ROS Noetic 講習資料

つくばろぼっとサークル

2022 年 8 月

はじめに

まえがき

ロボットを動かす上で ROS の知識は欠かせないものだ。しかし、初心者にとって ROS のコマンドラインインターフェースは理解しにくい上、覚えにくい。そこで、わかりにくい ROS の概要を明文化して少しでも ROS を習得できる人が増えることを期待して、この資料を作成した。筆者は ROS および Linux への理解が深いわけではない上、急ぎ作成したものであるため、誤っている点や煩雑な点などがあるかもしれないが、ご了承願いたい。

2021 年 10 月
小林照

増補改訂にあたって

今年度の改訂では、より初学者に配慮し、Ubuntu のインストールやコマンドラインインターフェイスへの導入から記述し、またロボット製作で多用する Arduino の統合や、多様な型のメッセージの試用までを盛り込んだ。また、Ubuntu 18.04 のサポート終了が迫っていることを受け、使用するディストリビューションを Noetic に変更した。

出来るだけ要点を抑えられるよう努力したが、省略した箇所も多いので、公式ドキュメントなども併せて参照してほしい。また、理解不足から誤りが含まれている可能性も大いにある。ご了承ください。

2022 年 8 月
小西博翔

Copyright Disclaimer

第 1 章　Ubuntu をはじめよう

1.1　Ubuntu

Ubuntu は、広く使われている OS の 1 つである。OS(Operating System) とはコンピュータの基幹となるシステムであり、例えば我々のよく使う Windows はこの一種である。

Ubuntu は、Linux のディストリビューションであり、また、Linux は UNIX 系 OS と呼ばれている。この関係は、単に Ubuntu を使用する際に意識する必要は必ずしもないが、これを知っておくと、インターネットなどで文献をあたるときに役立つかもしれない。

Ubuntu は、フリー (自由かつ無償) ソフトウェアであり、公式サイトから入手できる。Ubuntu には様々なバージョンがあるが、今回は 20.04LTS を使うこととする。

可能であればネイティブ環境を用意することが望ましいが、困難である場合、Virtual Box というソフトウェアを用いて Windows 上に仮想的な Ubuntu 環境を構築することができるので活用してほしい。ちなみに、WSL という異なる構築方法も存在しており、用途によっては有用であるが、ROS での使用には適さないのでここでは使わないでほしい。

ネイティブ環境への Ubuntu のインストール

公式サイトにインストールのチュートリアルがあるので、説明は省略する。`https://ubuntu.com/tutorials/install-ubuntu-desktop`

Virtual Box を用いた環境構築

Virtual Box は、仮想マシン (Virtual Machine) を作るソフトウェアである。ホストとなる OS の上で、1 つの仮想的なコンピュータとして動作する。これを用いることで、Windows 上に Ubuntu の環境を構築する。

Windows への Virtual Box のインストール

公式サイトから最新バージョンのインストーラをダウンロードし、インストールする。

Virtual Box への Ubuntu のインストール

iso ファイルなどを用いて、Virtual Box 上に Ubuntu 環境を構築する。詳細は省略する。

Ubuntu のインストール

自分の環境に合わせ、Ubuntu 20.04 をインストール・セットアップしよう。

1.2 GUI と CLI

普段、あなたがコンピュータを使う時のことを考えてみよう。おそらく、画面上にはいろいろなウィンドウ・ボタン・アイコンが表示されているはずである。この 1 例として、Windows 11 のエクスプローラーを図 1.1 に示す。このようなユーザーインターフェースのことを、GUI(Graphical User Interface) という。

一方、図 1.2 に示すようなユーザーインターフェースを CUI(Character User Interface) あるいは CLI(Command Line Interface) などと呼ぶ。CLI では、アイコンやボタンは存在せず、テキストによる表示と「コマンド」と呼ばれるテキストで表す指令によって操作・表示が行われる。

Ubuntu 自体は GUI を持つものの、CLI による操作も多用される。標準的な ROS は GUI を持たないため、CLI で操作する必要がある。少しずつ CLI にも慣れていこう。

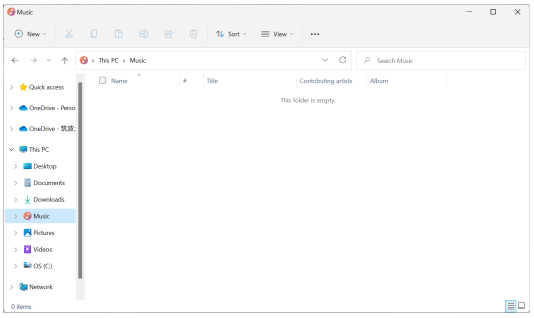


図 1.1　エクスプローラー

```
C:\>dir
 Volume in drive C is OS
 Volume Serial Number is B687-05F1

 Directory of C:\

2021/03/29  21:54    <DIR>          Apps
2021/03/30  15:07    <DIR>          DELL
2021/03/30  14:36    <DIR>          Drivers
2022/06/16  14:11    <DIR>          Intel
2020/05/06  09:03    <DIR>          langpacks
2021/06/05  21:10    <DIR>          PerfLogs
2022/07/14  16:42    <DIR>          Program Files
2022/05/11  10:46    <DIR>          Program Files (x86)
2022/02/01  11:36    <DIR>          Users
2021/11/11  10:54    <DIR>          vscode
2022/06/20  02:55    <DIR>          Windows
               0 File(s)              0 bytes
              11 Dir(s)  119,707,574,272 bytes free

C:\>
```

図 1.2　コマンドプロンプト

1.3 Ubuntu の CLI

Ubuntu においては、Bash というシェル[1)]がデフォルトで使用される。シェルの入門については、The Linux command line for beginners や The Missing Semester of Your CS Education など、参考になり信頼できるサイトが多くあるので省略することとし、本資料では、初学者が詰まりがちなポイントに絞って解説する。

1)　ユーザーのコマンド入力を処理するプログラム

(補) コマンドライン操作

Linux/UNIX 系 OS 上のコマンドライン操作に慣れていない者は、Ubuntu の「端末」(Terminal) を用い、習熟しておこう。

1.3.1 sudo

シェルでは、多彩なコマンドを実行することができるが、その一部には、危険なものやシステムに影響を及ぼすものも多くある。これらのコマンドは、一般ユーザーには実行できないようになっている。そのようなコマンドを実行するには、sudo を使う。sudo は、“switch user and do this command” の略で、ユーザーを（特別な権限のあるものに）切り替えてコマンドを実行することを示す。

sudo を使う際には、コマンドの頭に “sudo ” を付ける。実行時にはパスワードの入力を要求される[2]ので、**そのときログインしているユーザーの**パスワードを入力する。[3]

