Robovie-Z ROS SDK

取扱説明書

(2020.8.21)

Vstone®

ヴイストン株式会社

目次

はじめに/注意事項

本書は、小型二足歩行ロボット「Robovie-Z」(以降「ロボット」及び「ロボット本体」と記述)を ROS で制御するための SDK(以降「本 SDK」と記述)に関する取扱説明書及びチュートリアルです。ロボット本体や関連ソフトウェア「MotionWorks」に関する操作方法は、製品に付属の各種資料を別途ご参照ください。同じく、Raspberry Pi4(以降「Raspberry Pi」と記述)の基本的な取り扱いにつきましては、それらに付属の各種資料をご参照ください。

本製品の使用にあたっては下記注意事項に従い、正しくご使用ください。

○ 本製品を無許可で複製、再配布、再販することはできません。ただし、著作権法で認められた範囲における複製については許可されます。
○ 本製品の対応環境は、Raspberry Pi4 Model B、Rasbian Buster、ROS Melodic です。それ以外の環境での使用についてはサポート対象外となる他、それによって生じたいかなる損害についても、製造元および販売元は何らの責任を負いません。
○ 本製品の使用にあたっては、本製品に含まれない公開ライブラリおよびアプリケーションを多数使用する必要 がございます。本製品に含まれないライブラリについてはサポート対象外となる他、その使用によって生じたいかなる損害についても、製造元および販売元は何らの責任を負いません。
○ 本製品を使用する Raspberry Pi はお客様にてご準備ください。お使いの環境及び周辺機器の種類によっては、デバイスドライバの対応状況等により、一部の機能が正常に動作しない可能性がございますが、 デバイス固有の問題についてはサポート対象外となる他、それによって生じたいかなる損害についても、製造元および販売元は何らの責任を負いません。

動作環境

本 SDK の動作環境は下記の通りです。

○ ハードウェア: Raspberry Pi4 Model B
○ OS:Raspbian Buster
○ ストレージ:2GB 以上の空き容量
○ インターネット接続されたネットワーク環境

ROS について

ROS の概要

ROS（Robot Operation System）は、OSRF（Open Source Robotics Foundation）によって開発・メンテナンスされているロボット用のミドルウェアです。分散処理が求められる複雑なロボットシステムを制御できる性能を備えており、世界中の研究者や開発者が作成した豊富なライブラリを使用することができます。ロボット制御システムの作成を効率化できることから、人型ロボットから車両型ロボット、水上・水中ロボットやドローンに至るまで、幅広い分野で活用されています。

ROS の特徴のひとつが、BSD ライセンスに基づくオープンソースとして公開されており、誰でも開発に参加し貢献できることです。ROS には強力な開発者コミュニティが存在し、誰でも使用可能な 5000 以上のライブラリのほとんどは、OSRF ではなくコミュニティによって開発・メンテナンスされています。

ROS 本体が BSD ライセンスによって提供されているため、ROS を用いて開発した成果物は、商用利用することが可能です。ROS を用いて動作する様々なロボットが発売されており、メガローバーもそのひとつです。ただし、ライブラリによっては BSD ではないライセンスによって提供されているものも存在するため、商用利用ではご注意ください。

ROS が提供する機能

ROS によって提供される主要な機能について、説明します。

メッセージ通信

ROS を用いて構成されるロボットシステムは分散処理が基本となっており、ユーザは「ノード」と呼ばれるプログラムを複数立ち上げることでシステムを作成します。例えば、ゲームパッドで操作できる台車ロボットであれば、「ゲームパッドの入力値を取得するノード」、「入力値に従って移動指令を出すノード」、「移動指令をもとにモータを回転させるノード」などが必要となります。当然、ノード間で情報を通信する仕組みが求められます。各ノード間で、センサ入力値の情報やカメラの映像、制御指令値といったデータをやり取りするために、ROS では Pub/Sub 方式によるメッセージ通信が提供されています。開発者はわずか数行のコードにより、任意の情報をパブリッシュ（配信）したり、サブスクライブ（購読）したりすることができます。メッセージには、構造体のような型が定められているため、ROS ノード同士であれば互換性を気にする必要はありません。また、型は自作することもできます。

パッケージ管理

ROS のライブラリやプログラムは、パッケージという単位で管理されています。パッケージの中にはノードやその設定ファイル、起動スクリプトなどが含まれており、ユーザは使用したい機能を持つパッケージをインターネット上からダウンロードし、ローカル環境に組み込むことができます。パッケージの追加や削除といった操作は非常に簡単に行う事ができます。また、ユーザが独自の制御プログラムを開発する際には、まずパッケージを作成し、その中で開発を行う事になります。

デバイスドライバ

ROS では様々なデバイスのドライバがパッケージの形式で提供されています。対応しているデバイスであれば、パッケージを導入し、デバイスを接続するだけで使用することができます。

ハードウェア抽象化

ROS による制御システムは複数のノードによる分散処理によって動作します。これにより、ハードウェアが異なるロボットでも、上流の制御システムは同じものが使用できます。

ライブラリ

ROS では、5000 を超える公開ライブラリを使用することができます。それらはデバイスドライバのようなものから、経路計画や動作生成といったものまで様々です。

視覚化ツール

ROS には、いくつかの視覚化ツールが存在しており、ロボットの操縦 UI やデバッグ等のために活用されています。中でも「Rviz」は強力なツールです。3D 表示機能を持つこのツールは、実に多くのパッケージで情報を表示するために使われています。表示情報のカスタマイズが容易ですので、ユーザオリジナルの UI を作成することも容易です。

ROS に関する情報の集め方

ROS を用いた開発を行う際には、使用するパッケージの情報など、様々な情報が必要になります。ROS やその開発に関する情報は書籍から集めることもできますが、ここではインターネットから情報を集める際に参考になるサイトをいくつかご紹介します。

