

ROSの基本操作

Updated on: 2019-11-12

[トップページ](#)

- [基本的な用語](#)
- [ソースコードを格納するワークスペース](#)
- [ROSノードの理解とビルド・実行](#)
 - [送信ノードの作成](#)（基本的なコードを読み解く）
 - [ビルド&実行](#)
 - [受信ノードの作成](#)
 - [ビルド&実行](#)
- [システムとして実行する](#)
 - [launchファイルを読み解く](#)
 - [roslaunchでシステムを起動](#)
- [ROSノードの作成](#)
 - [パッケージの作成](#)
 - [ノードを作成](#)
 - [CMakeLists.txtにノードを追加](#)
 - [ノードのソースの作成](#)
 - [ビルド&実行](#)
- [サーボの状態を確認](#)
 - [ノードを作成](#)
 - [CMakeLists.txtにノードを追加](#)
 - [ノードのソースの作成](#)
 - [ビルド&実行](#)
- [課題](#)

基本的なROS上で動くプログラムの書き方とコンパイル方法を学習します。

基本的な用語

パッケージ

ノードや設定ファイル、コンパイル方法などをまとめたもの

ノード

ROSの枠組みを利用して動作する実行ファイル

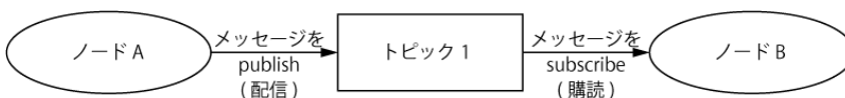
メッセージ

ノード間でやりとりするデータ

トピック

ノード間でメッセージをやりとりする際にメッセージを格納する場所

ノード、メッセージ、トピックの関係は以下の図のように表せます。



基本的に、ソフトウェアとしてのROSはノード間のデータのやりとりをサポートするための枠組みです。

加えて、使い回しがきく汎用的なノードを世界中のROS利用者と共有するコミュニティも大きな意味でのROSの一部となっています。

ソースコードを格納するワークスペース

ROSでは、プログラムをビルドする際にcatkinというソフトウェアパッケージを使用しています。

また、catkin は、cmake というソフトウェアを使っており、ROS用のプログラムのパッケージ毎にcmakeの設定ファイルを作成することで、ビルドに必要な設定を行います。

以下の手順で本作業用の新しいワークスペースを作ります。

```
1  $ mkdir -p ~/catkin_ws/src
2  $ cd ~/catkin_ws/src
3  $ catkin_init_workspace
4  Creating symlink "/home/[ユーザ名]/catkin/src/CMakeLists.txt"
5    pointing to "/opt/ros/kinetic/share/catkin/cmake/CMakeLists.txt"
6  $ ls
7  CMakeLists.txt
8  $ cd ..
9  $ ls
10 src
11 $
```

catkin_wsディレクトリ内にある、build、develは、catkinシステムがプログラムをビルドする際に使用するものなので、ユーザが触る必要はありません。

catkin_ws/srcディレクトリは、ROSパッケージのソースコードを置く場所で、中にあるCMakeLists.txt は、ワークスペース全体をビルドするためのルールが書かれているファイルです。

このディレクトリに、本作業用のパッケージをダウンロードします。

```
1  $ cd ~/catkin_ws/src
2  $ git clone https://github.com/gbiggs/rsj_tutorial_2017_ros_intro
3  $ ls
4  CMakeLists.txt  rsj_tutorial_2017_ros_intro
5  $
```

gitは、ソースコードなどの変更履歴を記録して管理する分散型バージョン管理システムと呼ばれるものです。

今回のセミナーでは詳細は触れませんが、研究開発を行う上では非常に有用なシステムですので、利用をお勧めします。

公式の解説書、[Pro Git](#)などを参考にして下さい。

GitHubは、ソースコードなどを格納、保管、管理するためのWebリポジトリサービスです。

オープンソースソフトウェアの開発、共同作業および配布のためによく利用されており、ROSではソースコードの保存と配布する場所としてもっとも人気なサービスとなっています。

バイナリパッケージとして配布されているROSパッケージ以外の利用をする場

合、GitHubを利用します。

URLが分かれば上の手順だけで簡単にROSのパッケージが自分のワークスペースにインポートし利用することができます。

では、次にパッケージのディレクトリ構成を確認します。

ダウンロードしているパッケージがバージョンアップされている場合などには、下記の実行例とファイル名が異なったり、ファイルが追加・削除されている場合があります。

```
1  $ cd ~/catkin_ws/src/rsj_tutorial_2017_ros_intro/
2  $ ls
3  CMakeLists.txt  launch  msg  package.xml  src
4  $ ls launch/
5  say_hello.launch
6  $ ls msg/
7  Greeting.msg
8  $ ls src/
9  displayer.cpp  greeter.cpp
10 $
```

