ROS 2: Callbacks Handling, Control & Behavior Modeling

Course: Robotic Programming Environments

Michał Drwięga
michal.drwiega@pwr.edu.pl
www.mdrwiega.com/edu/rpe

Department of Cybernetics and Robotics
Wrocław Univeristy of Science and Technology

Wrocław University
of Science and Technology

For educational use only
©2026 Michał Drwięga

Content

1. ROS 2: Callback Groups
2. ROS 2: Managed Nodes
3. ros2_control
4. Behavior modeling: FSM, BT

For educational use only
©2026 Michał Drwięga

ROS 2: Callback Groups

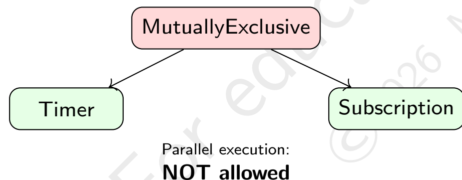
For educational use only
©2026 Michał Dziega

Callback Groups

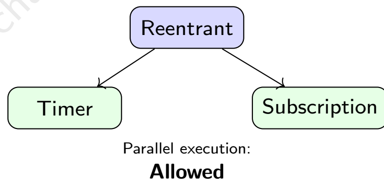
Definition

Callback groups organize callbacks (subscriptions, timers, services) to control their parallel execution within a node.

Mutually Exclusive Group
Only one callback at a time



Reentrant Group
Multiple callbacks in parallel



This mechanism is relevant when using a `MultiThreadedExecutor`.

For educational use only © 2026 Michał Drwięga

Problem Overview

In ROS 2:

- Every node has a **default callback group**.
- Default type: `MutuallyExclusive`.
- Even with `MultiThreadedExecutor` callbacks in the same group do not run concurrently.

For educational use only
©2026 Michał Drwięga

Example: Default Callback Group

```python
class MyNode(Node):
    def __init__(self):
        super().__init__("my_node")

        self.sub1 = self.create_subscription(
            String, "topic1", self.cb1, 10)

        self.sub2 = self.create_subscription(
            String, "topic2", self.cb2, 10)

    def cb1(self, msg):
        self.get_logger().info("Callback 1")

    def cb2(self, msg):
        self.get_logger().info("Callback 2")
```

Result

Both subscriptions use the default `MutuallyExclusive` callback group.

MultiThreadedExecutor

```
executor = MultiThreadedExecutor()

node = MyNode()
executor.add_node(node)
executor.spin()
```

Even with multiple executor threads:

- `cb1()` and `cb2()` are serialized
- They cannot execute at the same time

Callback Group rules > Executor threads

For educational use only ©2020 Michał D

Execution Visualization

Default Callback Group

Time	Thread 1	Thread 2	Result
t1	cb1()	idle	serialized
t2	cb1()	idle	serialized
t3	cb2()	idle	serialized

Second thread is unused because the callback group prevents parallel execution.

educational use only 2026 Michał Drwięga

Creating Reentrant Callback Groups

```python
from rclpy.subscription import SubscriptionOptions
from rclpy.callback_groups import ReentrantCallbackGroup

group1 = ReentrantCallbackGroup()

options = SubscriptionOptions()
options.callback_group = group1

self.sub1 = self.create_subscription(
    String, "topic1", self.cb1, 10, callback_group=group1)
```

- Reentrant groups allow concurrent execution
- Now the same callback can be executed simultaneously in multiple instances.

