

# ROS Basic Programming

---

WeGo & WeCAR

# 목 차

1. ROS 통신 소개
2. ROS Topic
3. ROS Service
4. ROS Action
5. ROS Message

01

---

# ROS 통신 소개

## 01 ROS 통신 소개

- ROS 내에서는 노드 사이의 데이터 통신을 메시지를 이용하며, 통신 방식으로는 Topic, Service, Action이 있습니다.
- ROS는 작업 공간을 catkin을 이용하여 관리, 구성, 빌드하게 됩니다.
- 이전에 생성한 ~/catkin\_ws/src 를 기준으로 진행하겠습니다.

```
%% git clone https://github.com/JacksonK9/basics
```

02

---

**ROS Topic**

## 02 ROS Topic

- `catkin_create_pkg` 명령어로 `rospy`를 의존성으로 가지는 이름이 `basics`인 패키지를 생성
- `$ catkin_create_pkg basics rospy`
- 이 후, `basics` 폴더로 이동하여, 정상적으로 패키지가 생성되었는지 확인합니다.
- `$ cd basics && ls`

```
wego@wego-GF63-Thin-10SCXR:~/study_ws/src$ catkin_create_pkg basics rospy
Created file basics/CMakeLists.txt
Created file basics/package.xml
Created folder basics/src
Successfully created files in /home/wego/study_ws/src/basics. Please adjust the
values in package.xml.
wego@wego-GF63-Thin-10SCXR:~/study_ws/src$ cd basics && ls
CMakeLists.txt  package.xml  src
wego@wego-GF63-Thin-10SCXR:~/study_ws/src/basics$
```

## 02 ROS Topic

- 패키지 생성 시, 패키지 매니페스트인 `package.xml`이 생성됩니다.
- 이를 에디터를 이용하여 사용자의 환경에 맞게 수정하여 사용할 수 있습니다.
- `package.xml`에 대한 수정 및 작성은 생성된 파일에 예시를 참고하면 됩니다.

## 02 ROS Topic

```
1 <?xml version="1.0"?>
2 <package format="2">
3   <name>basics</name>
4   <version>0.0.0</version>
5   <description>The basics package</description>
6
7   <!-- One maintainer tag required, multiple allowed, one person per tag -->
8   <!-- Example: -->
9   <!-- <maintainer email="jane.doe@example.com">Jane Doe</maintainer> -->
10  <maintainer email="wego@todo.todo">wego</maintainer>
11
12
13  <!-- One license tag required, multiple allowed, one license per tag -->
14  <!-- Commonly used license strings: -->
15  <!--   BSD, MIT, Boost Software License, GPLv2, GPLv3, LGPLv2.1, LGPLv3 -->
16  <license>TODO</license>
17
18
19  <!-- Url tags are optional, but multiple are allowed, one per tag -->
20  <!-- Optional attribute type can be: website, bugtracker, or repository -->
21  <!-- Example: -->
22  <!-- <url type="website">http://wiki.ros.org/basics</url> -->
23
24
25  <!-- Author tags are optional, multiple are allowed, one per tag -->
26  <!-- Authors do not have to be maintainers, but could be -->
27  <!-- Example: -->
28  <!-- <author email="jane.doe@example.com">Jane Doe</author> -->
29
30  <!-- Use buildtool_depend for build tool packages: -->
31  <!-- <buildtool_depend>catkin</buildtool_depend> -->
32
33  <!-- Use exec_depend for packages you need at runtime: -->
34  <!-- <exec_depend>message_runtime</exec_depend> -->
35
36  <!-- Use test_depend for packages you need only for testing: -->
37  <!-- <test_depend>gtest</test_depend> -->
38
39  <!-- Use doc_depend for packages you need only for building documentation: -->
40  <!-- <doc_depend>doxygen</doc_depend> -->
41
42  <buildtool_depend>catkin</buildtool_depend>
43  <build_depend>roscpp</build_depend>
44  <build_export_depend>roscpp</build_export_depend>
45  <exec_depend>roscpp</exec_depend>
46
47
48  <!-- The export tag contains other, unspecified, tags -->
49  <export>
50    <!-- Other tools can request additional information be placed here -->
51
52  </export>
53 </package>
54
55
56
57
58
59
60
61
62
63
```



## 02 ROS Topic

- 패키지 생성 시, C++를 사용 시, 빌드 환경에 대한 정보를 저장하는 CMakeLists.txt 파일도 생성됩니다.
- 일반적으로 사용하는 에디터를 이용하여 환경에 맞게 수정할 수 있습니다.
- Python 사용 시에는 특별한 수정 없이 사용할 수 있습니다.