CMakeLists.txtとpackage.xmlには、使っているライブラリの一覧や、生成する実行ファイルとC++のソースコードの対応など、このパッケージをビルドするために必要な情報が書かれています。

launchディレクトリには、複数のノードでできたシステムの定義が、msgディレクトリには、このパッケージ独自のデータ形式の定義が、srcディレクトリには、このパッケージに含まれるプログラム(ノード)のソースコードが含まれています。

catkin_makeコマンドで、ダウンとロードしたrsj_tutorial_2017_ros_introパッケージを含む、ワークスペース全体をビルドします。catkin_makeは、ワークスペースの最上位ディレクトリ(~/catkin_ws/)で行います。

ROSノードの理解とビルド・実行

先作成したワークスペースを利用します。

ターミナルを開き、パッケージが正しく存在しているか確認します。

```
1  $ cd ~/catkin_ws/src/
2  $ ls
3  CMakeLists.txt  rsj_tutorial_2017_ros_intro
4  $ cd ..
5  $
```

ソースファイルの編集にはお好みのテキストエディタが利用可能です。

Linuxでのプログラム開発がはじめての方には、Ubuntuにデフォルトでインストールされているgeditがおすすめです。

お好みのテキストエディタで

~/catkin_ws/src/rsj_tutorial_2017_ros_intro/src/greeter.cpp
を開きます。

```

s/greeter.cpp
3 #include <ros/ros.h>
4 #include <rsj_tutorial_2017_ros_basics/Greeting.h>
5
6 #include <string>
7
8 int main(int argc, char **argv) {
9     ros::init(argc, argv, "Greeter");
10    ros::NodeHandle node;
11
12    std::string hello_text;
13    std::string world_name;
14    ros::param::param<std::string>("~hello_text", hello_text, "hello");
15    ros::param::param<std::string>("~world_name", world_name, "world");
16
17    ros::Publisher pub = node.advertise<rsj_tutorial_2017_ros_basics::Greeting>("greeting", 1);
18
19    ros::Rate rate(1);
20
21    while (ros::ok()) {
22        rsj_tutorial_2017_ros_basics::Greeting greeting;
23        greeting.hello_text = hello_text;
24        greeting.world_name = world_name;
25        pub.publish(greeting);
26
27        rate.sleep();
28    }
29
30    return 0;
31 }

```

送信ノードの作成（基本的なコードを読み解く）

このコードが実行されたときの流れを確認しましょう。

まず、先頭部分では、必要なヘッダファイルをインクルードしています。

```
1 #include <ros/ros.h>
```

続いて、本ノードが利用するメッセージのヘッダファイルをインクルードしています。

```
1 #include <rsj_tutorial_2017_ros_basics/Greeting.h>
```

std::stringが利用されるので、ヘッダファイルをインクルードします。

```
1 #include <string>
```

続いて、C++のmain関数が定義されています。

本ノードは非常に簡単な構成としているため、すべての機能をmain関数に入れています。

複雑な機能や色々なデータを持つノードには、クラスとしての実装することをおすすめします。

```

1 int main(int argc, char **argv) {
2     ros::init(argc, argv, "Greeter");
3     ros::NodeHandle node;
4
5     std::string hello_text;
6     std::string world_name;
7     ros::param::param<std::string>("~hello_text",
8     ros::param::param<std::string>("~world_name",
9
10    ros::Publisher pub = node.advertise<rsj_tutorial_2017_ros_basics::Greeting>("greeting", 1);
11
12    ros::Rate rate(1);

```

```

13
14     while (ros::ok()) {
15         ros::spinOnce();
16
17         ROS_INFO("Publishing greeting '%s %s'", hello_text, world_name);
18         rsj_tutorial_2017_ros_basics::Greeting greeting;
19         greeting.hello_text = hello_text;
20         greeting.world_name = world_name;
21         pub.publish(greeting);
22
23         rate.sleep();
24     }
25
26     return 0;
27 }

```

main関数は、まずノードのセットアップを行います。

`ros::init`はROSのインフラストラクチャの初期設定を行いノードを初期化します。

1、2番目の引数には、main関数の引数をそのまま渡し、3番目の引数には、このノードの名前(この例では"Greeter")を与えています。

その次にある`ros::NodeHandle node`は、ノードを操るための変数を初期化します。

次の4行はパラメータの初期化です。ROSでは、純粋コマンドラインを利用するよりROSのパラメータ機能を利用することが標準的です。

こうすることで、コマンドラインだけではなく、roslaunch（複数のノードを起動するためのツール）やGUIツールからもパラメータの設定が簡単にできます。

パラメータの初期化が終わったら、データ送信のためのパブリッシャーを初期化します。

この変数の作成によりトピックが作成され、**このノードからデータの送信が可能になります。**

以下の引数を与えています。

"greeting"