たとえば、`apt update` というコマンド（意味は後述する）を実行するには、`sudo apt update` と入力することとなる。

1.3.1.1 パスワード

sudo の使用時をはじめとして、シェルにパスワード入力を要求されることはままある。Windows のインターフェイスに慣れているならば、そのような時には「*****」のような表示が出るイメージがあるかもしれないが、UNIX 系のソフトウェアでは画面に何も出ないことが多いので注意してほしい。

(補) コマンドライン操作

`sudo apt update` して、挙動を確認してみよう。

1.3.2 apt

1.3.2.1 apt と apt-get

Ubuntu や ROS などについてインターネットで検索していると、`apt` と `apt-get` という 2 つのコマンドを目にすることが多いだろう。これらは、簡単に説明すると、OS 上のアプリを管理するものだ。ソフトウェアをインストール・更新・アンインストールするときなどに用いられる。

これら 2 つは、(誤解を恐れずに言うならば) 大した違いはないが、`apt` の方が `apt-get` より新しい分やや優れている。古い文献では `apt-get` が使われがちである。適宜 `apt` に書き換えても（ほ

2） ただし、パスワードを入れたばかりの時は要求されない
3） root ユーザーのパスワードではないことに注意する。

とんどの場合）差し支えない。詳細は興味を持ったときに自分で調べてみてほしい。

1.3.2.2 terminator のインストール

ここで、apt の練習も兼ね、ROS 開発の際によく使うソフトをインストールしてみよう。

Terminator は、今使っている端末と似たようなソフトウェアであるが、画面を分割して複数のバッシュを 1 画面内に表示させることができ便利である。

```
1 sudo apt update
2 sudo apt install terminator
```

ここで、`sudo apt update` は、ソフトウェアパッケージのリスト[4)]を更新するコマンド[5)]であり、`sudo apt install terminator` は、Terminator をインストールするコマンドである。

apt を用いたインストール

Terminator をインストールしよう。

1.3.2.3 インストール済ソフトウェアの更新

```
1 sudo apt update
2 sudo apt upgrade
```

ソフトウェアの更新

ソフトウェアの更新を行おう。

4) リストには、ソフトウェアの最新バージョンがいくらであるとか、インストーラがどこで手に入るとか、そういった類の情報がまとまっている。

5) apt を実行するときは、必ずこのコマンドを先に実行しよう。

第 2 章　Hello, ROS
—ROS のセットアップと単純な Pub/Sub—

2.1　インストール

ROS の詳細を説明する前に、まずはインストールを済ませてしまおう。Ubuntu 20.04LTS に対応する ROS のバージョンは Noetic である。公式サイトのガイドラインに従って、インストールすることができる。https://wiki.ros.org/noetic/Installation/Ubuntu

これらのコマンドを逐一実行してもよいのだが、これらを一度に済ませることができるスクリプトがあるので、今回はこれを用いよう。

```
1  wget -c https://raw.githubusercontent.com/qboticslabs/ros_install_noetic/master/
       ros_install_noetic.sh && chmod +x ./ros_install_noetic.sh && ./
       ros_install_noetic.sh
```

上記のコマンドを入力することでスクリプトを実行し、画面の指示に従うと、インストールを行える。

ROS のインストール

ROS Noetic をインストールしよう。

インストールには数十分かかることがある。気長に待とう。

ここで、ROS の開発に便利なツールである catkin_tools もインストールしておこう。上記によりスクリプトを実行した後[1)]、

```
1  sudo apt install python3-catkin-tools
```

を実行する。

2.2　ROS とは

ROS は、Robot Operating System の略称である。ロボットを制御するためのオープンソースソフトウェアであり、本サークルでソフト班に所属する人は、このソフトウェアを使用することになる。基本的にはバッシュからコマンドを打ち込んで使用する。

1)　わざわざ順序を指定しているのには理由がある。apt によってインストールできるソフトウェアは、登録されているリポジトリのソフトだけだ。ROS 関連のソフトウェアが掲載されているリポジトリは、デフォルトでは登録されておらず、上のスクリプトの途中で登録されているのだ。つまり、スクリプトの実行前に catkin_tools をインストールしようとしても、自分でリポジトリを登録しない限りはインストールできないのである。

混乱しやすいことに ROS は Operating System という名前を持つものの、Ubuntu などの OS 上で動作するソフトウェアに過ぎない。

2.3 ROS のディレクトリ構成

ROS を使用する環境であるディレクトリ構成について説明する。図 2.1 のディレクトリ構成図を見ながら読み進めてほしい。

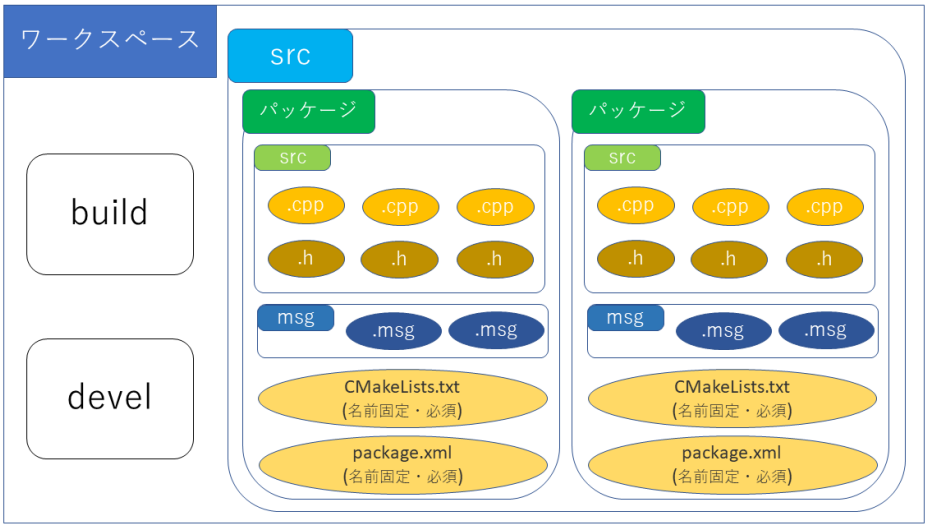


図 2.1 ROS のディレクトリ構成図

2.3.1 ワークスペース

ROS 関係のファイル全てが置かれたディレクトリ。名前は任意である。ワークスペースの中には build や devel,src などのディレクトリが存在する。src の中に各パッケージ (後述) を入れる。(この資料では build や devel にはこれ以上触れない。)