- ROS Wiki - http://wiki.ros.org/

ROS の公式 Wiki です。ROS のインストール方法からチュートリアル、各公開パッケージの情報まで、様々な情報が公開されています。ただ、パッケージの情報などが更新されないまま古くなっていることもありますので、ご注意ください。

- ROS Wiki(ja) - http://wiki.ros.org/ja

ROSWiki の日本語訳版です。現在も有志による精力的な翻訳作業が行われていますが、古い情報も多いので、英語版とあわせて確認した方がよいでしょう。

- ROS Answers - https://answers.ros.org/questions/

ROS の Q&A フォーラムです。パッケージを使用した際のエラーの解消法など、様々な情報が蓄えられています。

ROS Melodic のインストール

ROS Melodic を Raspberry Pi にインストールする方法を説明します。ここで述べる手順は、2020 年 7 月現在、公式の ROS Wiki で説明されている Raspberry Pi への ROS の導入手順を元にしています。
http://wiki.ros.org/ROSberryPi/Installing%20ROS%20Melodic%20on%20the%20Raspberry%20Pi

以降の作業は、SSH によるアクセスもしくは RaspberryPi にモニタとキーボードを接続するなどして、内部 OS にログインしてください。また、RaspberryPi をインターネットに接続できる状態にしてください。

説明中、コマンド記載行(黒背景の箇所)の行頭に付加された$は、コンソール画面上のプロンプトを表しているので、実際のコマンド実行の際には$は不要です。また、コマンドが長い場合は複数行にわたり記載されていますが、次の$が存在する箇所までが一つのコマンドになるため、途中で改行を挟まず 1 文で実行してください。

リポジトリの準備

リポジトリと鍵の設定を行います。

```
$ sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" >
/etc/apt/sources.list.d/ros-latest.list'
$ sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net:80 --recv-key
C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654
```

インストールされたパッケージを更新します。

```
$ sudo apt-get update
$ sudo apt-get upgrade
```

ブートストラップの依存関係を解決します。

```
$ sudo apt install -y python-rosdep python-rosinstall-generator python-wstool python-rosinstall
build-essential cmake
```

rosdep を初期化します。

```
$ sudo rosdep init
$ rosdep update
```

ROS のインストール

ROS のコアをビルドするため、catkin のワークスペースフォルダを作成します。下記コマンドでディレクトリを作成し、カレントディレクトリを移動してください。

```
$ mkdir -p ~/ros_catkin_ws
$ cd ~/ros_catkin_ws
```

ROS コアのビルドを行います。

```
$ rosinstall_generator desktop --rosdistro melodic --deps --wet-only --tar > melodic-desktop-wet.rosinstall
$ wstool init src melodic-desktop-wet.rosinstall
```

依存関係を解決します。

```
$ cd ~/ros_catkin_ws
$ rosdep install -y --from-paths src --ignore-src --rosdistro melodic -r --os=debian:buster
```

一度ワークスペース全体をビルドします。

```
$ sudo ./src/catkin/bin/catkin_make_isolated --install -DCMAKE_BUILD_TYPE=Release --install-space /opt/ros/melodic
```

以上で ROS の基本部分はインストール完了です。併せて下記のコマンドで setup.bash を操作して環境変数の設定を行ってください。

```
$ source /opt/ros/melodic/setup.bash
$ echo "source /opt/ros/melodic/setup.bash" >> ~/.bashrc
$ echo "source ~/ros_catkin_ws/devel/setup.bash" >> ~/.bashrc
$ source ~/.bashrc
```

リリースパッケージの追加

ROS の利用に便利な各種パッケージを導入します。

先ほど作成したディレクトリに移動します。既にディレクトリが移動済みであれば下記の入力は不要です。

```
$ cd ~/ros_catkin_ws
```

ros_comm、ros_controll、joystick_drivers の各パッケージをインストールします。

```
$ rosinstall_generator ros_comm ros_control joystick_drivers --rosdistro melodic --deps --wet-only
--tar > melodic-custom_ros.rosinstall
```

ワークスペースをアップデートします。

```
$ wstool merge -t src melodic-custom_ros.rosinstall
$ wstool update -t src
```

rosdep を実行し、新しいパッケージ群の依存関係を解決します。

```
$ rosdep install --from-paths src --ignore-src --rosdistro melodic -y -r --os=debian:buster
```

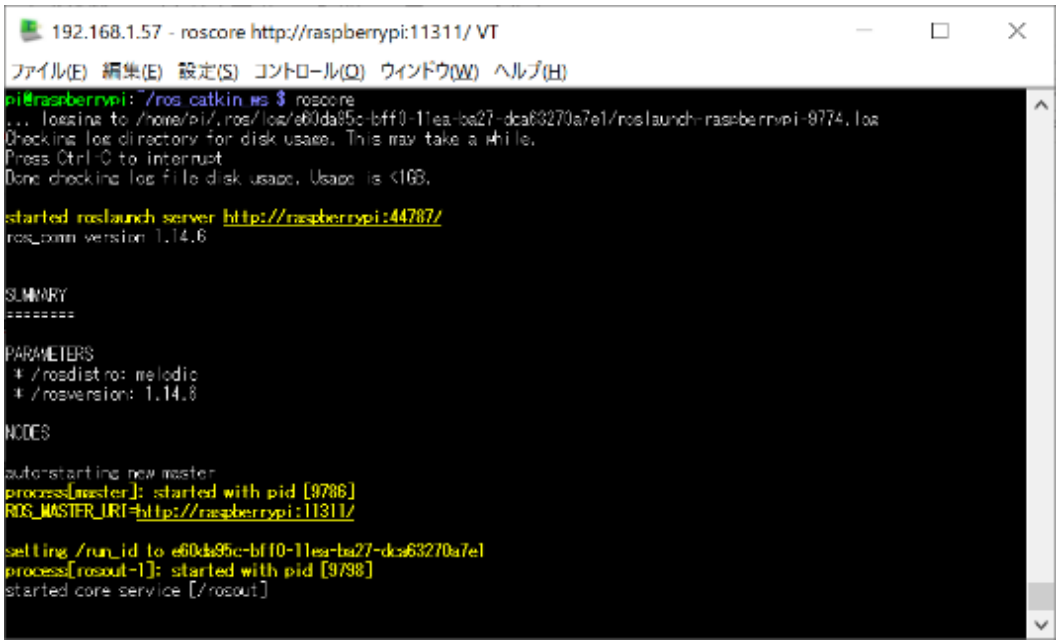
ワークスペースをビルドしなおします。

```
$ sudo ./src/catkin/bin/catkin_make_isolated --install -DCMAKE_BUILD_TYPE=Release --install-space
/opt/ros/melodic
```