トピック名：データをこのトピックに送信する

1

メッセージのバッファリング量を指定 (大きくすると、処理が一時的に重くなったときなどに受け取り側の読み飛ばしを減らせる)

`advertise`関数についている、

`<rsj_tutorial_2017_ros_basics::Greeting>`の部分は、メッセージの型を指定しています。

これは、幾何学的・運動学的な値を扱うメッセージを定義している

`rsj_tutorial_2017_ros_basics`パッケージの、並進・回転速度を表す `Greeting`型です。

(この指定方法は、C++のテンプレートという機能を利用していますが、ここでは「`advertise`のときはメッセージの型指定を<>の中に書く」とだけ覚えておけば問題ありません)

セットアップの最後として、`ros::Rate rate(1)`で周期実行のためのクラスを初期化しています。

初期化時の引数で実行周波数(この例では1 Hz)を指定します。

`while(ros::ok())`で、メインの無限ループを回します（すなわちこのノードのメインプロセッシングループです）。

`ros::ok()`を`while`の条件にすることで、ノードの終了指示が与えられたとき(**Ctrl+C**が押された場合も含む)には、ループを抜けて終了処理などが行えるようになっています。

ループ中では、まず、*`ros::spinOnce()`を呼び出して、ROSのメッセージを受け取る* といった処理を行います。

`spinOnce`は、その時点で届いているメッセージの受け取り処理を済ませた後、すぐに処理を返します。

`rate.sleep()`は、先ほど初期化した実行周波数を維持するように`sleep`します。

`ros::spinOnce()`と`rate.sleep()`の間に本ノードの処理を入れました。

最初は、ROSでログ情報を画面などに出力する際に用いる`ROS_INFO()`関数を呼び出してメッセージを表示しています。

他にも、`ROS_DEBUG()`、`ROS_WARN()`、`ROS_ERROR()`、`ROS_FATAL()`などのデバッグログ関数が用意されています。

その後、データを送信します。

まずは送信するデータ型

(`rsj_tutorial_2017_ros_basics::Greeting`)を初期化し、値を設定します。

さきほどセットアップで作成したパラメータの値を利用します。

こうすることで、送信されるデータの内容は実行するときに自由に変更できます。

そして、`pub.publish(greeting)`によってデータを送信します。

この行でデータはバッファーに入れられ、別のスレッドが自動的にサブスクライバに送信します(ROSのデフォルトでは非同期送信となります)。

main ループが終了すると作成した変数は自動的にクリーンアップを実行し、ノードのシャットダウンを行います。

ビルド&実行

ROS パッケージをビルドするためには、`catkin_make`コマンドを用います。

下記コマンドをターミナルで実行してみましょう。

```
1 $ cd ~/catkin_ws/  
2 $ catkin_make
```

ROSシステムの実行の際、ROSを通してノード同士がデータをやりとりするために用いる「roscore」を起動しておく必要があります。

2つ目のターミナルを開き、それぞれで以下を実行して下さい。

1つ目のターミナルで下記を実行します。

```
1 $ roscore
```

ROSでワークスペースを利用するとき、ターミナルでそのワークスペースに環境変数などを設定することが必要です。

このためにワークスペースの最上位のディレクトリで`source devel/setup.bash`を実行します。

このコマンドはワークスペースの環境編情報などを利用中のターミナルに読み込みます。

しかし、ターミナルごとに環境変数はリセットされますので、新しいターミナルでワークスペースを利用しはじめるときには、まず`source`

`devel/setup.bash`を実行しなければなりません。

一つのターミナルで一回だけ実行すれば十分です。そのターミナルを閉じるまで有効となります。

2つ目のターミナルで下記を実行します。

```
1 $ cd ~/catkin_ws/
2 $ source devel/setup.bash
3 $ rosrn rsj_tutorial_2017_ros_basics greeter
4 [ INFO] [1494840089.900580884]: Publishing greeting
```

上記のようなログが表示されれば成功です。

ソースコードにパラメータを利用したので、コマンドラインからパラメータ設定を試してみましょう。

ノードを実行した2つ目のターミナル（**注意：roscoreのターミナルではなくて**）に`Ctrl+c`を入力してノードを終了します。

そして以下を実行してください。

```
1 $ rosrn rsj_tutorial_2017_ros_basics greeter _he
2 [ INFO] [1494840247.644756809]: Publishing greeting
```