For educational use only ©2026

Parallel Execution Example

Reentrant Callback Groups

Time	Thread 1	Thread 2	Result
t1	cb1()	cb1()	parallel
t2	cb1()	cb1()	parallel

For educational use only
©2026 Michał Drwięga

Creating Mixed Callback Groups

```
from rclpy.callback_groups import ReentrantCallbackGroup,
    MutuallyExclusiveCallbackGroup

# cb1: must NOT run in parallel with itself
cb1_group = MutuallyExclusiveCallbackGroup()

# cb2: can run in parallel
cb2_group = ReentrantCallbackGroup()

self.sub1 = self.create_subscription(
    String, "topic1", self.cb1, 10, callback_group=cb1_group
)

self.sub2 = self.create_subscription(
    String, "topic2", self.cb2, 10, callback_group=cb2_group
)
```

- cb1 is serialized (safe for shared state)
- cb2 runs concurrently (independent work)

Mixed Execution Example (MultiThreadedExecutor)

Time	Thread 1	Thread 2	Result
t1	cb1()	cb2()	parallel
t2	cb2()	cb2()	parallel
t3	cb1()	cb2()	parallel

cb1 never overlaps with itself, but can run alongside cb2.

For educational use only ©2026 Michał Drwięga

Fully Controlled Parallel Execution

Goal

We want to control callback execution such that:

- `cb1` must NOT run in parallel with itself
- `cb2` must NOT run in parallel with itself
- `cb1` and `cb2` can run in parallel

For educational use only ©2026 Michał D...

Solution: Callback Group Separation

Use **two separate MutuallyExclusive callback groups** to enforce per-callback serialization while allowing cross-callback parallelism.

- `cb1` → MutuallyExclusive group A (no self-parallelism)
- `cb2` → MutuallyExclusive group B (no self-parallelism)
- Different groups → can execute in parallel (MultiThreadedExecutor)

For educational use only ©2026 Michał D...

Fully Controlled Parallel Execution

```
from rclpy.callback_groups import
    MutuallyExclusiveCallbackGroup

# cb1 must not overlap with cb1
cb1_group = MutuallyExclusiveCallbackGroup()

# cb2 must not overlap with cb2
cb2_group = MutuallyExclusiveCallbackGroup()

self.sub1 = self.create_subscription(
    String, "topic1", self.cb1, 10,
    callback_group=cb1_group
)

self.sub2 = self.create_subscription(
    String, "topic2", self.cb2, 10,
    callback_group=cb2_group
)
```

Callback Groups: Summary

- ROS 2 nodes use a default `MutuallyExclusive` callback group
- MultiThreadedExecutor alone does not provide concurrency
- Callback groups determine whether callbacks may execute in parallel
- Use:
    - `Reentrant`
    - or separate callback groups

  for true multithreaded execution

Executor threads + proper callback groups = parallel execution

For educational use only
©2026 Michał Drwięga

ROS 2: Managed Nodes

For educational use only
©2026 Michał Dróżdża

ROS 2 Lifecycle Nodes

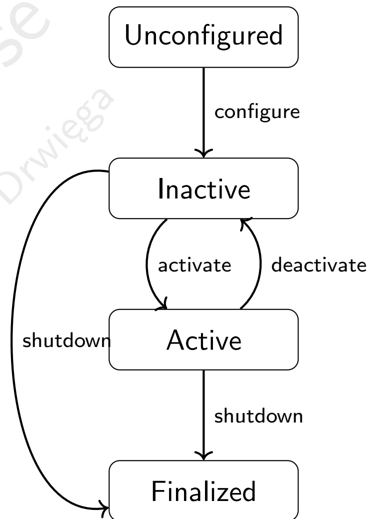
Lifecycle nodes provide a managed execution model that enables controlled startup, shutdown, and recovery of system components.

Main states:

- **Unconfigured** – node not initialized
- **Inactive** – configured, no processing
- **Active** – normal operation
- **Finalized** – node shutdown state

Advantages:

- Controlled system startup
- Improved fault handling
- Deterministic initialization sequence



For educational use only ©2026 Michał Drwięga

ROS 2 Lifecycle Node – Python Example

```python
import rclpy
from rclpy.lifecycle import LifecycleNode,
    TransitionCallbackReturn


class CameraNode(LifecycleNode):
    def __init__(self):
        super().__init__('camera_node')

    def on_configure(self, state):
        self.get_logger().info('Configuring node')
        return TransitionCallbackReturn.SUCCESS

    def on_activate(self, state):
        self.get_logger().info('Activating node')
        return TransitionCallbackReturn.SUCCESS

    ...