## 02 ROS Topic

- 이 후, Topic 송신을 위한 Publisher 노드를 파이썬으로 작성합니다.
  - \$ cd basics
  - \$ mkdir scripts
  - \$ cd scripts && touch topic\_publisher.py && gedit topic\_publisher.py

```
1  #!/usr/bin/env python
2  import rospy
3  from std_msgs.msg import Int32
4
5  rospy.init_node('topic_publisher')
6
7  pub = rospy.Publisher('counter', Int32)
8
9  rate = rospy.Rate(2)
10
11  count = 0
12
13  while not rospy.is_shutdown():
14      pub.publish(count)
15      count += 1
16      rate.sleep()
```

## 02 ROS Topic

- 또한, Topic 수신을 위한 Subscriber 노드도 생성하고, 파이썬 코드를 작성합니다.
  - \$ roscd basics/scripts
  - \$ gedit topic\_subscriber.py

```
1  #!/usr/bin/env python
2
3  import rospy
4  from std_msgs.msg import Int32
5
6  def callback(msg):
7      print msg.data
8
9  rospy.init_node('topic_subscriber')
10
11 sub = rospy.Subscriber('counter', Int32, callback)
12
13 rospy.spin()
```

## 02 ROS Topic

- 이 후, 각각의 노드를 실행 가능하도록 실행 권한을 부여합니다.
  - `$ roscd basics/scripts`
  - `$ sudo chmod +x ./*`
- 마스터를 실행한 후, 각각의 노드를 실행하여 확인할 수 있습니다.
  - `$ roscore`
  - `$ rosrn basics topic_publisher.py`
  - `$ rosrn basics topic_subscriber.py`

## 02 ROS Topic

- Subscriber를 실행한 터미널에서 다음과 같은 결과를 확인할 수 있습니다.

```
wego@wego-GF63-Thin-10SCXR:~/study_ws$ source devel/setup.bash
wego@wego-GF63-Thin-10SCXR:~/study_ws$ rosrun basics topic_subscriber.py
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
```

## 02 ROS Topic

- rostopic 명령어로, 출력되는 Topic의 이름 및 메시지를 확인할 수 있습니다.
  - \$ rostopic list → Topic의 이름 확인
  - \$ rostopic echo /counter → 특정 Topic의 메시지 값 확인

```
^Cwego@wego-GF63-Thin-10SCXR:~/study_ws$ rostopic list
/counter
/rosout
/rosout_agg
wego@wego-GF63-Thin-10SCXR:~/study_ws$ rostopic echo /counter
data: 529
---
data: 530
---
data: 531
---
data: 532
---
data: 533
---
data: 534
---
data: 535
---
^Cwego@wego-GF63-Thin-10SCXR:~/study_ws$
```

**03**

---

## **ROS Service**

## 03 ROS Service

- 기존에 생성된 패키지에 서비스 타입을 추가 생성하겠습니다.
- Service 생성을 위해서는 새로운 타입의 Service를 등록해야합니다.
  - `$ cd ~/catkin_ws/src/basics`
  - `$ mkdir srv`
  - `$ cd srv && touch WordCount.srv`
- 이 후, WordCount.srv 파일을 에디터로 열어서, 아래와 같이 입력합니다.
- ---를 기준으로 위의 string words은 요청. 아래의 uint32 count는 응답에 사용할 메시지입니다.

```
string words
---
uint32 count
```



## 03 ROS Service

- 이에 해당하는 서비스 타입을 생성하기 위해, CMakeList.txt (아래 3개 이미지) 및 package.xml (위 1개 이미지) 에 아래와 같은 구문을 추가해줍니다.

```
<build_depend>message_generation</build_depend>  
<exec_depend>message_runtime</exec_depend>
```

```
find_package(catkin REQUIRED COMPONENTS  
  rospy  
  std_msgs  
  roscpp  
  message_generation  
)  
add_service_files(  
  FILES  
  WordCount.srv  
)  
generate_messages(  
  DEPENDENCIES  
  std_msgs # Or other packages containing msgs  
)
```