以上で ROS Melodic のインストールは完了です。正常にインストールできているかの確認のため、一度 roscore を起動してみましょう。

```
$ cd ~/ros_catkin_ws
$ roscore
```

roscore を起動すると下記のような画面が表示されます。起動が確認できたら、Ctrl+C キーなどで roscore を終了させてください。



SDK の導入

次に、下記の手順で github のリポジトリより本 SDK をインストールします。引き続きコンソール上でコマンドを実行してください。

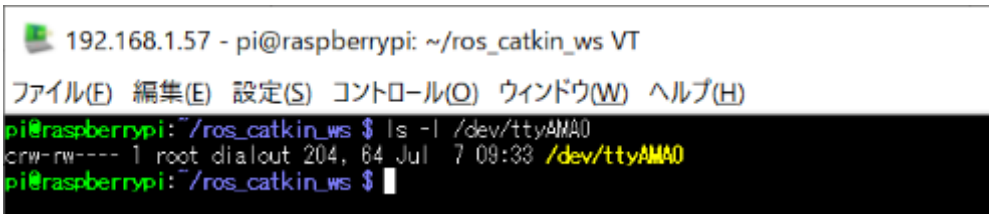
ワークスペースの src フォルダに移動し、github よりプロジェクトをクローンします。

```
$ cd ~/ros_catkin_ws/src
$ git clone https://github.com/vstoneofficial/roboviez_ros_samples.git
```

クローンしたプロジェクトをビルドします。表示が[100%]になればビルド完了です。

```
$ cd ~/ros_catkin_ws
$ catkin_make
```

本 SDK に含まれるノードを実行して動作確認してみましょう。本 SDK はロボット本体の CPU ボード(VS-RC026)をシリアル通信で制御します。シリアルポートは OS から「/dev/ttyAMA0」として認識されています。



なお、OS からデバイスが認識されていても、CPU ボードの電源が OFF になっている等の場合は正しく通信できないため、本 SDK でロボットを制御する場合は、必ず CPU ボード側の電源も ON にしてください。

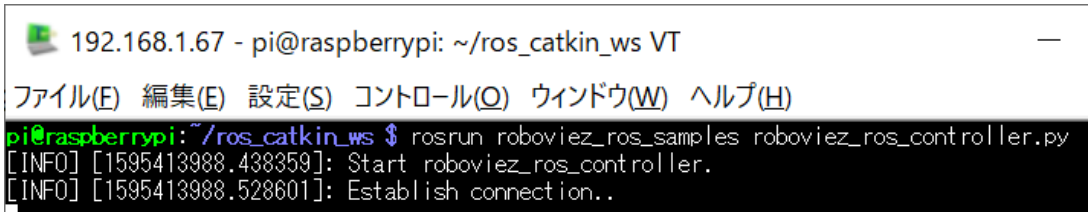
動作確認として、ロボットに搭載された 6 軸センサの情報を読み出してみます。まずはコンソールから roscore を実行してください。

```
$ roscore
```

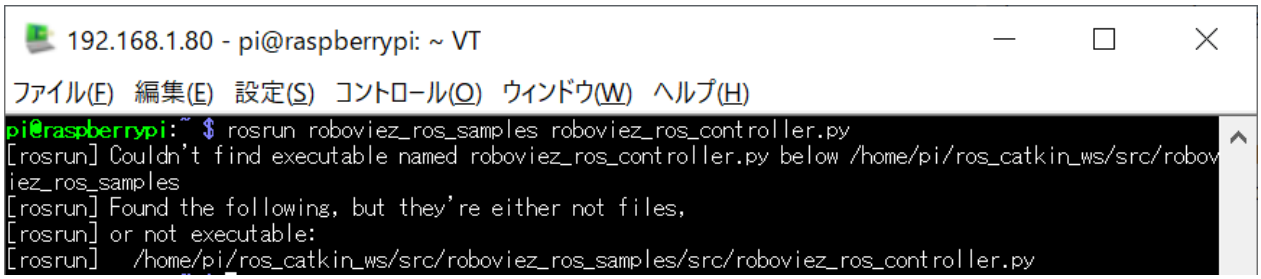
続いて本 SDK のノードを実行します。ここでは、先程実行した roscore を動作させたまま別のコマンドを入力していきます。先程とは別のコンソールを新たに立ち上げて、下記のコマンドを実行してください。

```
$ rosrun roboviez_ros_samples roboviez_ros_controller.py
```

roboviez_ros_controller は、ROS からロボット本体を制御するための基本的な処理を実装したノードです。実行すると画面に下記のような文字列が表示されます。



もし下記のように表示されてすぐにノードが終了する場合、roboviez_ros_controller.py に実行権限が付与されていない可能性があります。chmod +x などのコマンドで、roboviez_ros_samples の src フォルダに含まれる roboviez_ros_controller.py のファイルに実行権限を与えてください。



続いて、ノードを実行したまま更に別のコマンドを実行します。もう一つ別のコンソールを立ち上げて、下記のコマンドを入力してください。

```
$ rostopic echo /roboviez_ros_controller/imu
```