上記のようなログが表示されれば成功です。

実行後は両方のターミナルで`Ctrl+c`でノードとroscoreを終了します。

受信ノードの作成

前節で作成したノードはメッセージを送信するノードでした。

次にメッセージを受信するノードを作成してみましょう。

以下のソースは`rsj_tutorial_2017_ros_basics/src/displayer.cpp`ファイルにあります。

```
1 #include <ros/ros.h>
2 #include <rsj_tutorial_2017_ros_basics/Greeting.h>
3
4 #include <iostream>
5
6 void callback(const rsj_tutorial_2017_ros_basics::Greeting& msg) {
7     std::cout << msg->hello_text << " " << msg->world << "\n";
8 }
9 int main(int argc, char **argv) {
```

```

10     ros::init(argc, argv, "Displayer");
11     ros::NodeHandle node;
12
13     ros::Subscriber sub = node.subscribe("greeting
14
15     ros::spin();
16
17     return 0;
18 }

```

本ノードはgreetingというトピックから取得したデータをターミナルに表示します。

greeterからの差は以下のようです。

まずはcallback関数です。

この関数はトピックのデータ型に合っているポインタを引数としてもらいます。

トピックからデータを受け取ったら、callback関数は呼ばれます。

そのデータをstd::coutに出力し終了します。

const rsj_tutorial_2017_ros_basics::Greeting::ConstPtr は、const型(内容を書き換えられない)、rsj_tutorial_2017_ros_basicsパッケージに含まれる、Greeting型のメッセージの、const型ポインタを表しています。

&msgの&は、参照型(内容を書き換えられるように変数を渡すことができる)という意味ですが、(const型なので)ここでは特に気にする必要はありません。msgはクラスへのポインタなので「->」を用い、以降はクラスのメンバ変数へのアクセスなので「.」を用いてアクセスしています。

main関数内にパラメータの初期化はなくなりました。本ノードはパラメータを利用しません。

ros::Publisherの初期化もなくなり、代わりにros::Subscriberを初期化します。

この行はトピックへのアクセスを初期化し、データが取得したらの対応を示します。

引数は以下です。

"greeting"

トピック名

10

バッファサイズ

callback

メッセージを受け取ったときに呼び出す関数を指定 (callback関数)

最後に、main ループでの処理は不要となります。

本ノードはデータが届くとき以外何もしないので、無限ループになる

ros::spin()を呼びます。

ros::spin()はgreeterのwhile(...)とros::spinOnce()と類似の機能の中で持つので、ノードがシャットダウンされるまでに戻りません。

ビルド&実行

作成したノードを実行してみましょう。

1つ目のターミナルで以下を実行します。

1

\$ roscore

そして2つ目のターミナルで以下を実行します。

```
1 $ cd ~/catkin_ws/
2 $ source devel/setup.bash
3 $ rosrun rsj_tutorial_2017_ros_basics greeter
4 [ INFO] [1494840089.900580884]: Publishing greeting
```

最後に、3番目のターミナルを開いて、下記を実行します。

```
1 $ cd ~/catkin_ws/
2 $ source devel/setup.bash
3 $ rosrun rsj_tutorial_2017_ros_basics displayer
```

「hello world」と表示されれば成功です。

以上の手順で、ROSパッケージに含まれるノードのソースコードを編集し、ビルドして、実行できるようになりました。

システムとして実行する

roslaunchを利用して、複数のノードでできたシステムを開始、終了することができます。

手動でノードを1ノードずつ起動することは面倒ですし、ミスを誘発する可能性も高まります。

そのためにROSにroslaunchというツールがあります。

roslaunchを利用すると、システム全体をまとめて起動し、状況をモニターし、そしてまとめて終了することが可能となります。

launchファイルを読み解く

launchファイルは、ノードやパラメータの組み合わせを定義するためのファイルです。

フォーマットはXMLです。

システムに含まれるノードとそのノードの起動方法を一つずつ定義します。

rsj_tutorial_2017_ros_basicsパッケージに以下のファイルがlaunch/say_hello.launchとして存在します。

このファイルはさきほど手動で起動したシステムを定義します。

```
1 <launch>
2   <node name="greeter" pkg="rsj_tutorial_2017_ros_basics" exec="greeter" />
3     <param name="hello_text" value="allo"/>
4     <param name="world_name" value="earth"/>
5   </node>
6
```

```

7   <node name="displayer" pkg="rsj_tutorial_2017_ros_basics" type="displayer" output="screen"/>
8 </launch>

```

nodeタグは2つあります。

属性は下記の通りです。

name

ノードインスタンスの名

pkg

ノードを定義するパッケージ名

type

ノードの実行ファイル名

output

stdoutの先：定義しないとstdout（ROS_INFOやstd::coutへの出力等）はターミナルで表示されず、~/ros/log/に保存されるログファイルだけに出力される。

