# Week_6

### 챕터 21 ~ 24

## 21장 ROS2 시간

### 21.1 시계(Clock)과 시간(Time)

- 로봇이 다수의 센서를 사용할 때에,

  각 센서의 **변화량을 감지**하고 센서들 간의 **시간을 동기화**하는 것은 매우 중요하다.

- 1초 간격으로 데이터를 퍼블리시하는 노드와
- 2초 간격으로 데이터를 퍼블리시하는 노드가 있다고 가정할때

  이 2개의 토픽을 Subscribe해서 로봇의 위치추정에 활용하기 위해서는 **가까운 시간에 퍼블리시**된 데이터를 인풋으로 넣어주어야 정확한 결과를 얻을 수 있다.

  →그렇기 때문에 ROS2에서는 데이터뿐만 아니라 토픽의 Publish 시점도 함께 포함할 수 있고

  Publish 시점을 포함하는 **std_msgs/msg/header** 타입의 데이터는 sensor_msgs, geometry_msgs, nav_msgs등의 **기본**(표준) **메시지 타입**에 기본으로 포함되어 있다.

```
kody@desktop:~$ ros2 interface show std_msgs/msg/Header
# Standard metadata for higher-level stamped data types.
# This is generally used to communicate timestamped data
# in a particular coordinate frame.

# Two-integer timestamp that is expressed as seconds and nanoseconds.
builtin_interfaces/Time stamp
        int32 sec
        uint32 nanosec

# Transform frame with which this data is associated.
string frame_id
```

- builtin_interfaces/Time stamp
  int32 sec
  uint32 nanosec
  - Time stamp는
    - int32형식의 sec와
    - uint32형식의 nanosec으로 publish됨을 알 수 있다.

- string frame_id
  - frame_id를 정의한다.
  - frame_id는 해당 메시지가 어떤 프레임(frame)에 속해있는지를 의미한다.

## 21.2 시간 추상화

- ROS2에서는 현재 시간을 표현할 수 있을뿐만 아니라
  과거 데이터를 다루기 위해**(ros2bag)** 과거 시점에서 시간을 재생하거나 / 시간을 배속하거나 / 멈출 수 있는 기능을 갖고있다.

- 이를 위해 ROS2에서는 시간을 추상화하였고

이러한 시간은 **시간 소스(Time Source)를 선언**하여 사용할 수 있다.

- rcl/time.h

```
enum rcl_time_source_type_t
{
  RCL_TIME_SOURCE_UNINITIALIZED = 0,
  RCL_ROS_TIME,
  RCL_SYSTEM_TIME,
  RCL_STEADY_TIME
};
```

- RCL_TIME_SOURCE_UNINITIALIZED

  부분은 다른 시간 소스 유형

  (RCL_ROS_TIME,
  RCL_SYSTEM_TIME,
  RCL_STEADY_TIME)

  중에 하나가 할당되기 전의 시간 소스 초기 상태를 설정하는 부분이다.

- RCL_ROS_TIME

  은 ROS clock에 따른 시간으로 시뮬레이션 환경에 따라

  going back in time, slowing down, speeding up 등의 작업을 할 수 있다.

- RCL_SYSTEM_TIME

  컴퓨터 내부 clock에 따른 시간으로, 현재 시간을 나타내고 싶을 때에 사용한다.

- RCL_STEADY_TIME

  ROS 시스템의 지속 시간(duration)과 간격(interval)을 측정하기 위해 사용되는 시간으로,

  설정할 수 없고, 시스템 내부 clock을 조정해도 정지되지 않는다.

## 21.3 Time API

time_rclcpp_example/src/main.cpp 예제

```
#inlcude <memory>
#include <utility>

#include "rclcpp/rclcpp.hpp"
#include "rclcpp/time_source.hpp"

#include "std_msgs/msg/header.hpp"

int main(int argc, char* argv[])
{
  rclcpp::init(argc,argv);

  auto node = rclcpp::Node::make_shared("time_example_node");
  auto time_publisher= node->create_publisher<std_msgs::msg::Header>("time",10);
  std_msgs::msg::Header msg;

  rclcpp::WallRate loop_rate(1.0);
  rclcpp::Duration duration(1, 0);

  while (rclcpp::ok())
  {
    static rclcpp::Time past = node->now();

    rclcpp::Time now = node->now();
    RCLCPP_INFO(node->get_logger(), "dec %lf nsec%ld", now.second(), now.nanosecond());

    if ((now - past).nanosecond() * 1e-9 > 5)
    {
      RCLCPP_INFO(node->get_logger(), "Over 5 seconds!");
      past = node->now();
      msg.stamp = now +duration;
      time_publisher->publish(msg);

      rclcpp::spin_some(node);
      loop_rate.sleep();
    }

    rclcpp::shutdown();
    return 0;
  }
```

## 21.3.1 Time

- Time클래스는 오퍼레이터를 제공하여 결과를 seconds 또는 nanoseconds단위로 반환한다.

```
if ((now - past).nanosecond() * 1e-9 > 5)
    {
      RCLCPP_INFO(node->get_logger(), "Over 5 seconds!");
      past = node->now();
    }
```

- now 멤버 함수를 이용하면 노드 시간을 확인할 수 있다.

```
rclcpp::Time now = node->now();
```

## 21.3.2 Duration

- Duration클래스는 **기간(duration)을 다룰 수 있는** 오퍼레이터를 제공하여,
  결과를 seconds나 nanoseconds단위로 반환한다.
  Duration이 이전 시간을 표현하는 경우, 이는 **음수로 표기**된다.