2.3.2 パッケージ

目的に沿ったノードやメッセージなどが入ったディレクトリ。中には src ディレクトリや CMakeLists.txt（名前固定・必須）や package.xml(名前固定・必須) などが存在する。src には.cpp や.h といったノード (後述) のソースコードが置かれる。CMakeLists.txt はビルド (コンパイル) 時に読み込まれるファイルで、コンパイル条件などを記述する。package.xml にはパッケージに必須の情報を記述する。パッケージ名はディレクトリ名ではなく、package.xml の中の name タグによって決められている。一般的にはパッケージ単位で開発を行う。

確認問題

ワークスペースとパッケージの関係を述べよ。

2.4　ROS のグラフ概念

ROS を理解する上で重要なグラフ概念について説明する。図 2.2 のグラフ概念図を見ながら読み進めてほしい。

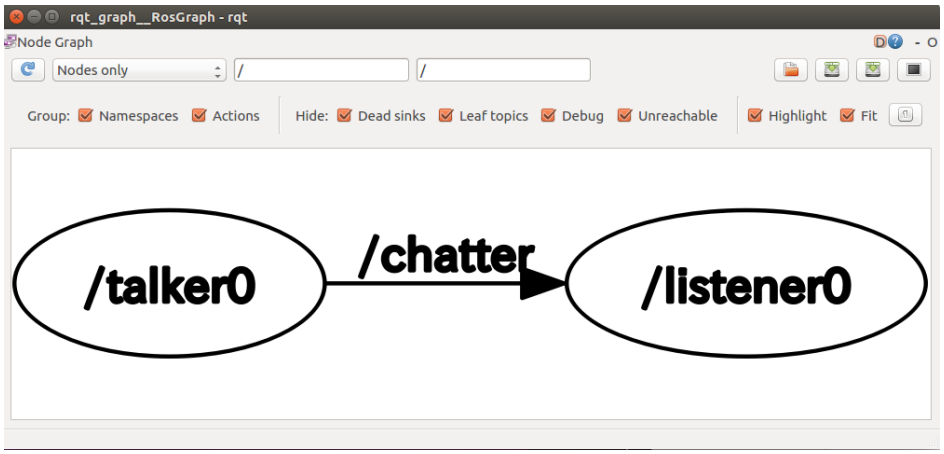


図 2.2　ROS のグラフ概念図

2.4.1　ノード

ROS パッケージ内の実行ファイルのことである。図 2 の “talker0” と “listener0” にあたる。ノードは他のノードと通信をして、情報のやりとりをする。ROS では、このノード同士の通信によって目的のシステムを構成していくことになる。一般的には Python や C++などのプログラミング言語を使って内容を記述する。

2.4.2　トピック

ノード同士がやりとりするデータのこと。図 2 の “chatter” にあたる。正確には異なるが、「ノード同士の通信の名前」と捉えるとプログラムが書きやすいかもしれない。トピックを送ることを Publish といい、送る側のノードを Publisher という。また、トピックを受け取ることを Subscribe といい、受け取る側のノードを Subscriber という。

2.4.3　メッセージ

トピックのデータ型のこと。Arduino や C 言語と同じように、整数や浮動小数、配列などの型が標準で用意されている。

カスタムメッセージをつくることで、自分で新たなメッセージ (型) を定義することもできる。これを活用すると、整数や浮動小数などを複数もつメッセージの作成もできる。

2.4.4 マスター

厳密にはグラフ概念に関する用語ではないが、後の説明を理解する上で役立つので、ここで解説しておく。マスターとは、ノードの名前の管理を行っているもののこと。ノード同士が通信をするためには、マスターが立ち上がっている必要がある。マスターを立ち上げるためのコマンドは roscore。(「ノードの実行」のところで再度出てくる。)

確認問題

- トピックとメッセージはどう異なるか。
- マスターはどのような役割を持つか。

2.5 ROS の使用手順

2.5.1 ワークスペースの作成

ROS には、ワークスペースをつくるためのコマンドは存在しない。したがって、Linux のコマンドを使用して、ワークスペースとなるディレクトリを作成する。例えば、“catkin_ws” という名前のワークスペースを作成する場合、以下のコマンドを実行する。

```
1 source /opt/ros/noetic/setup.bash
2 mkdir -p catkin_ws/src
```

mkdir はディレクトリを作成するためのコマンドで、-p をつけることで複数階層のディレクトリを 1 度に作成できる。このコマンドでは、catkin_ws の中にさらに src を作成している。

ワークスペースの作成

- 以上のコマンドを実行し、ワークスペース catkin_ws を作成しよう。
- ls コマンドを用いるなどして、作成したワークスペースのディレクトリ構成を確認してみよう。

また、その後に以下のコマンドを実行して、エラーが出ずにビルドできることを確認してみよう。

```
1 cd catkin_ws
2 catkin build
3 source devel/setup.bash
```

cd はディレクトリを移動するためのコマンドで、上記の例では catkin_ws の中に移動している。catkin build はビルドを行うためのコマンドで、ワークスペース内のプログラムやファイルの内容を ROS に解釈させる (読み込ませる)。プログラムの構文やファイルに誤りがある場合はエラーが出る。ワークスペース内のファイルに変更を加えた場合は、プログラムを実行する前に必ず catkin build を実行し、エラーが出ないことを確認する癖をつけよう。もしエラーが出た場合はまず source devel/setup.bash のコマンドを実行してみて、改善しないときはエラーコードで検索をかけるなどして調べるか、先輩に聞いてみよう。

ワークスペースの確認

以上のコマンドを実行しよう。

2.5.2 パッケージの作成

ROS にはパッケージを作成するためのコマンドが用意されている。例えば、“beginner_tutorials” という名前のパッケージを作成する場合、**cd コマンドでワークスペース内の src の中まで移動してから**、以下のコマンド[2)]を実行する。

```
1 catkin_create_pkg beginner_tutorials std_msgs rospy roscpp
```

パッケージの作成

以上のコマンドを実行し、パッケージ beginner_tutorials を作成しよう。

2.5.3 メッセージの作成

ここでは、オリジナルのメッセージファイル (カスタムメッセージ) の作成方法を説明する。**cd コマンドで、先ほど作成したパッケージの中まで移動してから**、以下のコマンドを実行する。

```
1 mkdir msg
2 echo "int64 num" > msg/Num.msg
```

mkdir コマンドで msg というディレクトリを作成している。次に echo コマンドで msg の中に Num.msg というメッセージファイルを作成すると同時に、Num.msg に “int64 num” という内容を書き込んでいる。“int64 num” とは 64bit 長の整数型の num という名前の変数を表している。つまり、Num.msg は上記の変数 num だけをもつメッセージということになる。