1番目の<node>はgreeterノードの定義です。

本要素の中でパラメータの設定も行っています。

なお、パラメータの設定を行わない場合はノードのソースに定義したデフォルト値が利用されるので、記述は必須ではありません。

2番目の<node>はdisplayerノードの定義です。

パラメータはありませんが、出力されることをターミナルで表示するようにします。

roslaunchでシステムを起動

開いているターミナルにroscoreや起動中のノードをすべて **Ctrl+c** で停止します。

その後、いずれか1つのターミナルで以下を実行します。

```

1  $ cd ~/catkin_ws
2  $ source devel/setup.bash
3  $ roslaunch rsj_tutorial_2017_ros_basics say_hello.launch
4  ... logging to /home/geoff/.ros/log/40887b56-3956-11e9-80d0-080027121483
5  Checking log directory for disk usage. This may
6  Press Ctrl-C to interrupt
7  Done checking log file disk usage. Usage is <1GB
8
9  started roslaunch server http://alnilam:40672/
10
11  SUMMARY
12  =====
13
14  PARAMETERS
15  * /greeter/hello_text: allo
16  * /greeter/world_name: earth
17  * /roscpp__core__name: /roscpp__core__name
18  * /rosversion: 1.12.7
19
20  NODES
21  /
22  displayer (rsj_tutorial_2017_ros_basics/displayer)
23  greeter (rsj_tutorial_2017_ros_basics/greeter)

```

```

24
25 auto-starting new master
26 process[master]: started with pid [11098]
27 ROS_MASTER_URI=http://localhost:11311
28
29 setting /run_id to 40887b56-395c-11e7-b868-d8cb8
30 process[rosout-1]: started with pid [11111]
31 started core service [/rosout]
32 process[greeter-2]: started with pid [11118]
33 process[displayer-3]: started with pid [11126]
34 allo earth
35 allo earth

```

「allo earth」が繰り返して表示されたら成功です。

Ctrl+c でシステムを停止します。

```

1 allo earth
2 allo earth
3 allo earth
4 [Ctrl+c]
5 ^C[displayer-3] killing on exit
6 [greeter-2] killing on exit
7 [rosout-1] killing on exit
8 [master] killing on exit
9 shutting down processing monitor...
10 ... shutting down processing monitor complete
11 done
12 $

```

これでシステムの起動、停止が簡単にできるようになりました。

roslaunchを利用する場合は、別のターミナルでのroscoreの実行は不要です。

roslaunchは中でroscoreを起動したり停止したりします。

ROSノードの作成

パッケージとノードを自分で作成しマニピュレータのサーボモータを操作します。

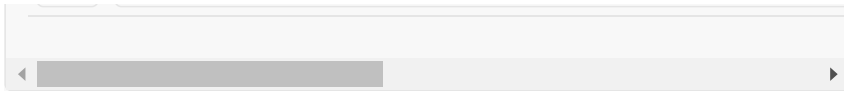
パッケージの作成

ワークスペースに新しいパッケージを作成するために、以下を実行してください。

```

1 $ cd ~/catkin_ws/src
2 $ catkin_create_pkg rsj_2017_servo_control roscpp
3 Created file rsj_2017_servo_control/CMakeLists.txt
4 Created file rsj_2017_servo_control/package.xml
5 Created folder rsj_2017_servo_control/include/rsj_
6 Created folder rsj_2017_servo_control/src
7 Successfully created files in /home/geoff/catkin_

```



引数の1番目はパッケージ名です。

2番目と3番目は依存パッケージの定義です。

今回はC++で記述するため、roscppに依存します。

そしてDynamixelのサーボを利用するのでハードウェアとインターフェースするdynamixel_controllersに依存します。

また、サーボコントローラにコマンドを送るためにDynamixel用のメッセージタイプの利用が必要なのでdynamixel_msgsにも依存します。

生成されたパッケージの中身を確認します。

```
1 $ cd rsj_2017_servo_control/  
2 $ ls  
3 CMakeLists.txt  include  package.xml  src
```

以前と同様に、package.xmlはパッケージの情報を定義し、CMakeLists.txtはパッケージのビルド方法を定義します。

package.xmlをエディターで開くと以下の行が含まれていると見えます。

```
1 <buildtool_depend>catkin</buildtool_depend>  
2 <build_depend>dynamixel_controllers</build_depend>  
3 <build_depend>dynamixel_msgs</build_depend>  
4 <build_depend>roscpp</build_depend>  
5 <run_depend>dynamixel_controllers</run_depend>  
6 <run_depend>dynamixel_msgs</run_depend>  
7 <run_depend>roscpp</run_depend>
```

これらはパッケージの依存を定義します。ROSとcatkinはこれらの利用によってパッケージのビルド順番等を決めます。

普段はpackage.xmlのメール等(<maintainer email>タグ等)も編集するべきですが、今回は時間のために省略します。

以上、パッケージの作成ができました。

ノードを作成

作成したパッケージにノードを作成します。

ノードを作成するために以下の手順を行います。

1. CMakeLists.txtにノードのコンパイル方法を追加する
2. ノードのソースファイルを作成する

CMakeLists.txtにノードを追加

rsj_2017_servo_controlパッケージにある

CMakeLists.txt (~/.catkin_ws/src/rsj_2017_servo_control/CMakeLists.txt) をエディタで開き、以下のようにソースを編集します。