このコマンドは、ノードから配信されている 6 軸センサの情報をコンソールにエコーバックします。実行すると下記のように現在のジャイロ・加速度・温度センサの情報が画面に表示されます。

temperature は温度(℃)、gyro は x/y/z の角速度(degree per second)、acc は x/y/z の加速度(G)をそれぞれ表しています。

もしメッセージが画面に表示されない場合、CPU ボードの電源が ON になっていない可能性があります。CPU ボードへの電源共有が行われていない・通信経路が断線しているなどの問題により、ノードが CPU ボードと正しく通信できない場合、配信されるはずの値が取得できなかったり、取得できても全て 0 になったりする場合があります。

本 SDK のノードでは、6 軸センサ以外の様々な情報を取得したり、逆にモーション呼び出しなど外部からの指示を受けてロボットを制御したりする機能を提供しています。これらの詳細は「メッセージ仕様」の項目をご参照ください。

SDK の構成

本 SDK のディレクトリ・ファイルの概要は下記の通りです。

roboviez_ros_msgs

本 SDK で使用するメッセージ・サービスに関して設定した ROS パッケージです。msg ディレクトリにメッセージ、srv ディレクトリにサービスの設定がそれぞれ含まれています。

roboviez_ros_samples

本 SDK のコントローラ・ノードに関する ROS パッケージです。src ディレクトリには ROS のノード及びロボット本体との通信に関するソースコードが含まれています。launch ディレクトリにはゲームパッドでの歩行操縦などのサンプルを実行する launch ファイルが含まれています。mesh 及び urdf ディレクトリは rviz での 3D モデル描画に関するデータを格納しており、前者は 3D モデルの頂点データ、後者は軸構成などを記録した xacro ファイルが含まれます。

ソースコードは「roboviez_ros_controller.py」がロボットを制御するメイン処理の ROS ノード、「roboviez_joy_walk.cpp」が rviz による 3D 表示に関連した ROS ノード、「roboviez_joy_walk.cpp」がゲームパッド歩行サンプルの ROS ノードに関するものです。

メッセージ仕様

本 SDK に含まれるコントローラ「roboviez_ros_controller」が備えているメッセージ及びサービスについて説明します。

Publish

コントローラが配信(Publish)するメッセージです。これらを購読(Subscribe)することで、ロボットの情報を得ることができます。

6 軸センサ

6 軸センサから得られる角速度・加速度・気温の情報です。数値は、MotionWorks などで取得できるメモリマップ上の生値から実単位に変換されています。X/Y/Z の座標系はロボット本体の仕様と同一です。

メッセージ名	/roboviez_ros_controller/imu
型	roboviez_ros_msgs/IMU
内容	Temperature:　//気温(float64)。単位は℃ gyro:　//角速度(geometry_msgs/Vector3)。単位は degree per second x:　//　x 軸周りの角速度（float64） y:　//　y 軸周りの角速度（float64） z:　//　z 軸周りの角速度（float64） acc:　//加速度(geometry_msgs/Vector3)。単位は G x:　//　x 軸方向の加速度（float64） y:　//　y 軸方向の加速度（float64） z:　//　z 軸方向の加速度（float64）

ゲームパッド

ロボット本体に付属のコントローラ「VS-C3」の入力に関する情報です。各数値はメモリマップ上の生値と同一です。

メッセージ名	/roboviez_ros_controller/gamepad
型	roboviez_ros_msgs/Gamepad
内容	button:　//ボタン入力(int16)。 各 bit とボタンの対応はロボット本体の資料を参照 stick_lx:　//左スティックの横方向の傾き（int16)。範囲は-128～+127 stick_ly:　//左スティックの縦方向の傾き（int16)。範囲は-128～+127 stick_rx:　//右スティックの横方向の傾き（int16)。範囲は-128～+127 stick_ry:　//右スティックの縦方向の傾き（int16)。範囲は-128～+127

モーション実行状況

ロボットが MotionWorks の操作マップで作成したプログラムを実行している場合、現在実行しているモーションの情報を得られます。マップ番号は 0～3 の数値が MotionWorks で作成可能な Map1～4 に対応します。モーション番号は、0 がアイドリングモーション、1 以降は各マップにリストアップされたモーションに対して上から連番で振られた番号に相当します。

メッセージ名	/roboviez_ros_controller/motionmap
型	roboviez_ros_msgs/MotionMap
内容	map_num:　　//現在実行中の操作マップのマップ番号（int16)。 motion_num:　//現在実行中の操作マップ内のモーション番号（int16)。

バッテリ電圧

ロボットの CPU ボードに対して供給されている電圧値です。数値はメモリマップ上の生値から実単位に変換されています。

メッセージ名	/roboviez_ros_controller/voltage
型	std_msgs/Float64
内容	data:　　//現在の CPU ボードのバッテリ電圧(float64)。単位は V

モータポテンショ

各関節のモータから得られた位置情報です。数値はメモリマップ上の生値から実単位に変換されています。

メッセージ名	/roboviez_ros_controller/positions
型	std_msgs/Float64MultiArray
内容	data:　　//現在の各モータ角度を格納した配列(float64[])。 //先頭から ID1～ID26 の角度を順に記録。単位は rad

Subscribe

コントローラが購読(Subscribe)するメッセージです。ここに対してメッセージを配信(Publish)することで、ロボットを制御することができます。

モータ ON/OFF

ロボットのモータの ON/OFF を切り替えます。モータを ON にすると、その時点で設定されている各モータ角度でモータが ON になるため、直立状態などあらかじめ安全に ON にできるポーズを実行してから行うようにして下さい。また、各モータのゲインが 0 に設定されていると、本メッセージでモータを ON にしても実際にモータに力が入りません。

モータを OFF にする際も、転倒等が発生しないよう本体を手で支える・転倒の心配がないポーズにするなどしてから行うようにして下さい。

メッセージ名	/roboviez_ros_controller/poweron
型	std_msgs/Bool
内容	data:　　//true ならモータ ON、false なら OFF にする