これから package.xml や CMakeLists.txt の中身を変更したり、プログラムを書いたりしていくが、この際必ず**半角**を使うこと。特に全角のスペースなどは入れないようにする。

2) なお、ここで紹介した catkin_create_pkg のほか、catkin create pkg も良く用いられる。

メッセージファイル自体は作成できたが、次は作成したメッセージファイルをパッケージに認識させる必要がある。そこでパッケージ内にある package.xml を開いて図 2.3 の行があることを確認し、図 2.4 のようにコメントを解除した後、保存する。（**右側の”–>” も消す。**）

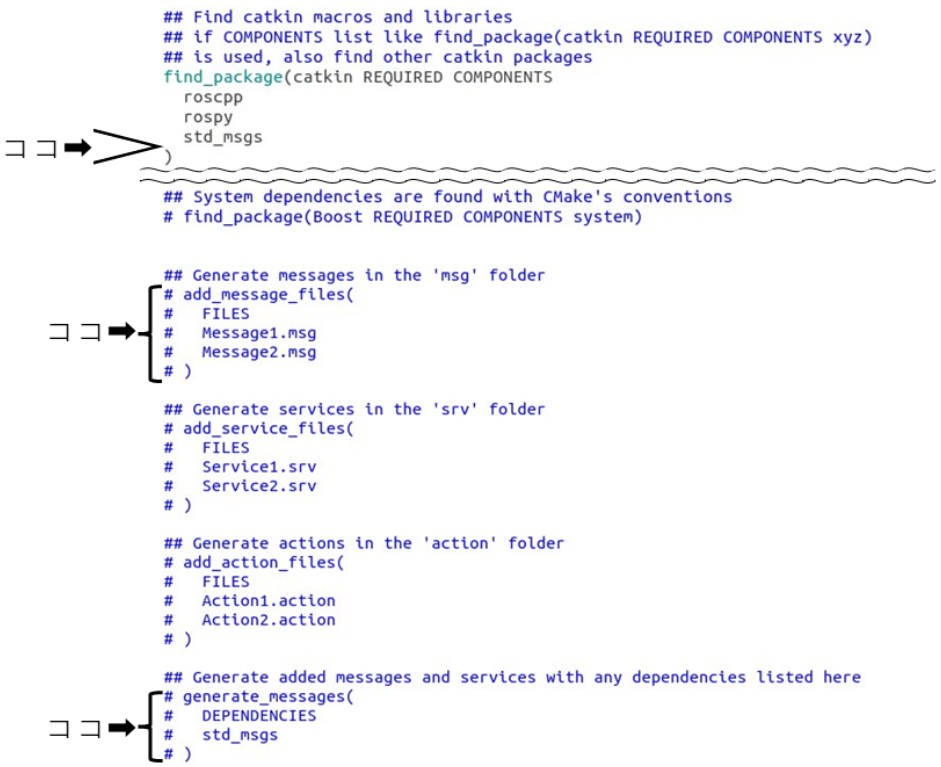
```
  <!-- The *depend tags are used to specify dependencies -->
  <!-- Dependencies can be catkin packages or system dependencies -->
  <!-- Examples: -->
  <!-- Use depend as a shortcut for packages that are both build and exec dependencies -->
  <!--   <depend>roscpp</depend> -->
  <!--   Note that this is equivalent to the following: -->
  <!--   <build_depend>roscpp</build_depend> -->
  <!--   <exec_depend>roscpp</exec_depend> -->
  <!-- Use build_depend for packages you need at compile time: -->
ココ➡<!--   <build_depend>message_generation</build_depend> -->
  <!-- Use build_export_depend for packages you need in order to build against this package: -->
  <!--   <build_export_depend>message_generation</build_export_depend> -->
  <!-- Use buildtool_depend for build tool packages: -->
  <!--   <buildtool_depend>catkin</buildtool_depend> -->
  <!-- Use exec_depend for packages you need at runtime: -->
ココ➡<!--   <exec_depend>message_runtime</exec_depend> -->
  <!-- Use test_depend for packages you need only for testing: -->
  <!--   <test_depend>gtest</test_depend> -->
  <!-- Use doc_depend for packages you need only for building documentation: -->
  <!--   <doc_depend>doxygen</doc_depend> -->
  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>roscpp</build_depend>
  <build_depend>rospy</build_depend>
  <build_depend>std_msgs</build_depend>
  <build_export_depend>roscpp</build_export_depend>
  <build_export_depend>rospy</build_export_depend>
  <build_export_depend>std_msgs</build_export_depend>
  <exec_depend>roscpp</exec_depend>
  <exec_depend>rospy</exec_depend>
  <exec_depend>std_msgs</exec_depend>
```

図 2.3　メッセージ作成時の package.xml（変更前）

```
  <!-- The *depend tags are used to specify dependencies -->
  <!-- Dependencies can be catkin packages or system dependencies -->
  <!-- Examples: -->
  <!-- Use depend as a shortcut for packages that are both build and exec dependencies -->
  <!--   <depend>roscpp</depend> -->
  <!--   Note that this is equivalent to the following: -->
  <!--   <build_depend>roscpp</build_depend> -->
  <!--   <exec_depend>roscpp</exec_depend> -->
  <!-- Use build_depend for packages you need at compile time: -->
ココ➡<build_depend>message_generation</build_depend>
  <!-- Use build_export_depend for packages you need in order to build against this package: -->
  <!--   <build_export_depend>message_generation</build_export_depend> -->
  <!-- Use buildtool_depend for build tool packages: -->
  <!--   <buildtool_depend>catkin</buildtool_depend> -->
  <!-- Use exec_depend for packages you need at runtime: -->
ココ➡<exec_depend>message_runtime</exec_depend>
  <!-- Use test_depend for packages you need only for testing: -->
  <!--   <test_depend>gtest</test_depend> -->
  <!-- Use doc_depend for packages you need only for building documentation: -->
  <!--   <doc_depend>doxygen</doc_depend> -->
  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>roscpp</build_depend>
  <build_depend>rospy</build_depend>
  <build_depend>std_msgs</build_depend>
  <build_export_depend>roscpp</build_export_depend>
  <build_export_depend>rospy</build_export_depend>
  <build_export_depend>std_msgs</build_export_depend>
  <exec_depend>roscpp</exec_depend>
  <exec_depend>rospy</exec_depend>
  <exec_depend>std_msgs</exec_depend>
```