5行目、12行目、17行目および20行目を追加しました。

ファイル内の133行目ぐらいから始まります。

インストールされたcatkinのバージョンにより、編集する具体的な行番号が変化する可能性があります。

```

1  ## Declare a C++ executable
2  ## With catkin_make all packages are built withi
3  ## The recommended prefix ensures that target na
4  # add_executable(${PROJECT_NAME}_node src/rsj_20
5  add_executable(${PROJECT_NAME}_set_servo_pos src,
6
7  ## Rename C++ executable without prefix
8  ## The above recommended prefix causes long targ
9  ## target back to the shorter version for ease o
10 ## e.g. "roslaunch someones_pkg node" instead of "r
11 # set_target_properties(${PROJECT_NAME}_node PRO
12 set_target_properties(${PROJECT_NAME}_set_servo_
13
14 ## Add cmake target dependencies of the executab
15 ## same as for the library above
16 # add_dependencies(${PROJECT_NAME}_node ${${PROJECT
17 add_dependencies(${PROJECT_NAME}_set_servo_pos $
18
19 ## Specify libraries to link a library or execut
20 target_link_libraries(${PROJECT_NAME}_set_servo_

```

なお、**必ず** ファイルトップにある`add_compile_options(-std=c++11)`の行をコメントアウトしてください。

これでcatkinに`set_servo_pos`というノードのコンパイルを指定できました。

ノードのソースの作成

rsj_2017_servo_controlパッケージ内のsrc/ディレクトリに

`set_servo_pos.cpp`というファイル

(`~/catkin_ws/src/rsj_2017_servo_control/src/set_servo_pos.cpp`)を作成します。

そしてエディタで`set_servo_pos.cpp`を開き、以下のソースを入力します。

```

1  #include <ros/ros.h>
2  #include <std_msgs/Float64.h>
3
4  #include <string>
5  #include <vector>
6
7  int main(int argc, char **argv) {
8      ros::init(argc, argv, "set_servo_pos");
9      ros::NodeHandle node;
10
11      std::string servo_command_topic;
12      ros::param::param<std::string>(
13          "~servo_command_topic",
14          servo_command_topic,

```

```

15     "/finger_servo_controller/command");
16
17     ros::Publisher pub = node.advertise<std_msgs::
18
19     std::vector<std::string> my_args;
20     ros::removeROSArgs(argc, argv, my_args);
21     if (my_args.size() < 2) {
22         ROS_FATAL("Usage: rosrund set_servo_pos [posi
23         ros::shutdown();
24         return 1;
25     }
26     std_msgs::Float64 servo_pos;
27     servo_pos.data = std::stof(my_args[1]);
28
29     ROS_INFO("Setting servo position to %f", servo.
30     ros::Rate r(1);
31     while (ros::ok()) {
32         pub.publish(servo_pos);
33         ros::spinOnce();
34         r.sleep();
35     }
36
37     return 0;
38 }

```

このソースはコマンドラインから指定のサーボモータの位置を読み、パラメータで指定されたトピック（デフォルトでは/finger_servo_controller/command）にパブリッシュします。このトピックはサーボコントローラがサブスクライブしています。

なぜサーボの位置はパラメータとしてノードに渡さなかったのでしょうか。パラメータはいわゆる「configuration」です。ノードの振る舞いを制御するためのことです。サーボのトピックはノードの振る舞いですが、サーボの位置はコマンドです。このノードはコマンドラインで利用すべきなツールなので普通のコマンドラインパラメータを利用しました。

ビルド&実行

パッケージ内のソースを変更・追加した後、必ず再ビルドが必要です。ターミナルでcatkin_makeコマンドを実行して再ビルドします。

1つ目のターミナル：

```

1  $ cd ~/catkin_ws/
2  $ catkin_make

```

この際、**ビルドエラー**が出力されていないかよく確認して下さい。エラーが出ている場合は、ソースコードの該当箇所を確認・修正して下さい。

実行する前にまずはマニピュレータが壊れないようにします。マニピュレータのグリッパーが動くので、**電源を入れる前にマニピュレータのグリッパーは何にもぶつからないような姿勢にしましょう**。

マニピュレータのサーボコントローラを起動することが必要です。

rsj_2017_servo_controlパッケージの中に以下のファイルを作成します。

~/catkin_ws/src/rsj_2017_servo_control/configと

~/catkin_ws/src/rsj_2017_servo_control/launchディレクトリは作成した上でファイルを編集します。

```
1 $ cd ~/catkin_ws/src/rsj_2017_servo_control
2 $ mkdir config
3 $ mkdir launch
```