## 03 ROS Service

- 이후, `catkin_make`를 실행하면, `WordCount`, `WordCountRequest`, `WordCountResponse`에 해당하는 세 개의 클래스가 생성됩니다.
- 생성된 클래스는 서비스 요청 및 응답에 사용합니다.
- 생성 여부 확인을 위해,
  - `$rossrv show WordCount`
- 위 명령어를 입력하여, 서비스 타입이 새롭게 생성되었는지 확인할 수 있습니다.

```
wego/~/study_ws/ rossrv show WordCount
[basics/WordCount]:
string words
---
uint32 count
```

## 03 ROS Service

- 이제 새로운 서비스 타입을 정의하였고, 이를 사용하여 서비스를 구현하겠습니다.
- 서비스도 Topic과 동일하게 콜백 기반의 처리를 사용합니다.
- 서버는 서비스가 호출되면 실행하는 콜백 함수를 정의하고, 요청을 대기합니다.
  - `$ cd ~/catkin_ws/src/basics/scripts && touch service_server.py`
- 생성된 파일을 에디터로 열어서 아래와 같이 수정합니다.

```
#!/usr/bin/env python

import rospy

from basics.srv import WordCount, WordCountResponse

def count_words(request):
    return WordCountResponse(len(request.words.split()))

rospy.init_node('service_server')

service = rospy.Service('word_count', WordCount, count_words)

rospy.spin()
```

## 03 ROS Service

- 동일하게, rospy를 임포트하고, basics 패키지 아래의 srv 폴더에 생성된 WordCount와 응답 전달 시 사용할 WordCountResponse를 임포트합니다.
- 콜백 함수 count\_words를 정의하여, 요청 메시지의 글자 수를 입력으로 WordCountResponse에 전달하여 클래스를 생성한 후, 반환합니다.
- rospy.Service를 통해, 등록할 서비스의 이름, 타입, 콜백 함수를 지정합니다

```
#!/usr/bin/env python

import rospy

from basics.srv import WordCount, WordCountResponse

def count_words(request):
    return WordCountResponse(len(request.words.split()))

rospy.init_node('service_server')

service = rospy.Service('word_count', WordCount, count_words)

rospy.spin()
```

## 03 ROS Service

- 생성한 파이썬 스크립트에 아래의 명령어를 이용하여 권한 전달 후 실행합니다.
  - `$ chmod +x service_server.py`
  - `$ rosrun basics service_server.py`
- 아래 명령어를 이용하여, Service가 정상적으로 등록되었는 지 확인합니다.
  - `$ rosservice list`

```
wego/~/study_ws/ rosservice list
/rosout/get_loggers
/rosout/set_logger_level
/service_server/get_loggers
/service_server/set_logger_level
/word_count
```

## 03 ROS Service

- 아래의 명령어를 통해, 서비스 정보 또한 확인이 가능합니다.
  - `$ rosservice info /word_count`

```
wego/~/study_ws/ rosservice list
/roscout/get_loggers
/roscout/set_logger_level
/service_server/get_loggers
/service_server/set_logger_level
/word_count
```

```
wego/~/study_ws/ rosservice info /word_count
Node: /service_server
URI: rosrpc://wego-GF63-Thin-10SCXR:45453
Type: basics/WordCount
Args: words
```

## 03 ROS Service

- 서비스 호출은 `$ rosservice call` 명령어로 요청을 전달할 수 있습니다.
  - `$ rosservice call /word_count 'one two three'`

```
wego/~/study_ws/ rosservice call /word_count 'one two three'  
count: 3
```



## 03 ROS Service

- 서비스 서버와 같이, 클라이언트 노드를 스크립트로 생성할 수 있습니다.
  - `$ cd ~/catkin_ws/src/basics/scripts && touch service_client.py`
- 이 후, 생성된 파일을 에디터로 열어서 아래와 같이 수정합니다.

```
#!/usr/bin/env python

import rospy

from basics.srv import WordCount

import sys

rospy.init_node('service_client')
rospy.wait_for_service('word_count')
word_counter = rospy.ServiceProxy('word_count', WordCount)

words = ' '.join(sys.argv[1:])

word_count = word_counter(words)

print(words, '->', word_count.count)
```



## 03 ROS Service

- rospy를 임포트하고, 요청 전달을 위한 WordCount를 임포트합니다.
- rospy.wait\_for\_service를 통해, 서비스가 존재 여부를 확인한 후, rospy.ServiceProxy를 이용하여 서비스 호출 준비를 합니다. → 호출기 생성  
이에는 서비스 이름 및 서비스 형식을 명시해야합니다.
- 이 후, 서비스 호출기에 서비스 요청 메시지를 같이 전달하여 서비스를 호출한다.

```
#!/usr/bin/env python

import rospy

from basics.srv import WordCount

import sys

rospy.init_node('service_client')
rospy.wait_for_service('word_count')
word_counter = rospy.ServiceProxy('word_count', WordCount)

words = ' '.join(sys.argv[1:])

word_count = word_counter(words)

print(words, '->', word_count.count)
```