図 2.4　メッセージ作成時の package.xml（変更後）

次に CMakeLists.txt を開いて図 2.5 の行を探し、図 2.6 のように変更した後、保存する。




```
## Find catkin macros and libraries
## if COMPONENTS list like find_package(catkin REQUIRED COMPONENTS xyz)
## is used, also find other catkin packages
find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  std_msgs
)

## System dependencies are found with CMake's conventions
# find_package(Boost REQUIRED COMPONENTS system)


## Generate messages in the 'msg' folder
# add_message_files(
#   FILES
#   Message1.msg
#   Message2.msg
# )

## Generate services in the 'srv' folder
# add_service_files(
#   FILES
#   Service1.srv
#   Service2.srv
# )

## Generate actions in the 'action' folder
# add_action_files(
#   FILES
#   Action1.action
#   Action2.action
# )

## Generate added messages and services with any dependencies listed here
# generate_messages(
#   DEPENDENCIES
#   std_msgs
# )
```

図 2.5　メッセージ作成時の CMakeLists.txt（変更前）

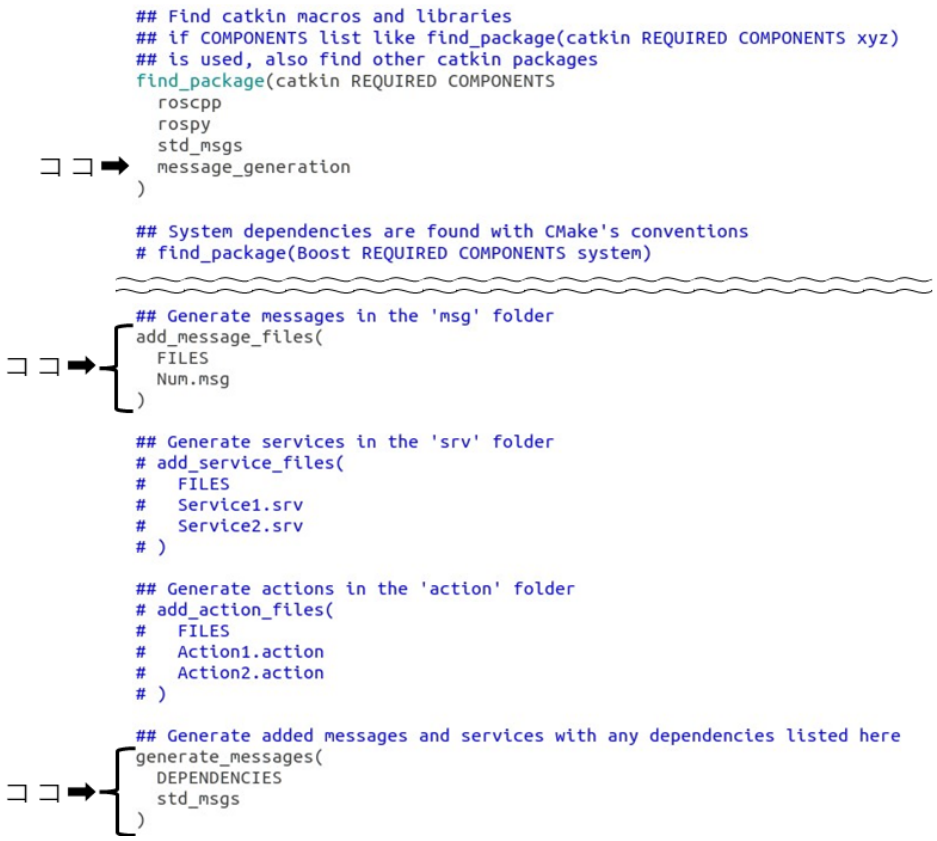


図 2.6　メッセージ作成時の CMakeLists.txt（変更後）

2.5.4　ノードの作成

パッケージ内にノードを作成しよう。ノードは、1 つのパッケージに対して複数作成できる。

今回は図 2.2 のように、Publisher と Subscriber を一つずつ作って通信させてみる。

まず、**cd コマンドでパッケージ内の src に移動した後**、以下のコマンドを実行して Publisher と Subscriber の.cpp ファイルを作成する。（ノードは Python で書くこともできるが、ここでは C++ で書くものとして説明する。）

```
1 touch talker.cpp
2 touch listener.cpp
```

touch はファイル作成を行うコマンドである。

ソースコードのファイルの作成

パッケージ `beginner_tutorials` の src ディレクトリに、talker.cpp・listener.cpp を作成しよう。

以下、talker.cpp を Publisher、listener.cpp を Subscriber としてプログラムを書き込んでいく。

2.5.4.1 Publisher の作成

Publisher として、ソースコード 2.1 に示すものを用いることとする。

ソースコード 2.1 talker.cpp

```
1  #include "ros/ros.h"
2  #include "std_msgs/String.h"
3
4  #include <sstream>
5
6  int main(int argc, char **argv)
7  {
8
9      ros::init(argc, argv, "talker");
10
11     ros::NodeHandle n;
12
13     ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);
14
15     ros::Rate loop_rate(10);
16
17     int count = 0;
18     while (ros::ok())
19     {
20         std_msgs::String msg;
21
22         std::stringstream ss;
23         ss << "hello world " << count;
24         msg.data = ss.str();
25
26         ROS_INFO("%s", msg.data.c_str());
27
28         chatter_pub.publish(msg);
29
30         ros::spinOnce();
31
32         loop_rate.sleep();
33         ++count;
34     }
35     return 0;
36 }
```

上記のプログラムを talker.cpp に書き込み、保存する。

以下はソースコードの解説。

```
1 #include "ros/ros.h"
```

ros/ros.h は、ROS のノードを書くときに使用するヘッダファイル。ROS のノードを書くときはとりあえず、このヘッダファイルを include する。

```
1 #include "std_msgs/String.h"
```

std_msgs は、既存で用意されたメッセージファイルがあるディレクトリのこと。String.h で、文字列を通信できる既存のメッセージファイル String.msg を読み込んでいる。

```
1 ros::init(argc, argv, "talker");
```

ROS の初期化を行っている。また第 3 引数にノード名を渡す。マスターは、このノード名によって各ノードを区別しているため、名前が他のノードと被ってはいけない。

```
1 ros::NodeHandle n;
```

ノードハンドル n を宣言している。この資料で扱うコードでは、Publish、Subscribe に関する情報をマスターに伝えるコードを書くために必要。(ノードハンドルについては筆者もこれ以上のことは理解していません。ごめんなさい m(___)m)

```
1   ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);
```

