение об окончании процесса.

В отличие от службы (Service), действие (Action) часто используется для управления сложными задачами робота, такими как передвижение к заданной точке, запуск лазерного сканирования, перемещение манипуляторов робота по сложной траектории и т.п.

## Файл описания действия (Action) .action

Файлы описания действия (Action) находятся в директории `./action` пакета, имеют расширение `.action`, и выглядят приблизительно так:

```
1  # Определение цели (goal)
2  uint32 dishes # Сколько мыть тарелок
3  ---
4  # Определение результата (result)
5  uint32 total_dishes_cleaned # Сколько всего было вымыто
6  ---
7  # Определение обратной связи (feedback)
8  uint32 dishes_cleaned # Сколько вымыто посуды сейчас
```

На основе этого файла `.action` создаются 6 вспомогательных сообщений, чтобы клиент и сервер могли общаться. Это создание сообщений автоматически запускается во время процесса сборки пакета.

Для файла `DoDishes.action` будут созданы файлы

```
1  DoDishesAction.msg
2  DoDishesActionGoal.msg
3  DoDishesActionResult.msg
4  DoDishesActionFeedback.msg
5  DoDishesGoal.msg
6  DoDishesResult.msg
7  DoDishesFeedback.msg
```

---

## Использование Action из консоли

Мы обсуждаем работу ActionLib как отдельного протокола, но на самом деле ActionLib именно с точки зрения "транспорта" использует механизм работы с Топиками (Topic).

Поэтому для отладки работы Actions мы будем использовать утилиту `rostopic`

Прежде чем начать, мы должны запустить некоторые программы, которые помогут нам разобраться с работой.

Запустим ноду симулятора `turtlesim`

```
roslaunch turtlesim turtlesim_node
```

Запустим тестовый ActionServer

```
roslaunch turtle_actionlib shape_server
```

Если вы получили ошибку запуска ноды, то возможно вы установили не полную версию ROS, выполните команды

```
sudo apt install ros-noetic-turtle-actionlib
```

Получим список работающих топиков `rostopic list`

```
1 ***
2 /turtle_shape/cancel
3 /turtle_shape/feedback
4 /turtle_shape/goal
5 /turtle_shape/result
6 /turtle_shape/status
```

Мы видим топики в одном пространстве имен `turtle_shape` : `/cancel` , `/feedback` , `/goal` , `/result` и `/status`

Эти топики реализуют работу нашего Action `shape_server`

При помощи довольно сложного вызова, мы можем отправить нашу Цель (Goal) на сервер для начала работы.

```
1 rostopic pub /turtle_shape/goal turtle_actionlib/ShapeActionGoal "header:  
2   seq: 0  
3   stamp:  
4     secs: 0  
5     nsecs: 0  
6   frame_id: ''  
7 goal_id:  
8   stamp:  
9     secs: 0  
10    nsecs: 0  
11   id: ''  
12 goal:  
13   edges: 6  
14   radius: 1.0"
```

Но нам не обязательно это все набирать, тут нас выручает подсказка через вызов

`Tab`

Мы можем набрать `rostopic pub /turtle_shape/goal` и нажать два раза таб для заполнения всей структуры сообщения.

Если мы заполним цель как `edges: 6, radius: 1.0`, то черепаха в эмуляторе начнет движение по шестиугольнику с гранями в 1 метр.



---

## Дополнительная информация

Модуль ActionLib имеет довольно сложную структуру взаимодействия и выполнения задач. Как Сервер, так и Клиент обрабатывают запросы по принципу Машины состояния (State Machine). Задачи могут отменяться, останавливаться и заново

запускаться. Подробное описание этого взаимодействия описано на Wiki странице <http://wiki.ros.org/actionlib/DetailedDescription>

Не так важно сейчас в этом полностью разобраться, как важно знать, что эти механизмы существуют, и обратиться к документации, когда это будет необходимо.

# Базовые понятия ROS

## Основные термины

### Мастер (Master), Мастер-Нода

Мастер выполняет роль сервера имен для возможности подключения между собой различных нод. Команда `roscore` запускает сервер мастера, и после этого к нему могут подключиться и зарегистрироваться ноды ROS. Связь между нодами (обмен сообщениями) невозможна без наличия запущенного мастера.

При запуске ROS `roscore` мастер будет запущен по адресу URI, установленному в переменной окружения `ROS_MASTER_URI`. По умолчанию адрес использует IP-адрес локального ПК и номер порта 11311

### Нода (Node)

Понятие ноды относится к наименьшей "рабочей" единице, используемой в ROS. Можно провести аналогию с одной исполняемой программой. ROS рекомендует создать одну ноду для каждой задачи, что позволит легче использовать ее в других проектах.

При запуске нода регистрирует информацию о себе на мастере (название ноды, типы обрабатываемых сообщений). Зарегистрированная нода может взаимодействовать с другими нодами (получать и отправлять запросы). Важно отметить, что обмен сообщениями между нодами работает без участия мастера (соединение между нодами происходит на прямую). Мастер обеспечивает только единое пространство имен для решения вопроса, куда подключиться к конкретной ноде. Адрес запуска ноды берётся из переменной окружения `ROS_HOSTNAME`, которая должна быть определена до запуска. Порт устанавливается на произвольное уникальное значение.