UART モーション

ロボット本体に MotionWorks より「パッド&UART 操作.RM4」のプログラムを書き込んでいる場合、このメッセージで指定した番号のモーションを呼び出して実行します。なお、実際にモータを動かす場合は、あらかじめ/roboviez_ros_controller/poweron にもメッセージを配信してモータを ON にしておく必要があります。

メッセージ名	/roboviez_ros_controller/uartmotion
型	std_msgs/Int16
内容	data:　　//再生させるモーション番号

ロボットのメモリマップ上では、呼び出したモーションが終了してもここで指定した数値が自動的に書き換わらないため、場合によっては指定したモーションが何度も実行されます。モーションを止める場合は、再度メッセージを配信して数値を適時書き換えてください。また、指定したモーションが実際に呼び出されたかは、/roboviez_ros_controller/motionmap のメッセージを購読してモーション番号を調べることで確認が可能です。

歩行

ロボット本体に MotionWorks より「パッド&UART 操作.RM4」のプログラムを書き込んでいる場合、このメッセージで歩行プログラムを呼び出して、前後左右の移動量・左右の旋回量を指定した歩行が可能です。なお、実際にモータを動かす場合は、あらかじめ/roboviez_ros_controller/poweron にもメッセージを配信してモータを ON にしておく必要があります。

メッセージ名	/roboviez_ros_controller/walking
型	roboviez_ros_msgs/Walking
内容	walking: //true で歩行開始、false で歩行終了 (bool)。 v_axis: //前後方向の移動量(float64) ±1.0 の範囲で指定 h_axis: //左右方向の移動量(float64) ±1.0 の範囲で指定 z_axis: //左右方向の旋回量(float64) ±1.0 の範囲で指定

Service

コントローラが提供しているサービス(Service)です。これらのサービスにクライアントよりリクエストすることでロボット本体を制御できます。

メモリ読み込み

ロボットのメモリマップからデータを読み込みます。データは指定のアドレスから始まる連続するメモリブロックから、指定のサイズ分取得できます。実データ部は 1byte 単位であり、データ長が 2byte、4byte のアドレスに対しては、リトルエンディアンでデータを分解して処理する必要があります(例:0x1234 という 2byte のデータの場合、Buf.data[0]=0x34、Buf.data[1]=0x12 になります)。

メッセージ名	/roboviez_ros_controller/memmap_read
型	roboviez_ros_msgs/MemRead
リクエスト	Address: //読み込みを開始するアドレス(Int16) Length: //読み込むデータ量(Int8)
レスポンス	Buf: //読み込んだデータ(uint8[])

メモリ書き込み

ロボットのメモリマップにデータを書き込みます。データは指定のアドレスから連続するメモリブロックに対して行われます。実データ部は 1byte 単位であり、データ長が 2byte、4byte のアドレスに対しては、リトルエンディアンでデータを分解して処理する必要があります(例:0x1234 という 2byte のデータの場合、Buf.data[0]=0x34、Buf.data[1]=0x12 になります)。

メッセージ名	/roboviez_ros_controller/memmap_write
型	roboviez_ros_msgs/MemWrite
リクエスト	Address: //書き込みを開始するアドレス(Int16) Buf: //書き込むデータ(std_msgs/UInt8MultiArray) data: //データ内容(uint8)
レスポンス	なし

制御サンプル

本 SDK を用いてロボット本体を制御する事例をいくつか紹介します。ここではコンソール上のコマンドで制御していますが、同様の方法で自作の ROS ノードなどからプログラムで制御することも可能です。

モーションの呼び出し

/roboviez_ros_controller/uartmotion にメッセージを配信して、ロボットのモーションを呼び出します。

ROS からの操作を行う前に、MotionWorks より、ロボット本体に「パッド&UART 操作.RM4」のプログラムを書き込んでください。モーションを書き込んだら PC との接続ケーブル(USB)を外し、バッテリ/外部電源からロボット本体を起動してください。

ロボットを起動したら、コンソールより roscore を実行してください。

```
$ roscore
```

続いて、別のコンソールを立ち上げて、roboviez_ros_controller を実行してください。

```
$ rosrun roboviez_ros_samples roboviez_ros_controller.py
```

続いて、更に別のコンソールを立ち上げて、下記コマンドでモータを ON にします。この時、ロボットが転倒したりしないよう、ロボットを仰向けで寝かせたり、指の挟みこみに注意して腰部分をつかんで持ち上げるなどしながら実行してください。

```
$ rostopic pub -1 /roboviez_ros_controller/poweron std_msgs/Bool "data: true"
```

モータが ON になったら、同じコンソールから下記コマンドを実行してモーションを呼び出してください。

```
$ rostopic pub -1 /roboviez_ros_controller/uartmotion std_msgs/Int16 "data: 1"
```

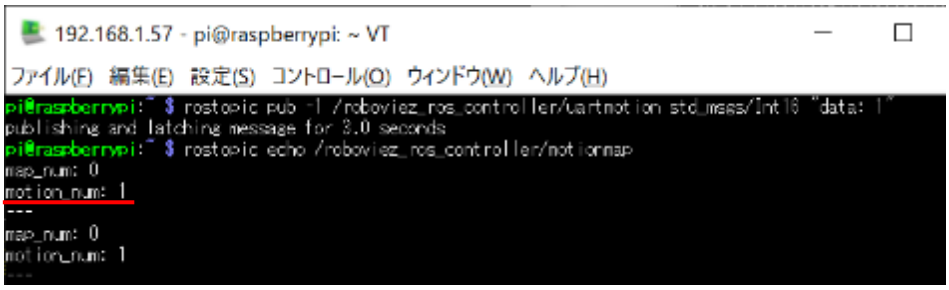
コマンドを実行すると、割り当てたモーションが実行されます。ただし、モーションが終了しても再度同じモーションが呼び出されます。これはロボットに指示したモーション番号の設定がモーションの終了では書き換わらないためです。モーションを止める場合は、下記コマンドを実行してロボットのモーション番号を更新してください。

```
$ rostopic pub -1 /roboviez_ros_controller/uartmotion std_msgs/Int16 "data: 0"
```