~/catkin_ws/src/rsj_2017_servo_control/config/dynamixel_test.yaml

:

```
1 finger_servo_controller:
2   controller:
3     package: dynamixel_controllers
4     module: joint_position_controller
5     type: JointPositionController
6   joint_name: finger_joint
7   joint_speed: 1.17
8   motor:
9     id: 5
10    init: 512
11    min: 0
12    max: 1023
```

~/catkin_ws/src/rsj_2017_servo_control/launch/dynamixel_test.launch

:

```
1 <launch>
2   <node name="dynamixel_manager" pkg="dynamixel_controllers"
3     type="controller_manager.py" required="true">
4     <rosparam>
5       namespace: dynamixel_controller_manager
6       serial_ports:
7         dxl_tty1:
8           port_name: "/dev/ttyUSB0"
9           baud_rate: 1000000
10          min_motor_id: 1
11          max_motor_id: 5
12          update_rate: 10
13     </rosparam>
14   </node>
15   <rosparam file="$(find rsj_2017_servo_control)/config/dynamixel_test.yaml"
16   <node name="finger_servo_spawner" pkg="dynamixel_controllers"
17     args="--manager=dynamixel_controller_manager --port dxl_tty1
18           finger_servo_controller"
19     output="screen"/>
20 </launch>
```

1つ目のターミナルで以下を実行し、マニピュレータのグリッパーサーボコントローラを起動します。

```

1  $ roslaunch rsj_2017_servo_control dynamixel_test
2  ... logging to /home/geoff/.ros/log/619c447c-396
3  checking log directory for disk usage. This may
4  Press Ctrl-C to interrupt
5  Done checking log file disk usage. Usage is <1GB
6
7  started roslaunch server http://alnilam:44912/
8
9  SUMMARY
10  =====
11  (省略)

```

2つ目のターミナルで以下のコマンドを実行します。

```

1  $ rosrund rsj_2017_servo_control set_servo_pos 0
2  [ INFO] [1494851539.189274395]: Setting servo pos
3  [Ctrl+cで止める]
4  $ rosrund rsj_2017_servo_control set_servo_pos -0.
5  [ INFO] [1494851548.085785357]: Setting servo pos
6  [Ctrl+cで止める]

```

サーボモータが動けば、サーボモータ制御は成功しています。

このソースは以下のURLでダウンロード可能です。

https://github.com/gbiggs/rsj_2017_servo_control

サーボの状態を確認

もう一つのノードを作成し、サーボモータの現在状況をターミナルで表示します。

上記と同じ手順でservo_statusというノードをrsj_2017_servo_controlパッケージに追加します。

ノードを作成

CMakeLists.txtにノードを追加

~/catkin_ws/src/rsj_2017_servo_control/CMakeLists.txtを編集して、新しいノードのコンパイル方法を指定します。

ファイルをエディターで開き以下の通りになるように編集します。

(6行目、14行目、20行目および24行目を追加しました。)

```

1  ## Declare a C++ executable
2  ## With catkin_make all packages are built withi
3  ## The recommended prefix ensures that target na

```



```

4   # add_executable(${PROJECT_NAME}_node src/rsj_20
5   add_executable(${PROJECT_NAME}_set_servo_pos src/
6   add_executable(${PROJECT_NAME}_servo_status src/
7
8   ## Rename C++ executable without prefix
9   ## The above recommended prefix causes long targ
10  ## target back to the shorter version for ease o
11  ## e.g. "roslaunch someones_pkg node" instead of "r
12  # set_target_properties(${PROJECT_NAME}_node PRO
13  set_target_properties(${PROJECT_NAME}_set_servo_
14  set_target_properties(${PROJECT_NAME}_servo_stat
15
16  ## Add cmake target dependencies of the executab
17  ## same as for the library above
18  # add_dependencies(${PROJECT_NAME}_node ${${PROJECT
19  add_dependencies(${PROJECT_NAME}_set_servo_pos $
20  add_dependencies(${PROJECT_NAME}_servo_status ${
21
22  ## Specify libraries to link a library or execut
23  target_link_libraries(${PROJECT_NAME}_set_servo_
24  target_link_libraries(${PROJECT_NAME}_servo_stat

```

ノードのソースの作成

rsj_2017_servo_controlパッケージ内のsrc/ディレクトリに
servo_status.cppというファイルを作成します。

エディター

~/catkin_ws/src/rsj_2017_servo_control/src/servo_status.cpp
を新規作成し、以下のソースを入力します。

```

1   #include <ros/ros.h>
2   #include <dynamixel_msgs/JointState.h>
3
4   #include <string>
5
6   void callback(const dynamixel_msgs::JointState::
7       ROS_INFO("--- Servo status ---");
8       ROS_INFO("Name: %s", msg->name.c_str());
9       ROS_INFO("ID: %d", msg->motor_ids[0]);
10      ROS_INFO("Temperature: %d", msg->motor_temps[0
11      ROS_INFO("Goal position: %f", msg->goal_pos);
12      ROS_INFO("Current position: %f", msg->current_
13      ROS_INFO("Position error: %f", msg->error);
14      ROS_INFO("Velocity: %f", msg->velocity);
15      ROS_INFO("Load: %f", msg->load);
16      ROS_INFO("Moving: %s", msg->is_moving ? "yes"
17      ROS_INFO(" ");
18  }
19
20  int main(int argc, char **argv) {
21      ros::init(argc, argv, "servo_status");
22      ros::NodeHandle node;
23
24      std::string servo_status_topic;