```

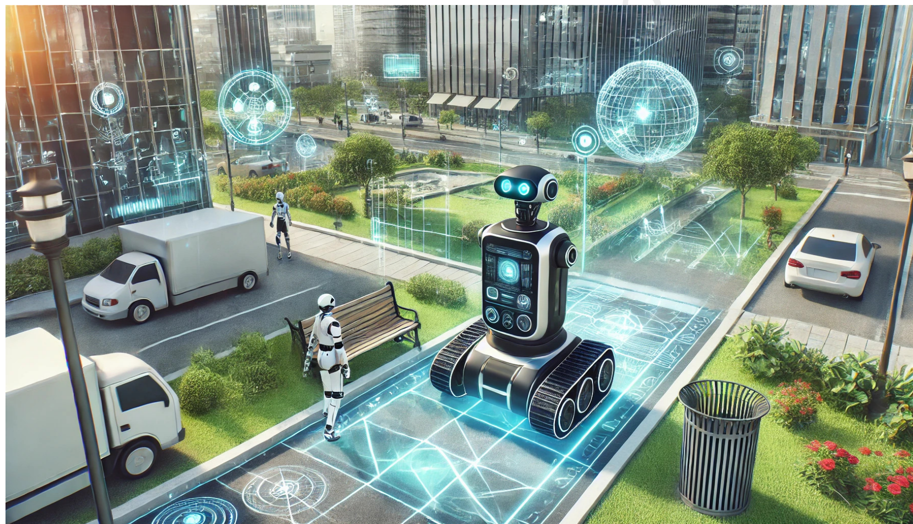
ROS 2 Lifecycle Commands

```
# List available lifecycle nodes
ros2 lifecycle nodes

# Show current node state
ros2 lifecycle get /camera_node

# Configure node
ros2 lifecycle set /camera_node configure

# Activate node
ros2 lifecycle set /camera_node activate

# Deactivate node
ros2 lifecycle set /camera_node deactivate

# Shutdown node
ros2 lifecycle set /camera_node shutdown
```

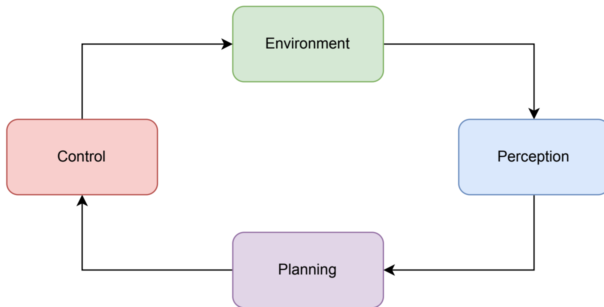
ros2_control

For educational use only
©2026 Michał Drwięga

What is Autonomous Robot?



Autonomous Robot: Perception, Planning, and Control



Introduction to ROS 2 Control (ros2_control)

- ROS 2 Control is a framework for real-time control of robotic hardware.
- Provides interfaces for controlling joints, actuators, and sensors in ROS 2.
- Built to address limitations of the ROS 1 Control framework with a more modular and efficient design.
- Ideal for robotics applications requiring precise control over hardware in real-time.

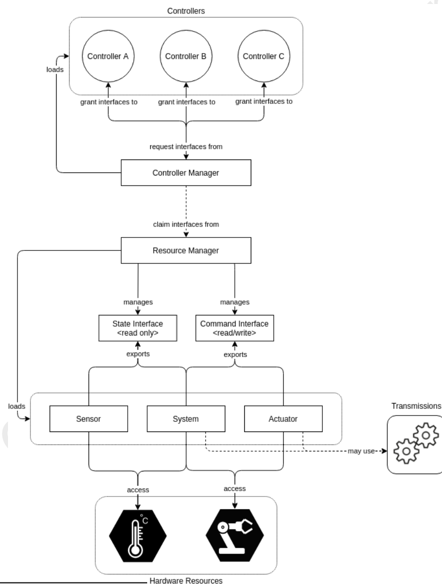
For educational use only
©2026 Michał Dwiega

ROS 2 Control: Key Components

- **Controller Manager** - Manages and loads controllers dynamically.
- **Resource Manager** - Abstracts physical hardware and its drivers (called hardware components) for the ros2_control framework.
- **Controllers** - Specific modules for joint position, velocity, and effort control.
- **Hardware Interface** - Defines communication with robot hardware (motors, sensors).