ワンショットでモーションを実行する場合、「モーションが呼び出されたのを確認してからモーション番号を 0 に戻す」という処理が適切ですが、「モーションが呼び出されたか」の確認には、/roboviez_ros_controller/motionmap のメッセージを購読します。

```
$ rostopic echo /roboviez_ros_controller/motionmap
```

メッセージの「motion_num」が指示したモーション番号と同じ場合は、既にモーションが呼び出されている状態を表すため、この数値を確認してからモーション番号を戻すメッセージを送信すると、モーションをワンショットで呼び出しできます(ただし、極端に実行時間が短いモーションの場合は、ROS 上でメッセージを読み取る前に終了してしまう可能性があります)。



歩行

/roboviez_ros_controller/walking にメッセージを配信して、ロボットを任意に歩行させて操縦します。前項のモーションの呼び出しと同じく、複数のコンソールより roscore、roboviez_ros_controller、rospub をそれぞれ実行すれば同様の制御が可能ですが、本 SDK ではこのサンプルに関して launch ファイル及び別のコントローラを準備しており、一つのコンソールウィンドウ上で一つの launch ファイルを実行するだけで、必要なノードが全て起動し、Raspberry Pi に接続したゲームパッドからロボットを操縦できます。

なお、本サンプルの実行には、市販の PC 用 USB ゲームパッドなど、Raspberry Pi で認識できるゲームパッドが必要です(ロボットに付属のコントローラ「VS-C3」は本サンプルでは使用できません)。操作方法は下図の通りで VS-C3 のボタン・スティック配置を想定しているため、これと同じ配置のゲームパッドのご利用を推奨します。同じ配置のゲームパッドが無い場合、「SELECT、START、歩行をアサインする 3 ボタン」「前後左右移動・左右旋回をアサインするアナログ入力 3 軸」を備えたゲームパッドをご用意ください。

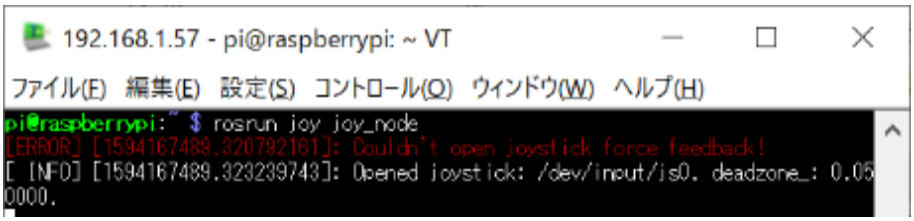
サンプルの実行の前に、お使いのゲームパッドの入力に応じてサンプルソースを書き換える必要があります。Raspberry Pi にゲームパッドを接続したら、コンソールより roscore を実行してください。

```
$ roscore
```

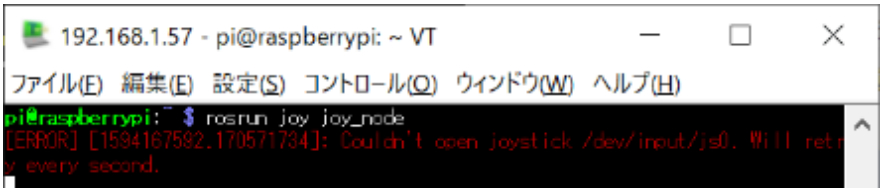
続いて、別コンソールを開いて下記コマンドを実行してください。これは、ROS 上でゲームパッドの制御を行うためのノードを実行しています。

```
$ rosrun joy joy_node
```

コマンドを実行し、画面に「Opened joystick:」と表示されたら ROS からゲームパッドが正しく認識されています。



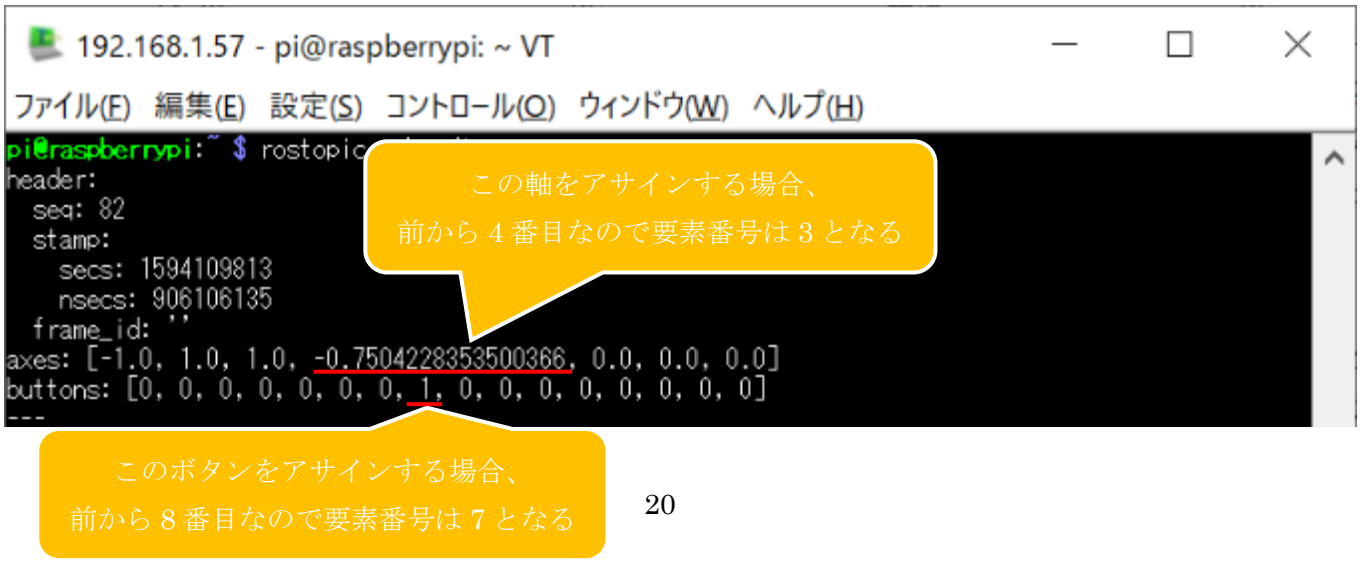
画面に「Couldn't open joystick /dev/～」と表示される場合、ROS から joystick が認識されていません。Raspberry Pi に正しくゲームパッドが接続されているか、ゲームパッドが Raspberry Pi/ROS に対応しているかご確認ください。