### Пакет (Package)

Пакет является основной единицей ROS. Любое приложение ROS оформляется в пакет, в котором определяются: конфигурация пакета, ноды, необходимые для работы пакета, зависимости от других пакетов ROS.

Работа с пакетами ROS очень похожа на работу с пакетами linux. Пакет ROS можно поставить готовым из репозитория пакетов или скачать и скомпилировать из исходных кодов.

Поиск доступных пакетов ROS возможен на странице <http://wiki.ros.org/>



# Стандарты ROS

## Единицы измерений

Данные используемые в ROS, должны соответствовать единицам СИ - стандарту, наиболее широко используемому в мире.

Это указано в рекомендации REP-01031 <http://www.ros.org/reps/rep-0103.html>.  
Например, используются длина в метрах, масса в килограммах, время в секундах, ток в амперах, угол в радианах, частота в герцах, сила в ньютонах, мощность в ваттах, напряжение в вольтах и температура в градусах Цельсия.

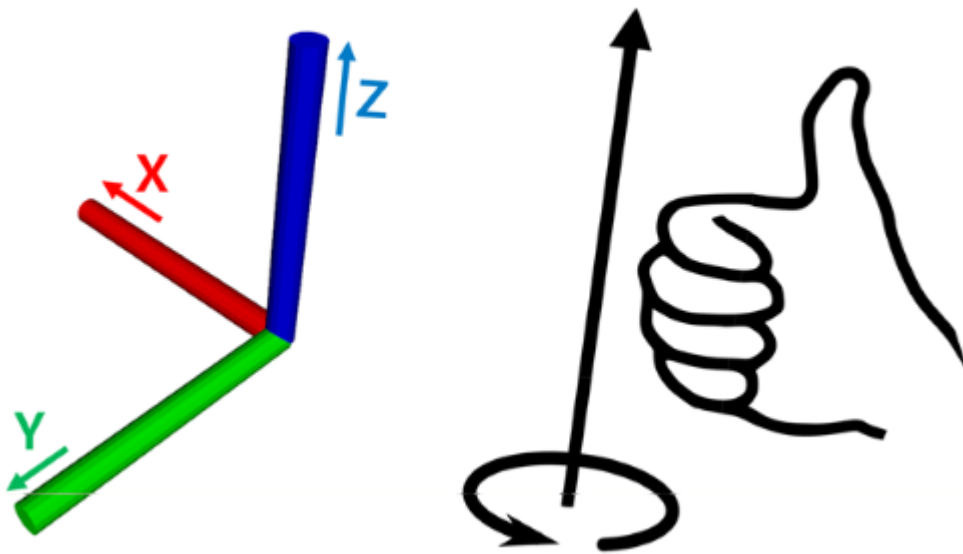
Все остальные блоки состоят из комбинации вышеупомянутых блоков. Например, скорость перемещения измеряется в метрах / сек, а скорость вращения измеряется в радиан / сек.

Рекомендуется соблюдать соответствие единицам СИ, поскольку это позволит другим пользователям переиспользовать ваш пакет без необходимости дополнительной конвертации.