다음 예제 코드는 실제 시간보다 1초 느린 값을 계산한다.

```
rclcpp::Duration duration(1, 0);
msg.stamp = now +duration;
time_publisher->publish(msg);
```

## 21.3.3 Rate

- Rate클래스는 반복문에서 특정 주기를 유지시켜 주는 API를 제공한다.

다음 예시는 WallRate클래스로 Hz단위를 사용하는 생성자를 생성한 후,
sleep()함수를 사용하여 주기를 맞추는 것을 확인할 수 있다.

```
rclcpp::WallRate loop_rate(1);

while (rclcpp::ok())
```

```
{
  (생략)
  loop_rate.sleep();
}
```

# 22장 ROS2의 파일 시스템

## 22.1 파일 시스템

- ROS2의 파일 시스템(Filesystem)을 활용하면
  - ROS 패키지 / 소스 코드 검색
  - 메시지 / 실행 파일 / 파라미터 파일 검색 및 설정
  - 환경설정 파일 검색 및 설정

    등의 기능을 활용할 수 있다.

## 22.2 패키지와 메타패키지

## 22.3 바이너리 설치와 소스코드 설치

- ROS2 패키지 설치는 **빌드없이** 바로 실행할 수 있는 설치 방법이고
- 소스코드를 내려받은 후에 사용자가 **빌드하여** 실행하는 방법이 있는데

- 소스코드나 패키지를 **변경**해 사용하거나 소스코드의 내용을 확인하려면 소스코드 형태로 설치하면 된다.

### 22.3.1 바이너리 설치

- 다음 명령어를 사용하여 해당 패키지를 설치할 수 있고 파일은 **/opt/ros/humble**에 저장된다.

```
$ sudo apt install ros-humble-teleop-twist-joy
```

- ros2 run이나 ros2 launch를 이용하면 **해당 패키지 내**의 **노드를 실행**시킬 수 있다.

### 22.3.2 소스코드 설치

- 소스코드 설치는 **사용자의 작업 폴더**(ex, ~/robot_ws/src)에 git clone 명령어를 이용해 원격 리포지터리를 복사한 후에 **빌드**하는 방법이다.

```
$ cd ~/robot_ws/src
$ git clone https://githum.com/ros/teleop_twist_joy.git
$ cd ~/robot_ws
$ colcon build --symlink-install --package-select teleop_twist_joy
```

- 빌드한 파일은 해당 패키지의 **install/<package>** 디렉토리에 포함된다.

## 22.4 기본 설치 폴더와 사용자 작업 폴더

- ROS의 데스크톱 버전을 설치하면, /opt 폴더에 ros폴더가 생성되고

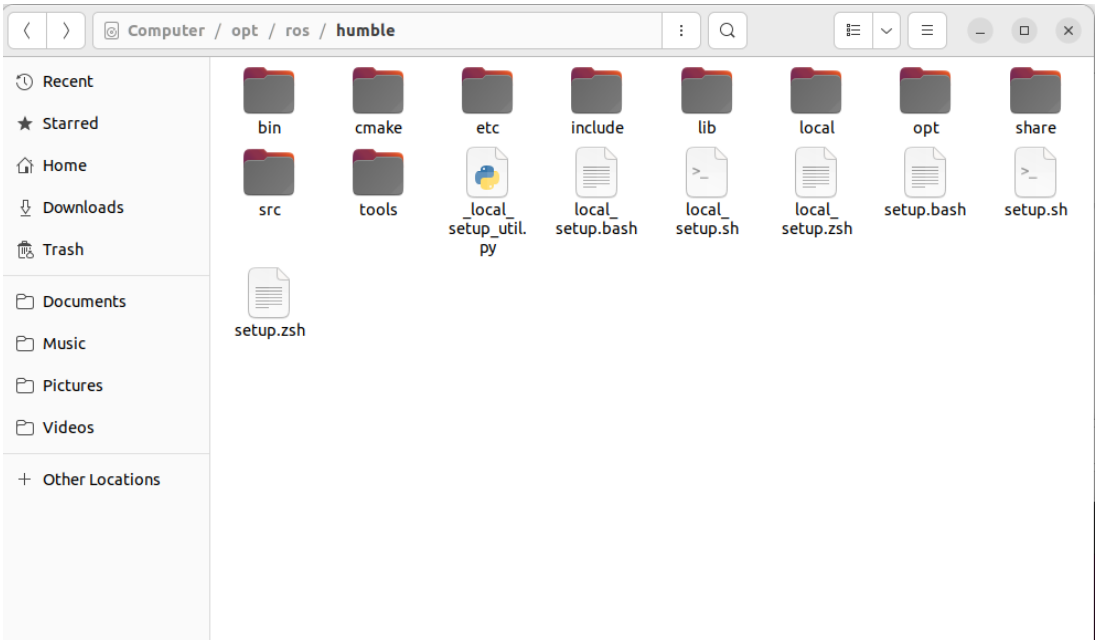
  각 버전별로 또 폴더가 생성된다.