「chatter_pub という名前の Publisher は、chatter というトピックに、メッセージファイルとして std_msgs の中の String.msg を使って、メッセージを送る」という内容をマスターに伝えるためのコード。非常に複雑であるため、以下に構文を示す。

ros::Publisher Publisher の名前 = ノードハンドル名.advertise<メッセージファイル名>(“トピック名”, 1000);

第 2 引数の 1000 は、捨てずにとっておける情報量を表すが、基本的にはこの値は 1000 にしておけばよい。(少なくとも筆者は 1000 以外にしたことはない。)

```
1 ros::Rate loop_rate(10);
```

下の while 文のループ間隔が、1 秒間に 10 回になるように設定している。

```
1   while (ros::ok())
```

この while 文は ROS が起動している間、ループし続ける。

```
1 std_msgs::String msg;
2
3 std::stringstream ss;
```

String メッセージを扱う変数 msg と、文字列を扱う変数 ss を宣言している。(stringstream は、sstream を include した際に使える C++のクラスで、ROS のクラスではない。)

```
1   ss << "hello world " << count;
2   msg.data = ss.str();
```

“hello world” という文字列と、count の値の文字列をつなげた文字列を ss に代入し、さらにそれを msg.data に代入している。通常、std_msgs に用意されたメッセージのデータには、.data をつけることでアクセスできる。なお、「5.3 メッセージの作成」のところで作成した Num.msg のデータにアクセスするには、.num をつける。(ss.str() は、ss に格納された文字列を取り出している。stringstream の変数は、=だけでは代入できないことに注意。)

```
1   ROS_INFO("%s", msg.data.c_str());
```

ROS_INFO は、printf とほぼ同様のはたらきをする関数。(ただし、msg.data の中のデータ型は stringstream なので、.c_str() をつける必要がある。)

```
1 chatter_pub.publish(msg);
```

ここで、実際に Publish を行っている。

```
1 ros::spinOnce();
```

コールバック関数を呼び出しているが、このコードにおいては、あってもなくても影響はない。詳しくは「Subscriber の作成」にて。

```
1 loop_rate.sleep();
```

ros::Rate loop_rate で指定した時間でループできるように、残った時間待機する。

2.5.4.2 Subscriber の作成

ソースコード 2.2 listener.cpp

```
1  #include "ros/ros.h"
2  #include "std_msgs/String.h"
3
4  void chatterCallback(const std_msgs::String::ConstPtr &msg)
5  {
6      ROS_INFO("I heard: [%s]", msg->data.c_str());
7  }
8
9  int main(int argc, char **argv)
10 {
11
12     ros::init(argc, argv, "listener");
```

```
13
14      ros::NodeHandle n;
15
16      ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);
17
18      ros::spin();
19
20      return 0;
21  }
```

上記のプログラムを listener.cpp に書き込み、保存する。

以下、ソースコードの解説。なお、「Publisher の作成」と重複する箇所は省略する。

```
1   void chatterCallback(const std_msgs::String::ConstPtr &msg)
```

これは、トピックに新しいメッセージが届くと呼び出される関数で、このような関数を一般にコールバック関数という。引数の与え方が複雑であるため、以下に構文を示す。

void コールバック関数名 (const メッセージファイル名::ConstPtr& メッセージ変数名)

なお、メッセージ変数名はコールバック関数内でのみ使用される変数で、自由に定義できる。

```
1   ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);
```

「sub という名前の Subscriber は、chatter というトピックからメッセージを受け取り、その後に chatterCallback という関数を実行する」という内容をマスターに伝えるためのコード。非常に複雑であるため、以下に構文を示す。

ros::Subscriber Subscriber の名前 = ノードハンドル名.subscribe(“トピック名”, 1000, コールバック関数名);

第 2 引数の 1000 は、捨てずにとっておける情報量を表すが、基本的にはこの値は 1000 にしておけばよい。(少なくとも筆者は 1000 以外にしたことはない。)

```
1 ros::spin();
```

ここで、トピックにメッセージが届いているかを確認して、もし届いていたら適切なコールバック関数を呼び出す、ということをする。ros::spinOnce() との違いは、ros::spin() は ROS が起動している間、上記の処理をし続けるが、ros::spinOnce() は、一度だけ上記の処理をすることである。

Publisher と Subscriber の書き込み

以上の内容に従って、talker.cpp と listener.cpp にソースコードを書き込もう。

この節の初めに、.cpp ファイルを作成した時、`touch` コマンドを用いたことを思い出そう。このコマンドは ROS のコマンドではなく、OS 標準のコマンドである。ROS は、このソースファイルをまだ認識していないのだ。

CMakeLists.txt を開き、末尾に以下を追加して、保存する。

```
include_directories(include ${catkin_INCLUDE_DIRS})

add_executable(talker src/talker.cpp)
target_link_libraries(talker ${catkin_LIBRARIES})

add_executable(listener src/listener.cpp)
target_link_libraries(listener ${catkin_LIBRARIES})
```

Publisher と Subscriber の書き込み

CMakeLists.txt を編集しよう。

最後にワークスペース内で、catkin build と source devel/setup.bash を実行する。これにより、パッケージが build され、実行可能な状態になる。

```
catkin build
source devel/setup.bash
```

ビルド

パッケージを build しよう

2.6 ノードの実行

まず、ターミナルを 4 つ立ち上げる。ワークスペース内で以下の 4 つのコマンドを実行する。なお、適宜 `source devel/setup.bash` を実行すること。

1 つ目のターミナルに以下のコマンドを実行する。

```
roscore
```

roscore は、マスターを立ち上げるコマンドで、rosrun（後述）を実行するために必要。

2 つ目のターミナルに以下のコマンドを実行する。

```
rosrun beginner_tutorials talker
```

rosrun はノードを立ち上げるためのコマンドで、Publisher である talker を立ち上げた。構文は、“rosrun パッケージ名 ノード名”

3 つ目のターミナルに以下のコマンドを実行する。

```
rosrun beginner_tutorials listener
```

Subscriber である listener を立ち上げた。
4 つ目のターミナルに以下のコマンドを実行する。

```
1 rosrun rqt_graph rqt_graph
```