---

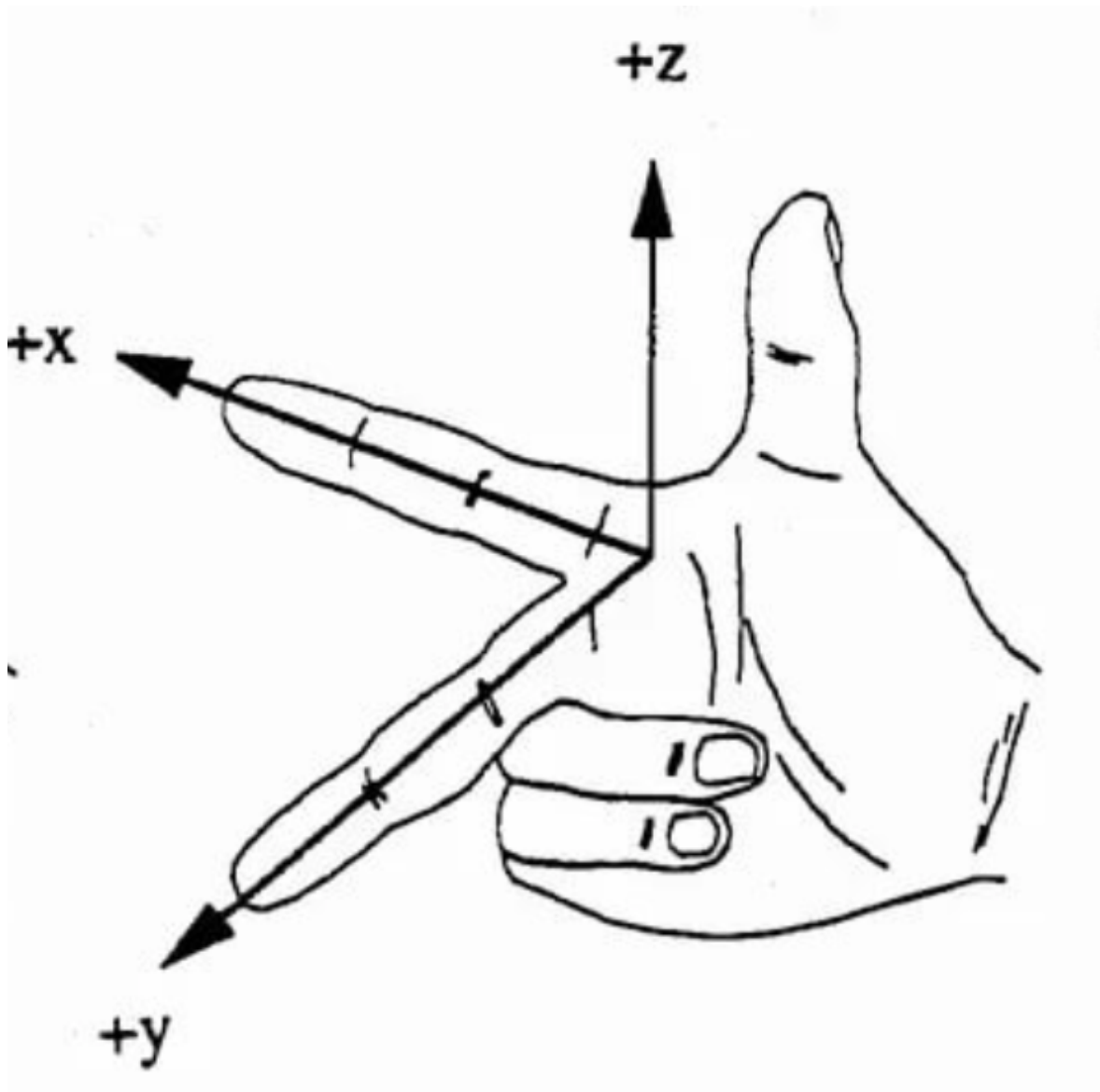
## Координаты XYZ

Оси  $x$ ,  $y$  и  $z$  в ROS используют правило правой руки (правило буравчика), как показано на рисунке.



Направление вперед - это положительное направление **оси X** , которое изображено красным. Направление налево представляет собой положительное направление **оси Y** , которое изображено зеленым. Наконец, направление вверх - это положительное направление **оси Z** , а ось обозначена синим. Цветовое кодирование легко запомнить по аббревиатуре RGB (стандарт представления цвета red-green-blue), которое соответствует осям XYZ.

Чтобы запомнить направления, вы можете расположить большой, указательный и средний палец в форме трех осей как показано на рисунке.



## Оси вращения

Следуя правилу буравчика, направление, которое ваша правая рука закручивает - это положительное направление вращения. Например, если робот вращается с 12 до 9 часов, то используя радианы для обозначения угла поворота, мы получаем, что робот вращается на  $+1.5708$  радиан по оси  $Z$ .

## Стандарты кодирования

ROS рекомендует разработчикам следовать единому стандарту оформления кода (именования и форматирования). Единый формат кодирования уменьшает количество дополнительной работы, которую разработчики часто должны делать при работе с исходным кодом. В частности, это улучшает понимание кода для других участников и облегчает проверку кода.

Правила по соблюдению стандарта не являются обязательными, но многие пользователи ROS соглашаются и придерживаются этих правил.

Перечислим принципы именования объектов

Объект	Правило	Пример
Пакет (Package)	under_scored	rst_ros_package
Topic, Service	under_scored	raw_image
File	under_scored	turtlebot3_fake.cpp
Variable	under_scored	string table_name;
Type	CamelCased	typedef int32_t PropertiesNumber;
Class	CamelCased	class UriTable
Structure	CamelCased	struct UriTableProperties
Function	camelCased	addTableEntry();
Method	camelCased	void setNumEntries(int32_t num_entries)
Constant	ALL_CAPITALS	const uint8_t DAYS_IN_A_WEEK = 7;
Macro	ALL_CAPITALS	#define PI_ROUNDED 3.0

# Практическое задание

## Установка ROS

Необходимо установить ROS, запустить программы turtlesim и turtlesim\_teleop\_key.  
Продemonстрировать управление "черепахой"

---

## ROS Topic

При помощи консольной утилиты rostopic

- Отправить в топик `/temp` сообщение типа `int` с частотой **5** герц
  - Вывести информацию топика `/temp` в консоль, удостовериться, что данные передаются
  - Удостовериться, что частота публикации данных с заданной частотой
- 

## ROS Service

Запустите команду для запуска тестового сервиса (сервис получает два числа и возвращает сумму)

```
roslaunch rospy_tutorials add_two_ints_server
```

При помощи утилиты `rosservice` :

- Найдите имя созданного сервиса
  - Получите структуру используемого сервисом сообщения
  - Выполните сложение двух цифр 5 и 10, используя сервис
-

## ROS Actions

Повторите самостоятельно примеры работы с Actions из консоли (движение черепахи по многоугольнику). Выполните движение по квадрату с длинной стороны 3 метра.