## 03 ROS Service

- Service Server를 실행한 상태에서, Service Client를 아래의 명령어로 실행한다.
  - `$ chmod +x service_client.py`
  - `$ rosrun basics service_client.py these are some words`
- 아래와 같이, 입력한 문자열에 포함된 단어의 개수를 응답으로 확인할 수 있다.

```
wego/basics/scripts/ rosrun basics service_client.py these are some words  
( 'these are some words', '->', 4)
```

**04**

---

## **ROS Action**

## 04 ROS Action

- Action은 Topic와 Service의 특징을 합친 형태로, 요청 후, 응답까지 걸리는 시간이 긴 경우 일반적으로 사용한다. ex) 목표 장소로 로봇을 이동할 때
- Action은 Service와 달리, 요청과 응답 뿐만 아니라, 진행 정보를 중간에 전달해주며, 목표(goal)과 결과(result) 그리고, 피드백(feedback)을 제공한다.
- 액션도 새롭게 정의 해주어야하며, 이는 패키지 내부의 action 폴더에 저장한다.
  - `$ cd ~/catkin_ws/src/basics && mkdir action && cd action && touch Timer.action`
- 위에서 생성한 파일을 에디터로 열어서 아래와 같이 수정한다.

```
duration time_to_wait
---
duration time_elapsed
uint32 updates_sent
---
duration time_elapsed
duration time_remaining
```

## 04 ROS Action

- 액션 파일은 세 개의 영역으로 나뉘며, 최상단 영역은 클라이언트에서 전달하는 목표, 중간 영역은 서버에서 보내는 결과, 그리고 최하단 영역은 서버에서 보내는 피드백이다.
- 액션 생성을 위해 CMakeLists.txt(아래 4개 이미지), package.xml(위 1개 이미지)에 아래의 내용을 추가한다.

```
<build_depend>actionlib_msgs</build_depend>
<exec_depend>actionlib_msgs</exec_depend>
```

```
find_package(catkin REQUIRED COMPONENTS
  rospy
  std_msgs
  roscpp
  message_generation
  actionlib_msgs
)
```

```
add_action_files(
  FILES
  Timer.action
)
```

```
generate_messages(
  DEPENDENCIES
  std_msgs # Or other package
  actionlib_msgs
)
```

```
catkin_package(
  # INCLUDE_DIRS include
  # LIBRARIES basics
  CATKIN_DEPENDS rospy message_runtime actionlib_msgs
  # DEPENDS system_lib
)
```

## 04 ROS Action

- 이 후, `catkin_make`를 실행하면, 액션 사용을 위한, `TimerAction.msg`, `TimerActionFeedback.msg`, `TimerActionGoal.msg`, `TimerActionResult.msg`, `TimerFeedback.msg`, `TimerGoal.msg`, `TimerResult.msg`가 생성되며, 이를 이용하여 액션을 구현할 수 있다.

## 04 ROS Action

- 생성된 메시지들을 이용하여, 액션 서버를 구현할 수 있으며, 콜백 함수를 사용한다.
- 가장 쉬운 방법은 actionlib 패키지에 있는 SimpleActionServer 클래스를 사용하는 방법이다.
- `$ cd ~/catkin_ws/src/basics/scripts && touch simple_action_server.py`
- 위에서 생성한 스크립트를 에디터로 열어 다음과 같이 수정합니다.

```
#!/usr/bin/env python
import rospy
import time
import actionlib
from basics.msg import TimerAction, TimerGoal, TimerResult

def do_timer(goal):
    start_time = time.time()
    time.sleep(goal.time_to_wait.to_sec())
    result = TimerResult()
    result.time_elapsed = rospy.Duration.from_sec(time.time() - start_time)
    result.updates_sent = 0
    server.set_succeeded(result)

rospy.init_node('timer_action_server')
server = actionlib.SimpleActionServer('timer', TimerAction, do_timer, False)
server.start()
rospy.spin()
```