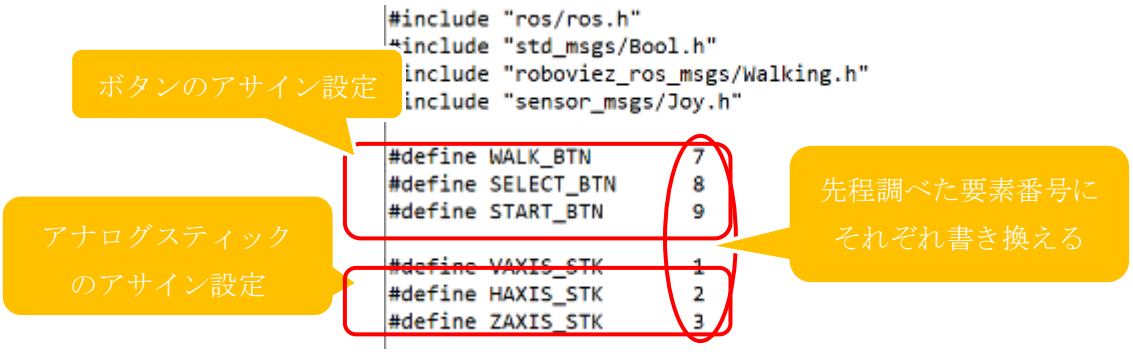
更に別のコンソールを開いて下記コマンドを実行してください。これは、上記で実行したノードから配信されるコントローラの入力情報を画面にエコーバックするコマンドです。

```
$ rostopic echo /joy
```

コマンドを実行してコントローラのボタン・スティックを操作すると、最後に開いたコンソールの画面に下記のような文字列が表示されます。「axes」にはコントローラの全アナログ入力要素、「buttons」には全ボタンの入力状態がそれぞれ表示されるので、アサインしたいボタン・アナログスティックの要素番号を確認してください。要素番号は先頭を 0 として並び順で数えます。



要素番号を調べたら、サンプルの C 言語ソースを書き換えます。テキストエディタなどで「/roboviez_ros_samples/src/roboviez_joy_walk.cpp」を開き、「#define」で定義されたマクロの数値を書き換えます。ボタンの割り当ては「WALK_BTN」「SELECT_BTN」「START_BTN」、アナログスティックの割り当ては「VAXIS_STK(前後移動)」「HAXIS_STK(左右移動)」「ZAXIS_STK(旋回)」を、それぞれ書き換えてください。



サンプルを書き換えたら上書き保存して、ワークスペースをビルドしてください。

```
$ cd ~/ros_catkin_ws
$ catkin_make
```

以上でサンプルを使用するための準備は完了です。サンプルを実行する場合は下記の手順で行います。

コンソールでコマンドを実行する前に、MotionWorks より、ロボット本体に「パッド&UART 操作.RM4」のプログラムを書き込んでください。モーションを書き込んだら PC との接続ケーブル(USB)を外し、バッテリ/外部電源からロボット本体を起動してください。

ビルドしたら、roslaunch コマンドで launch ファイルを実行します。下記コマンドを実行してください。

```
$ roslaunch roboviez_ros_samples joy_controller.launch
```

コマンドを実行すると、roscore や各種コントローラが自動的に実行され、ゲームパッドで操縦できる状態になります。ゲームパッドの SELECT と START ボタンを同時に押すとモータの ON/OFF を切り替えられます。WALK_BTN にアサインしたボタンを押すと歩行モーションを開始し、ボタンを押しながら前後・旋回をアサインしたアナログスティックを動かすと、入力に応じて移動します。

本サンプルを参考にすることで、を書き換えてゲームパッド以外の入力によってロボットを制御したり、このソースの内容を自作プログラムに取り込むといったことが容易に可能です。また、「加速度センサ情報を subscribe し、転倒が検出されたら起き上がりモーションを呼び出し/モータを OFF」と言った改変を行うことで、更に柔軟な歩行が可能になります。

メモリマップの読み書き

センサやゲームパッド等の情報は常に配信されますが、配信されていないメモリマップの他の情報については、コントローラのサービスにリクエストを出すことで操作することが可能です。ここではコンソールからユーザ変数を読み書きしてみましょう。

ここではユーザ変数の領域からアドレス 0x0f20(10 進数で 3872)に対して読み書きしてみます。なお、もしロボット内でこのアドレスを操作するモーションが実行されている場合、メモリマップの内容が意図したものと異なるものになる可能性があります。

ロボットを起動したら、コンソールより roscore を実行してください。

```
$ roscore
```

続いて、別のコンソールを立ち上げて、roboviez_ros_controller を実行してください。

```
$ rosrun roboviez_ros_samples roboviez_ros_controller.py
```

続いて、更に別のコンソールを立ち上げます。このコンソール上で rosservice コマンドを利用してサービスにリクエストを送信します。Address は読み込むアドレス番号、Length は読み込むサイズです。ユーザ変数は原則として 2byte のサイズなので、Length は 2 を指定します。数値は 0x のプリフィクスを付けると 16 進数で表記できます。

```
$ rosservice call /roboviez_ros_controller/memmap_read "{Address: 0x0f20, Length: 2}"
```