For educational use only ©2026 Michał Drwięga

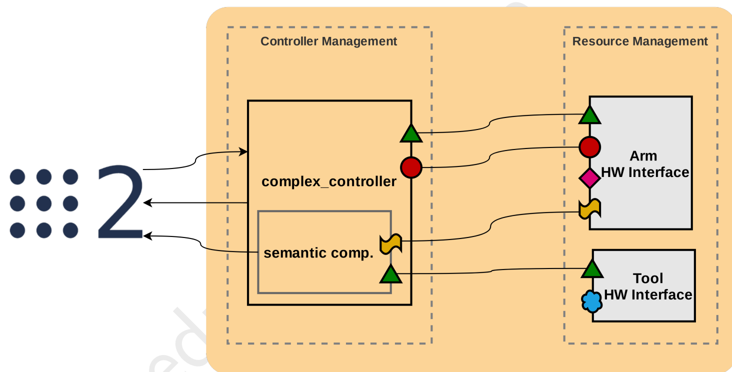
ROS 2 Control: Architecture



1

[1] https://control.ros.org/rolling/doc/getting_started/getting_started.html

ROS 2 Control: Architecture



CC-BY: Denis Stogl, Bence Magyar (ros2_control) [2]

[2] https://control.ros.org/rolling/doc/getting_started/getting_started.html

ROS 2 Control: Controller Manager

- **Lifecycle Management**: Load, configure, start, and stop controllers dynamically.
- **Real-time Safe Execution**: Ensures controllers update in a deterministic, efficient loop.
- Improves flexibility by managing controller states without disrupting other parts of the system.

For educational use only
©2026 Michał Drwięga

ROS 2 Control: Types of Controllers

- **Effort Controllers**: Control force/torque output for joints.
- **Velocity Controllers**: Control the speed of joints or wheels.
- **Position Controllers**: Set the position of joints.
- **Gripper Controllers**: Control the gripper (e.g. vacuum gripper).
- **Joint Trajectory Controllers**: Allows control over multiple joints for smooth trajectory execution.
- **PID Controller**
- **Custom Controllers**: Users can implement custom controllers for specific hardware needs.

For educational use only © 2026 Michał Drwięga

ROS 2 Control: Broadcasters

Broadcasters are used to publish sensor data from hardware components to ROS topics.

Types of Broadcasters:

- **Force Torque Sensor Broadcaster**: Publishes data from force and torque sensors.
- **IMU Sensor Broadcaster**: Transmits data from IMU, providing information on orientation, angular velocity, and acceleration.
- **Joint State Broadcaster**: Publishes the states of each joint (position, velocity, effort).
- **Range Sensor Broadcaster**: Sends data from range sensors like LIDAR or ultrasonic sensors.
- **Pose Broadcaster**: Provides the pose of a component.

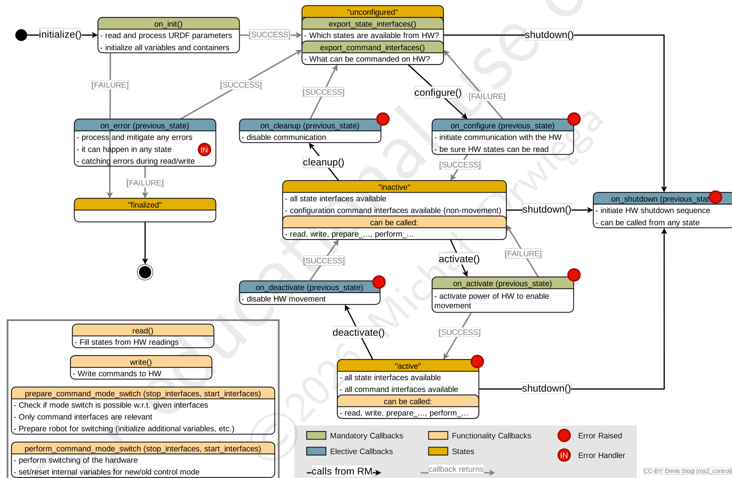
For educational use only
©2026 Michał Drwięga