## 04 ROS Action

- rospy를 импорт하며, action 사용을 위한 actionlib 및, 생성된 TimerAction, TimerGoal, TimerResult를 импорт합니다.
- 요청을 받은 후, 사용할 콜백 함수를 정의하고, actionlib.SimpleActionServer를 이용하여, 액션의 이름, 액션 타입, 콜백함수, 자동 실행 여부에 대한 정보를 등록하고, 서버를 실행합니다.

```
#!/usr/bin/env python
import rospy
import time
import actionlib
from basics.msg import TimerAction, TimerGoal, TimerResult

def do_timer(goal):
    start_time = time.time()
    time.sleep(goal.time_to_wait.to_sec())
    result = TimerResult()
    result.time_elapsed = rospy.Duration.from_sec(time.time() - start_time)
    result.updates_sent = 0
    server.set_succeeded(result)

rospy.init_node('timer_action_server')
server = actionlib.SimpleActionServer('timer', TimerAction, do_timer, False)
server.start()
rospy.spin()
```



## 04 ROS Action

- 아래의 명령어를 통해 액션 서버를 실행한다
  - `$ chmod +x simple_action_server.py`
  - `$ rosrun basics simple_action_server.py`
- 아래의 명령어를 이용하여, 액션에 해당하는 토픽을 확인할 수 있다.
  - `$ rostopic list`

```
wego/~/study_ws/ rostopic list
/rosout
/rosout_agg
/timer/cancel
/timer/feedback
/timer/goal
/timer/result
/timer/status
```

## 04 ROS Action

- 액션 사용의 경우, rostopic으로 goal에 직접 목표를 전달하여 사용할 수 있으며, client에 해당하는 노드를 작성하여 사용할 수도 있다.
  - `$ cd ~/catkin_ws/src/basics/scripts && touch simple_action_client.py`
- 위를 통해 생성한 client를 아래와 같이 수정한다.

```
#!/usr/bin/env python
import rospy

import actionlib
from basics.msg import TimerAction, TimerGoal, TimerResult

rospy.init_node('timer_action_client')
client = actionlib.SimpleActionClient('timer', TimerAction)
client.wait_for_server()
goal = TimerGoal()
goal.time_to_wait = rospy.Duration.from_sec(5.0)
client.send_goal(goal)
client.wait_for_result()
print('Time elapsed : %f'%(client.get_result().time_elapsed.to_sec()))
```

## 04 ROS Action

- `actionlib.SimpleActionClient`를 이용하여, Action server의 이름과 타입을 전달하고 server의 유무 확인 후, 목표를 전달할 `TimerGoal`의 인스턴스를 생성한다.
- 이 후, 목표에 있는 `time_to_wait`를 원하는 값으로 전달한 후, `client`를 이용하여 목표를 전달하고, 결과를 기다린다.

```
#!/usr/bin/env python
import rospy

import actionlib
from basics.msg import TimerAction, TimerGoal, TimerResult

rospy.init_node('timer_action_client')
client = actionlib.SimpleActionClient('timer', TimerAction)
client.wait_for_server()
goal = TimerGoal()
goal.time_to_wait = rospy.Duration.from_sec(5.0)
client.send_goal(goal)
client.wait_for_result()
print('Time elapsed : %f'%(client.get_result().time_elapsed.to_sec()))
```

## 04 ROS Action

- 각각의 Action Server와 Client를 실행하여, 정상적으로 Action이 동작하는지 확인한다.
  - \$ rosrn basics simple\_action\_server.py
  - \$ rosrn basics simple\_action\_client.py

```
wego/~/ rosrn basics simple_action_client.py  
Time elapsed : 5.005187
```

## 04 ROS Action

### fancy\_action\_server.py

```
#!/usr/bin/env python
import rospy
import time
import actionlib
from action_msgs.msg import TimerAction, TimerGoal, TimerResult, TimerFeedback

def do_timer(goal):
    start_time = time.time()
    update_count = 0

    if goal.time_to_wait.to_sec() > 60.0:
        result = TimerResult()
        result.time_elapsed = rospy.Duration.from_sec(time.time() - start_time)
        result.updates_sent = update_count
        server.set_aborted(result, 'Timer aborted due to too-long wait')
        return

    while (time.time() - start_time) < goal.time_to_wait.to_sec():

        if server.is_preempt_requested():
            result = TimerResult()
            result.time_elapsed = rospy.Duration.from_sec(time.time() - start_time)
            result.updates_sent = update_count
            server.set_preempted(result, 'Timer preempted')
            return

        feedback = TimerFeedback()
        feedback.time_elapsed = rospy.Duration.from_sec(time.time() - start_time)
        feedback.time_remaining = goal.time_to_wait - feedback.time_elapsed
        server.publish_feedback(feedback)
        update_count += 1

        time.sleep(1.0)

    result = TimerResult()
    result.time_elapsed = rospy.Duration.from_sec(time.time() - start_time)
    result.updates_sent = update_count
    server.set_succeeded(result, 'Timer completed successfully')

rospy.init_node('timer_action_server')
server = actionlib.SimpleActionServer('timer', TimerAction, do_timer, False)
server.start()
rospy.spin()
```