コマンドを実行すると、現在のアドレス 0x0f20 の内容が返って来ます。Buf:[]の部分が、アドレスの数値を 1byte ずつに分解したものになります(表記は 10 進数)。ユーザ変数の数値は原則として 2byte で処理されますが、本サービスの上では変数のサイズに限らず 1byte ずつに分解されるため、クライアント側で数値を再構築する必要があります。数値のバイトオーダーはリトルエンディアンで、

今回の場合 data の内容を 2byte に直すと 0x0000 となり、数値が 0 であることがわかります。

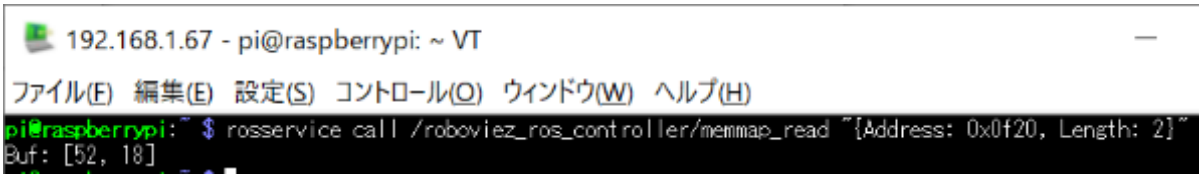
それでは、このアドレスに数値を書き込んでみましょう。ここでは 16 進数の 0x1234 を書き込みます。この数値をリトルエンディアンで 1byte ずつに分解すると、0x34,0x12 となります。下記のコマンドで数値を書き込んでみましょう。

```
$ rosservice call /roboviez_ros_controller/memmap_write "{Address: 0x0f20,Buf:{data: [0x34,0x12]}}"
```

コマンドを実行すると、コンソールには特に何も表示されませんが、メモリマップには指定の数値が書き込まれています。実際に数値が書き込まれたか、再度メモリ読み込みのサービスにリクエストを送信して確認してみましょう。

```
$ rosservice call /roboviez_ros_controller/memmap_read "{Address: 0x0f20, Length: 2}"
```

コマンドを実行すると下記のような文字列が表示されます。Data の数値は 52,18 ですが、それぞれ 16 進数に直すと 0x34,0x12 です。リトルエンディアンのバイトオーダーに倣い数値を 2byte に直すと 0x1234 となり、正しく数値が書き込めていることがわかります。



3D モデルの表示と情報のフィードバック

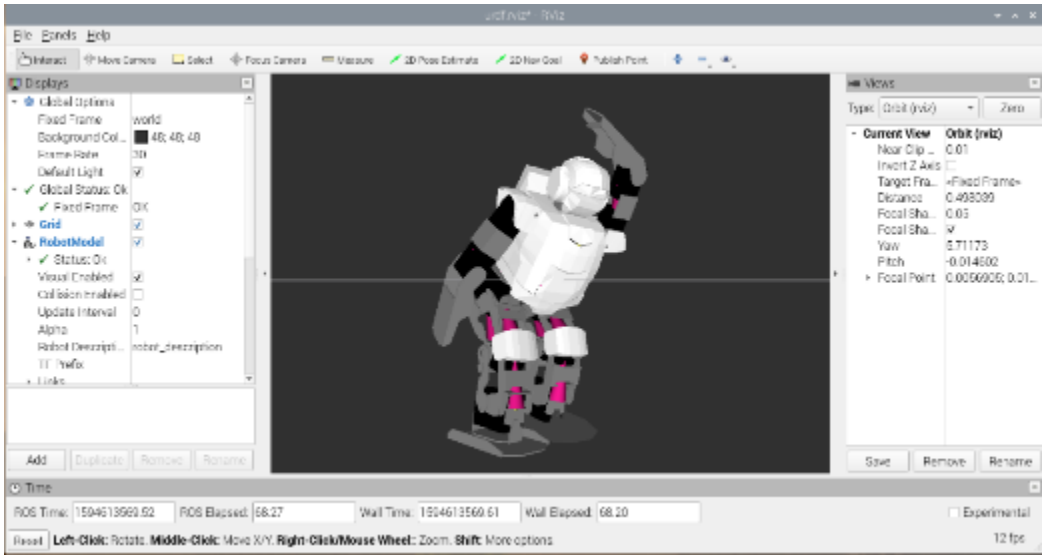
本 SDK にはロボットの簡易 3D モデルデータが含まれており、ROS の rviz でモデルを表示できます。また、ロボット本体から読み込んだ加速度と各モータ角度を画面上のモデルにフィードバックできます。

この機能は、Raspberry Pi の GUI 環境で行う必要があります。Raspberry Pi に GUI をインストールし、また本体にモニタ・キーボード・マウスを接続して直接ログインするなどして、GUI 環境上のコンソール(端末)からコマンドを実行してください。

この機能については、lunch ファイルが準備されています。コンソールから下記コマンドを実行してください。

```
$ roslaunch roboviez_ros_samples rviz.launch
```

コマンドを実行すると、画面にロボットの 3D モデルを表示します。3D モデルはロボット本体の傾き・ポーズが常にフィードバックされます。ただし、CPU ボードに外部電力が供給されてない場合、モータから角度を読み込めないためポーズはフィードバックされません。また、rviz はシミュレータとは異なるため、物体同士の干渉や絶対位置の移動と言った要素は再現されません。



商品に関するお問い合わせ

TEL: 06-4808-8701　　FAX: 06-4808-8702　　E-mail: infodesk@vstone.co.jp

受付時間　：9:00～18:00（土日祝日は除く）

ヴイストン株式会社　　www.vstone.co.jp

〒555−0012　大阪市西淀川区御幣島 2-15-28