ROS 2 Control: Hardware Interface

- **Hardware Abstraction Layer**: Interfaces between robot hardware and ROS 2.
- **Components**: Includes *actuators* (motors, etc.) and *sensors* (encoders, etc.).
- Standardized data flow for easy integration of various hardware types.

For educational use only
©2026 Michał Drwięga

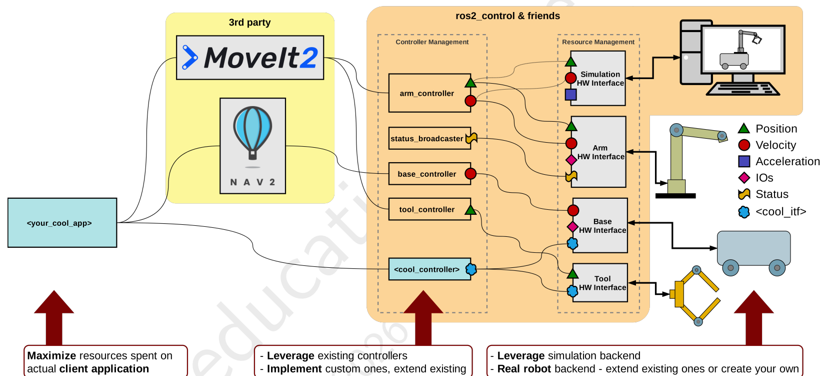
Hardware Interface Life Cycle



[3]

[3] https://control.ros.org/rolling/doc/getting_started/getting_started.html

ROS 2 Control: Example



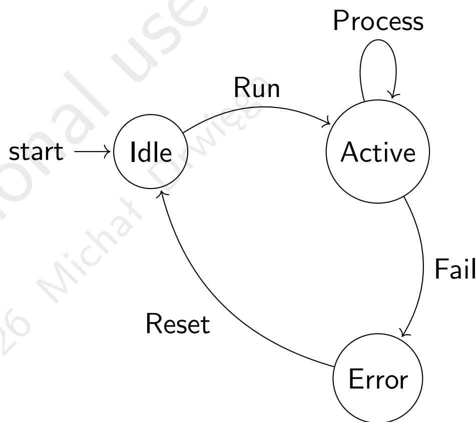
[4] https://control.ros.org/rolling/doc/resources/resources.html

Behavior modeling: FSM, BT

For educational use only
©2026 Michał Dr. Wega

Finite State Machine (FSM)

- A Finite State Machine consists of:
  - A set of states.
  - A starting state.
  - Input events causing transitions between states.



For educational use only ©2026 Michał Dwięg

FSM - example in Python

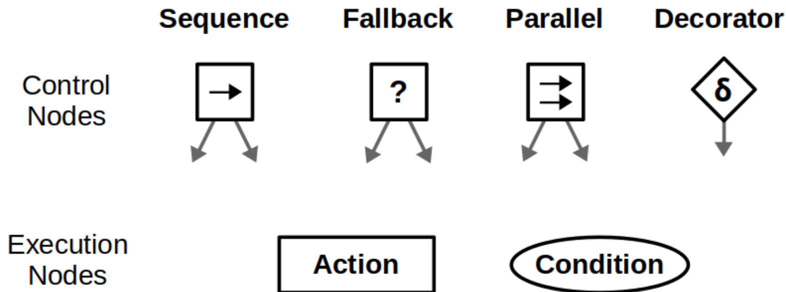
```python
state = "IDLE"

def handle_event(event):
    global state

    if state == "IDLE":
        if event == "start":
            state = "RUNNING"
    elif state == "RUNNING":
        if event == "stop":
            state = "IDLE"

if __name__ == "__main__":
    while True:
        event = input("Enter event (start/stop): ")
        handle_event(event)
```