```

```

25     ros::param::param<std::string>(
26         "~servo_status_topic",
27         servo_status_topic,
28         "/finger_servo_controller/state");
29
30     ros::Subscriber pub = node.subscribe<dynamixel
31         servo_status_topic,
32         10,
33         callback);
34
35     ros::spin();
36
37     return 0;
38 }

```

コールバックの中にdynamixel_msgs::JointStateというメッセージタイプを利用します。

このメッセージタイプはdynamixel_msgsパッケージに定義され、内容は以下のようになっています。

```

1     std_msgs/Header header
2         uint32 seq
3         time stamp
4         string frame_id
5     string name
6     int32[] motor_ids
7     int32[] motor_temps
8     float64 goal_pos
9     float64 current_pos
10    float64 error
11    float64 velocity
12    float64 load
13    bool is_moving

```

ビルド&実行

コンパイルして実行します。

1つ目のターミナルでcatkin_makeを実行します。

```

1     $ cd ~/catkin_ws/
2     $ catkin_make

```

実行する前にマニピュレータのサーボコントローラを起動することが必要です。

1つ目のターミナルで下記を実行してマニピュレータのグリッパーサーボコントローラを起動します。

```

1     $ source devel/setup.bash
2     $ roslaunch rsj_2017_servo_control dynamixel_test
3     ... logging to /home/geoff/.ros/log/619c447c-396

```

```

4   Checking log directory for disk usage. This may
5   Press Ctrl-C to interrupt
6   Done checking log file disk usage. Usage is <1GB
7
8   started roslaunch server http://alnilam:44912/
9
10  SUMMARY
11  =====
12  (省略)

```

2つ目のターミナルで下記を実行します。

```

1   $ cd ~/catkin_ws/
2   $ source devel/setup.bash
3   $ rosrun rsj_2017_servo_control servo_status
4   [ INFO] [1494855697.336794278]: --- Servo status
5   [ INFO] [1494855697.336922059]: Name: finger_joi
6   [ INFO] [1494855697.336968787]: ID: 5
7   [ INFO] [1494855697.337016662]: Temperature: 37
8   [ INFO] [1494855697.337069086]: Goal position: 0
9   [ INFO] [1494855697.337120585]: Current position
10  [ INFO] [1494855697.337170811]: Position error: 0
11  [ INFO] [1494855697.337226948]: velocity: 0.000000
12  [ INFO] [1494855697.337278499]: Load: 0.000000
13  [ INFO] [1494855697.337329531]: Moving: no
14  [ INFO] [1494855697.337372008]:
15  [ INFO] [1494855697.432684658]: --- Servo status
16  (省略)

```

(サーボモータの動作状況により上記数値が変化することがあります。)

3つ目のターミナルで下記を実行すると、`servo_status`のターミナルで数値の変更が確認できます。

```

1   $ cd ~/catkin_ws/
2   $ source devel/setup.bash
3   $ rosrun rsj_2017_servo_control set_servo_pos 0
4   [ INFO] [1494851539.189274395]: Setting servo pos
5   [Ctrl+cで止める]
6   $ rosrun rsj_2017_servo_control set_servo_pos -0.1
7   [ INFO] [1494851548.085785357]: Setting servo pos
8   [Ctrl+cで止める]

```

このソースは以下のURLでダウンロード可能です。

https://github.com/gbiggs/rsj_2017_servo_control

課題

1. サーボステータスに`error`と`load`と`is_moving`という値があります。
これらの利用により、サーボモータがストール(外力が加わり期待通りの回転ができない状態など)したかどうか判断できます。
サーボモータがストールした場合に警告をターミナルで表示するノードを作成してみましょう。

問い合わせ先：

高橋 三郎（パナソニック アドバンステクノロジー）（[takahashi dot saburo at jp dot panasonic dot com](mailto:takahashi.dot.saburo@jp.panasonic.com)）

長谷川 孔明（豊橋技術科学大学）

This project is maintained by [takahasi](#)

Generated on 2019-11-12

Hosted on GitHub Pages — Theme by [orderedlist](#)



This work is licensed under a [Creative Commons Attribution-ShareAlike 3.0 Unported License](#).