以上 4 つのコマンドを実行した結果、図 2 で示したのと同様のウィンドウが立ち上がれば、成功。

実行

ノードを実行し、出力を観察してみよう。

問題

listener.cpp を `ros::spinOnce()` を用いて書き換えてみよう。書き換え終わったら、実際に動作するか確かめてみよう。

2.7 rostopic

コマンド rostopic を用いると、ROS の中を流れるトピックを見ることができる。デバッグに便利なので覚えておこう。よく使うコマンドを紹介する。

rostopic list

流れているトピックのリストが見られる。

rostopic echo /topic_name

指定したトピックに流れているメッセージの中身が見られる。

rostopic

以上のコマンドを実行して、結果を確認しよう。

第 3 章　ROS 応用

ここまで、ROS を用いて最も基本的な Pub/Sub 通信を行うプログラムを作成した。ここでは、実際にロボットを製作できるように、さらに学習をすすめよう。

3.1　いろいろなメッセージ型を使ってみよう

これまでのプログラムでは、String 型の message をやり取りしてきた。このほかにも、ROS には標準のメッセージ型が多くある。その一覧は、公式 Wiki `https://wiki.ros.org/std_msgs` などで確認できる。また、ページ内のリンクからこれら各型の定義を見ることができる。

ここでは、Int16MultiArray を例に、その使用法を解説する。

3.1.1　Int16MultiArray の使用法

Int16MultiArray は、その名前から推測できるように、16bit 整数型の配列である。

単純な Publisher を紹介しよう。

ソースコード 3.1　publisher.cpp

```
1  #include "ros/ros.h"
2  #include "std_msgs/Int16MultiArray.h"
3
4  int main(int argc, char** argv)
5  {
6    ros::init(argc, argv, "talker0");
7    ros::NodeHandle nh;
8    ros::Publisher pub = nh.advertise<std_msgs::Int16MultiArray>("multi", 1000);
9
10   ros::Rate loop_rate(10);
11
12   while (ros::ok())
13   {
14     std_msgs::Int16MultiArray array;
15
16     array.data.resize(3);
17
18     for(int i = 0; i < 3; i++){
19       array.data[i]=i;
20     }
```

```
21
22      pub.publish(array);
23
24      ROS_INFO("Publishing");
25
26      ros::spinOnce();
27      loop_rate.sleep();
28    }
29  }
```

解説

ROS の標準メッセージ型の多くでは、data というメンバ変数にデータ本体が格納されている。C++の ROS プログラムにおいて、Int16MultiArray の data は、`std::vector<int16_t>`として解釈できる。[1)] 初期状態での array.data.size() は 0 なので、使う前に適切に resize() する必要がある。この点に注意すれば、通常のプログラムと同様に扱うことができる。もちろん、第 i 要素へは array.data[i] でアクセスできる。

Int16MultiArray を用いた Publisher

- 上記のコードを用いて Publisher を作ってみよう。
- Publisher を実行し、rostopic echo により topic を流れる内容を確認してみよう。

Int16MultiArray の応用課題

作成した Publisher から message を受け取り `ROS_INFO` で表示するプログラムを作成せよ。

3.1.2 std_msgs の使用に関する議論

`std_msgs` では、基本データ型や Empty 型・MultiArray など、便利な型が多く定義されている。しかし、これらは、“data” という名称のフィールドを持つのみで、その内容に関する意味をもたないから、プロトタイプには適しているものの、長期的 (long-term) な使用は意図されていない。一般的には、コードを分かりやすくするためには、汎用ではない（カスタム）メッセージを用いる方がよいとされている。

3.2 Arduino と通信しよう

多くの場合、ロボットには種々のマイコンが用いられる。これらのマイコンは、主にセンサやアクチュエータと接続されている。

1) その他の MultiArray は同様に対応する Vector 型。対応については `https://wiki.ros.org/msg` を参照せよ。

ここでは、ROS のシステムに我々がよく使う Arduino を組み込む方法として、rosserial を紹介する。

これを用いることで、Arduino1 台を 1 つのノードのように扱うことができる。詳しい仕組みの説明は省略するので、興味がある人は随時調べてみてほしい。

3.2.1 セットアップ

Arduino を扱う基本的な環境はすでに用意されているものとする。

```
1 sudo apt install ros-noetic-rosserial
2 sudo apt install ros-noetic-rosserial-arduino
```

rosserial の準備

以上のコマンドを実行して、rosserial をインストールしよう。

インストールが完了したら、Arduino 上で ROS に関する機能を用いるためのライブラリを生成させる。後述するように、生成したライブラリを Arduino のプログラムにおいて適切に読み込ませる必要がある。

```
1   cd <some_empty_directory>
2   rosrun rosserial_arduino make_libraries.py .
```

生成された `ros_lib` ディレクトリを Arduino の sketchbook/libraries フォルダにコピーする。

rosserial の準備

以上のコマンドを実行して、生成されたライブラリを Arduino のプログラムで読み込めるようにしよう。

3.2.2 Arduino 上の Publisher

Arduino を用い、“hello world!”” という文字列をメッセージとして流す Publisher を作ってみよう。

ソースコード 3.2 publisher.ino

```
1 #include <ros.h>
2 #include <std_msgs/String.h>
3
4 ros::NodeHandle nh;
5
6 std_msgs::String str_msg;
7 ros::Publisher chatter("chatter", &str_msg);
```

```
8
9  char hello[13] = "hello world!";
10
11 void setup()
12 {
13   nh.initNode();
14   nh.advertise(chatter);
15 }
16
17 void loop()
18 {
19   str_msg.data = hello;
20   chatter.publish( &str_msg );
21   nh.spinOnce();
22   delay(1000);
23 }
```

コードの詳しい解説は、https://wiki.ros.org/rosserial_arduino/Tutorials/Hello%20World にある。先日作ったプログラムとの類似点も多い。ここでは要点だけ述べる。

```
1  #include <ros.h>
2  #include <std_msgs/String.h>
```

ROS の Arduino プログラムを作る際は、ros.h 及び使用するメッセージのヘッダーファイル（この場合は String.h）を必ず読み込ませなければならない。

```
1 void loop()
2 {
3   str_msg.data = hello;
4   chatter.publish( &str_msg );
5   nh.spinOnce();
6   delay(1000);
7 }
```

Arduino Language では、setup() と loop() が存在しているのであった。これを活用し、loop() 内で spinOnce() する。

なお、master と接続できるまで spin しないようにするとより丁寧なプログラムになる。実用する際は、そのようにするべきだろう。

Publisher の作成

上記のコードを Arduino に書き込もう。

3.2.3 rosserial_arduino によるプログラムの実行

master は Ubuntu を走らせている PC となる。この PC に USB ケーブルで Arduino を接続する。

```
1  sudo chmod 777 /dev/ttyACM0
```