Behavioral Trees in ROS 2

- **What are Behavioral Trees?**
  - A hierarchical model for robot behavior management.
  - Combines logic and control for task execution.
  - Nodes represent actions, conditions, or control flow elements.
- **Advantages over Finite State Machines (FSMs)**
  - Modularity: Individual behaviors are isolated and reusable.
  - Flexibility: Easy to add or modify behaviors.
  - Robustness: Handles failure and retries naturally.
- **Behavioral Trees in ROS**
  - Popular libraries: `BehaviorTree.CPP`, `py_trees`.
  - Integration with ROS actions and services.
  - Useful for autonomous navigation, manipulation, and task planning.

For educational use only ©2026 Michał Drwięga

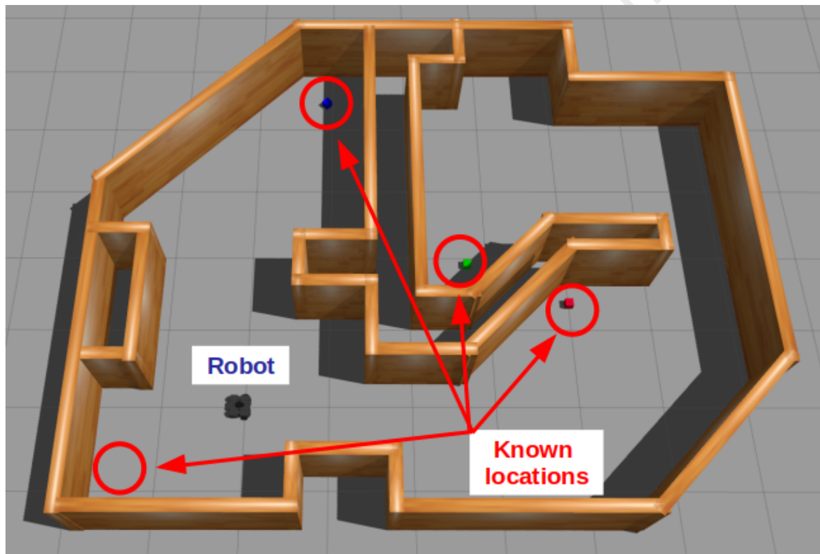
Behavioral Trees: Types of Nodes



5

[5] https://robohub.org/introduction-to-behavior-trees/

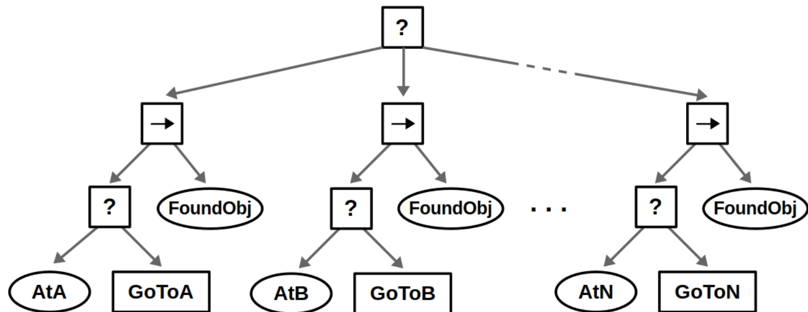
Behavioral Trees: Example



[6]

[6] https://robohub.org/introduction-to-behavior-trees/

Behavioral Trees: Example



[7]

[7] https://robohub.org/introduction-to-behavior-trees/

Summary

1. ROS 2: Callback Groups
2. ROS 2: Managed Nodes
3. ros2_control
4. Behavior modeling: FSM, BT

For educational use only
©2026 Michał Drwięga

The end

Thank you for your attention!

For educational use only
©2026 Michał Drwięga