## 04 ROS Action

### fancy\_action\_client.py

```
#!/usr/bin/env python
import rospy

import time
import actionlib
from action_study.msg import TimerAction, TimerGoal, TimerResult, TimerFeedback

def feedback_cb(feedback):
    print('[Feedback] Time elapsed : %f'%(feedback.time_elapsed.to_sec()))
    print('[Feedback] Time remaining : %f'%(feedback.time_remaining.to_sec()))

rospy.init_node('timer_action_client')
client = actionlib.SimpleActionClient('timer', TimerAction)
client.wait_for_server()

goal = TimerGoal()
goal.time_to_wait = rospy.Duration.from_sec(5.0)

# goal.time_to_wait = rospy.Duration.from_sec(500.0)
client.send_goal(goal, feedback_cb = feedback_cb)

# time.sleep(3.0)
# client.cancel_goal()

client.wait_for_result()
print('[Result] State : %d'%(client.get_state()))
print('[Result] Status : %s'%(client.get_goal_status_text()))
print('[Result] Time elapsed : %f'%(client.get_result().time_elapsed.to_sec()))
print('[Result] Updates sent : %d'%(client.get_result().updates_sent))
```

**05**

---

## **ROS Message**

## 05 ROS Message

- 만약 기존에 존재하는 메시지 타입이 아닌 새로운 형태의 메시지를 정의하여 사용하고 싶을 경우, 새롭게 메시지를 정의하여 사용할 수 있습니다.
- 메시지의 경우, 패키지 내부에 msg이름의 폴더를 생성하고, 내부에 메시지의 타입을 정의하여 사용하게 됩니다.
  - \$ roscd basics
  - \$ mkdir msg && cd msg
  - \$ gedit Complex.msg

```
float32 real  
float32 imaginary  
|
```



## 05 ROS Message

- 이 후, 새롭게 메시지를 빌드하기 위해, package.xml 파일에 아래의 줄을 추가합니다.
  - \$ roscd basics
  - \$ gedit package.xml

```
<build_depend>std_msgs</build_depend>  
<exec_depend>std_msgs</exec_depend>  
<build_depend>message_generation</build_depend>  
<exec_depend>message_runtime</exec_depend>
```

## 05 ROS Message

- 또한, CMakeLists를 수정하여, 메시지를 빌드할 수 있게 합니다.

```
find_package(catkin REQUIRED COMPONENTS
  rospy
  std_msgs
  roscpp
  message_generation
)
add_message_files(
  FILES
  Complex.msg
  # Message2.msg
)
generate_messages(
  DEPENDENCIES
  std_msgs # Or other packages containing msgs
)
catkin_package(
  # INCLUDE_DIRS include
  # LIBRARIES basics
  CATKIN_DEPENDS rospy message_runtime
  # DEPENDS system_lib
)
include_directories(
  # include
  ${catkin_INCLUDE_DIRS}
)
```

## 05 ROS Message

- 이 후, 원래의 작업 공간으로 이동하여, 메시지를 빌드합니다.
  - `$ cd ~/catkin_ws && catkin_make`
- 정상적으로 빌드가 되었다면 C++ 헤더의 경우, `~/catkin_ws/devel/include/basics`에 생성한 메시지 이름의 헤더가 생성됩니다.
- 마찬가지로 파이썬 클래스의 경우,  
`~/catkin_ws/devel/lib/python2.7/dist-packages/basics/msg`에 생성한 메시지 이름의 파이썬 클래스가 생성됩니다.
- 이 후, 다른 패키지에서 이에 해당하는 헤더파일 또는 클래스를 추가하여 사용할 수 있습니다.



# WeGo Robotics

**Tel.** 031 – 229 – 3553

**Fax.** 031 – 229 – 3554



**제품 문의:** [go.sales@wego-robotics.com](mailto:go.sales@wego-robotics.com)

**기술 문의:** [go.support@wego-robotics.com](mailto:go.support@wego-robotics.com)