  이 폴더에는 RQt, RViz, 관련 라이브러리, 시뮬레이션, 내비게이션 등이 포함된다.

### 22.4.1 기본 설치 폴더

- 경로: /opt/ros/<버전 이름(distro)>

- ros 폴더의 세부 내용
  - /bin : 실행 가능한 바이너리 파일
  - /cmake : 빌드 설정 파일
  - /include : 헤더 파일
  - /lib : 라이브러리 파일
  - /opt : 기타 의존 패키지
  - /share : 패키지의 빌드, 환경설정 파일
  - local_setup.* / setup.* : 환경설정 파일




## 22.4.2 사용자 작업 폴더

- 사용자 작업 폴더는 사용자가 원하는 위치에 위치시킬 수 있다.

- 세부 내용

- /build : 빌드 설정 파일용 폴더
- /install :  msg,srv,action헤더 파일과 사용자 패키지 라이브러리, 실행용 폴더
- /log : 로깅 파일용 폴더
- /src : 사용자 패키지용 폴더

- /src 내에는 사용자 패키지가 위치하는데 이는 다양한 파일로 구성되어 있다
  - /src /include /param /launch /test 등등

# 23장 빌드 시스템과 빌트 툴

## 23.1 빌드 시스템과 빌트 툴

- 빌드 시스템은 **단일 패키지를 대상**으로 하고
- 빌트 툴은 **시스템을 대상**으로 한다.

- ROS2는 코드의 재사용을 위해 패키지와 노드 단위로 구성되어있고,
  각 패키지는 코드의 재사용을 위해 다른 패키지와 상호 호환성을 갖게 된다.

- 이러한 상호 호환성은 기본적인 기능만 갖춘 패키지라면 Low level단에서
  RCL만 갖기도 하고, 복잡한 응용 패키지라면 수십 개의 의존성을 갖기도 한다.

- C++에서는 **단일 패키지**를 대상으로 **빌드 시스템**인 **catkin과 ament_cmake**를 사용하고,
- Python은 **Python setuptool**을 사용하고 있다.
  - ROS에서는 **수많은 패키지를 함께 빌드**하여 사용하는 구조이기 때문에 **각 패키지별로 서로 다른 빌드 시스템을 호출**하여 (복잡한 경우) 의존성 관계를 해석하여 **토폴로지 순서대로 빌드 해야만 한다.**

- 이러한 의존성 관계를 해석하여 **각 패키지에 대한 빌드 시스템(단일 패키지)을 호출**하는 것이 **ROS2의 빌트 툴(시스템을 대상으로 한다.)**이라 할 수 있다.

## 23.2 빌드 시스템(단일 패키지)

- ROS1의 빌드 **시스템**은
  - CMake를 사용하고
  - 빌드 환경은 CMakeList.txt파일에 작성한다.

- ROS에서는 CMake가
  - 리눅스, BSD, macOS, 윈도우 계열도 지원하고
  - 빌드 환경을 기술하는 CMakeLists.txt는 Visual Studio, Eclipse, Qt Creater등의 IDE에서 기본으로 지원하기 때문에 사용하고 있다.

- catkin **빌드 시스템**(단일 패키지)은 catkin_make **빌드 툴**(시스템)과 같이 사용할 수 있다.

---

- ROS2에서는 **빌드 시스템**(단일 패키지)으로 **ament**를 사용한다.
- ament는 **catkin**(ROS1)의 업그레이드 버전으로,

  **CMakeList.txt에 기술된 사항을 기반**으로, 빌드를 수행한다.

- ament는
  1. devel 공간을 사용하지 않고
  2. CMAKE_PREFIX_PATH가 아닌 AMENT_PREFIX_PATH와 같은 고유 환경설정을 사용할 수 있다.

## 23.3 빌드 툴(Build Tool)

빌드 툴은 전체 시스템을 대상으로 하고,

- ROS1에서는 **catkin_make**를 **빌드 툴**로 사용한다.
- ROS2에서는 **colcon build**를 기본 **빌드 툴**로 사용하고 있다.

  CLI형태로 사용할 수 있다.

```
$ colcon build
```

## 23.4 패키지 생성

- ROS2 패키지를 생성하는 방법은 두 가지가 있다.
  1. 직접 패키지 폴더와 디렉토리를 만들고

     package.xml / CMakeLists.txt / setup.py등의 파일을 만드는 것
  2. ros2cli 명령어를 이용하는 것

- ros2 pkg create 명령어는 다음과 같다.

```
$ ros2 pkg create [패키지이름] --build-type [빌드 타입] --dependendies [의존 패키지1] [의존 패키지~~]
```