し、Arduino に読み書きできる権限を得た後、バッシュを 2 つ開き、それぞれ

```
1  rosscore
```

```
1  rosrun rosserial_arduino serial_node.py /dev/ttyACM0
```

を実行する。

「デバイスが見つからない」という類のエラーが出た際は、`ls /dev/tty*`からそれらしいものを探して、“/dev/ttyACM0” を書き換えよう。

Arduino プログラムの実行

- 以上に従い実行しよう。
- rostopic echo により topic を流れる内容を確認してみよう。

ここでは Publisher として動かしたが、もちろん、Arduino を Subscriber とすることもできる。

3.2.4 Arudino 上の Subscriber

ソースコード 3.3 subscriber.ino

```
1  #include <ros.h>
2  #include <std_msgs/Empty.h>
3
4  ros::NodeHandle nh;
5
6  void messageCb( const std_msgs::Empty& toggle_msg){
7    digitalWrite(13, HIGH-digitalRead(13));   // blink the led
8  }
9
10 ros::Subscriber<std_msgs::Empty> sub("toggle_led", &messageCb );
11
12 void setup()
13 {
14   pinMode(13, OUTPUT);
15   nh.initNode();
16   nh.subscribe(sub);
17 }
```

```
18
19 void loop()
20 {
21   nh.spinOnce();
22   delay(1);
23 }
```

このプログラムは、Empty 型のメッセージを受け取るごとに Arduino の LED を点滅させるプログラムである。

Blink プログラムの実行

- 以上のプログラムを書き込み、実行しよう。
- `rostopic pub /toggle_led std_msgs/Empty` を実行してみよう。

3.2.5 MultiArray

Arduino で multiarray を取り扱うにはややトリッキーな操作が必要であるので補足しておく。たとえば、`std_msgs::Int32MultiArray vec;` をサイズ 6 で使うときは、setup において、

```
1   vec.data = (int32_t*)malloc(sizeof(int32_t) * 6);
2   vec.data_length = 6;
```

する必要がある。背景の説明は省略するので、おまじないか何かだと思っておいてほしい。

3.2.6 まとめ

以上、本節においては、Arduino を用いた ROS プログラムの記述について概観した。

最終課題

次のような要件のもと、プログラムを作成せよ。

- 1 つの PC と 1 つの Ardunio Mega を USB ケーブルで繋ぎ、rosserial により通信することとする。
- Arduino Mega には、超音波センサを 2 つ接続する。ポート番号は適当に定めてよい。
- それぞれの超音波センサから読み取られた物体までの距離を、PC 上で走る C++で書かれたノードから、`ROS_INFO` を用いて出力する。

ここまでの例においては、1 つのノードは 1 つの役割しか持たなかった。しかし、1 つのノードを Publisher かつ Subscriber として動作させることも可能である。実際のロボット製作では、そのように、複数の topic が 1 つのノードにかかわることが一般的だ。

発展課題

次のような要件のもと、プログラムを作成せよ。

- 1 つの PC と 1 つの Ardunio Mega を USB ケーブルで繋ぎ、rosserial により通信することとする。
- Arduino Mega には、フォトリフレクタを 1 つ接続する。ポート番号は適当に定めてよい。
- フォトリフレクタによって読み込まれた値を、PC 上で走る C++で書かれたノードから、`ROS_INFO` を用いて出力する。
- さらに、フォトリフレクタの値が「黒」と判定できる間、Arduino の LED を点灯させる。
- ただし、黒かどうかの判定は PC 上で行うものとする。

第 4 章　おまけ

4.1　個人的オススメソフト

Visual Studio Code

Microsoft 謹製のテキストエディタ[1)]。拡張機能を入れることで、自由自在に使うことができる。Windows 端末から Linux に SSH 接続して、Windows 側から開発することもできる。

ちなみに、Arduino 拡張をインストールすれば、Arduino IDE を使わずとも Arduino 開発ができる。Arduino IDE は正直使いづらい[2)]ので、VS Code を使うことを推奨する。

4.2　よく使うもの

ロボット製作でよく用いるものの分量の都合上取り扱えなかった要素を簡潔に紹介する。

4.2.1　コントローラの接続

ロボットを操作するのにコントローラを用いることが多い。パッケージを用いることで、コントローラを ROS に接続し、一つのノードのように取り扱うことができる。

4.2.2　STM32

今回は、マイコンとして Arduino を用いた。他にも広く使われているマイコンとして、STM32 がある。これについても同様に、ROS で取り扱うことができる。

4.2.3　roslaunch

本資料においては、ROS の master や各ノードの実行をするごとに、コマンドを入力していた。しかし、一般的に、あるロボットを動かすときに入力するコマンドはそう変わるものでない。Launch ファイルを記述し `roslaunch` コマンドにより用いると、master やノード[3)]を一度に実行できる。

---

1) テキストファイルの編集に用いるソフトウェア
2) Arduino IDE 2.0 は相当改善されているようなので、そちらを使うのでもよいが
3) もちろん Arduino やコントローラも OK

4.2.4 rosparam

ロボットには、様々なパラメータがある。それらをソースコードにそのまま定数として書き込むと、パラメータ調整やハードの変更があるごとにソフトをビルドし直す羽目になる。rosparam は、そのような起動時に定まるパラメータ[4)]を管理する、サーバーのようなものである。

`rosparam set` コマンドを用いたり、yaml ファイルに値を書き込んでおいてそれをロードしたりすることで、サーバーにパラメータを登録できる。そして、その値を ROS の各ノード[5)]や `rosparam get` コマンドから読み取ることができる。これを用いると、ソフトをビルドし直すことなく、パラメータを変更することができる。roslaunch で rosparam を設定することもできる。

4.3 公式サイトの紹介

Linux や ROS・Arduino に関する情報はインターネット上に数多く転がっているが、その質は玉石混淆である。それらのサイトを使うことは否定しないが、そこの情報を使っても上手くいかないとき、公式ドキュメントに立ち戻ると、思わぬ答えが書いてあることも多い。もちろん、最初から公式ドキュメントにあたることもよいだろう。（ややとっつきにくさはあるが。）

ここに、今回用いたものの公式サイトの一覧を紹介しておく。

- Bash `https://www.gnu.org/software/bash/manual/bash.html`
- ROS `https://wiki.ros.org/`
- catkin_tools `https://catkin-tools.readthedocs.io/en/latest/`
- Arduino `https://www.arduino.cc/en/Guide`

4) 動的に変化するパラメータも取り扱おうと思えば取り扱えるが、一般的ではない
5) もちろん Arduino でも用いれる