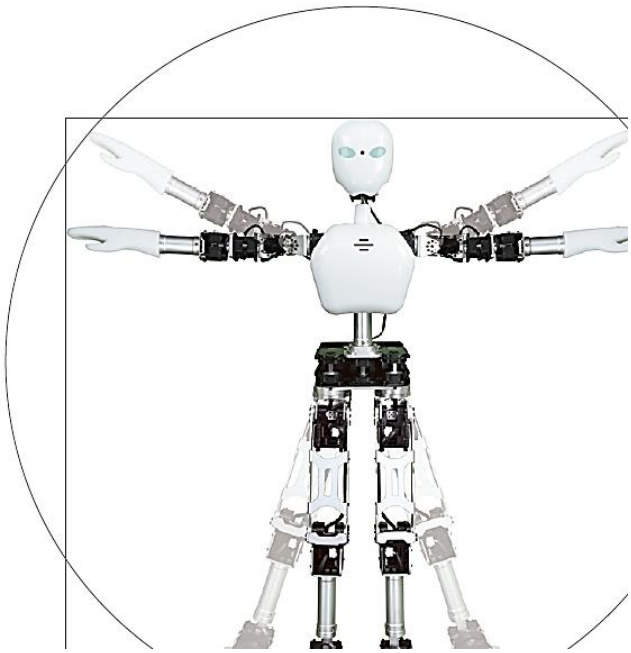
ROS Manual for UXA-90

ROS: Robot Operating System



Index

Summary

Name	UXA-90 Light
Weight	9.5kg
Height	100cm
Width	35cm
Walking speed	30cm/sec
DOF(Degree of Freedom)	23 DOF 12 DOF for legs 8 DOF for arms 1 DOF for waist 2 DOF for head
Sensor	IMU 2G 9Axis, ±180˚(Roll/Yaw), ±90˚(Pitch)
Computer	AMD E2-1800 Processor dual-core 1.7GHz 2GB DDR RAM 64GB(SSD)
Wireless	802.11 b/g/n
External interface	USB2.0 x 2, Ethernet 10/100/1000 Base T USB 3.0 x 2, HDMI x 1
Speaker	1ea
Microphone	1ea
Vision Camera	Logitech C905/HD 1600x1200 pixel
Battery	Lithium Polymer 18.5V, 2250mA
Operation hour	20min ~ 40min
Charging time	30min
	Windows7 Linux Ubuntu 14.04

This manual includes guidelines for ROS(Robot Operating System) for UXA90 Light. UXA90 is a humanoid that is 100cm tall and 9.5kg in weight. It has 23 DOF (Degree of Freedom) as well as other sensors and hardware elements.

UXA90 has an embedded PC and Linux Lubuntu system, which will be explained in this manual. The robot software platform ROS is included.

ROS is a robot software platform from Open Source Robotics Foundation (www.ros.org), and it provides debugging tools for hardware device control, data transfer between control processes, multiple process management, library support for development, and robot development.

UXA90 currently uses Linux Lubuntu 14.04 and ROS Indigo.

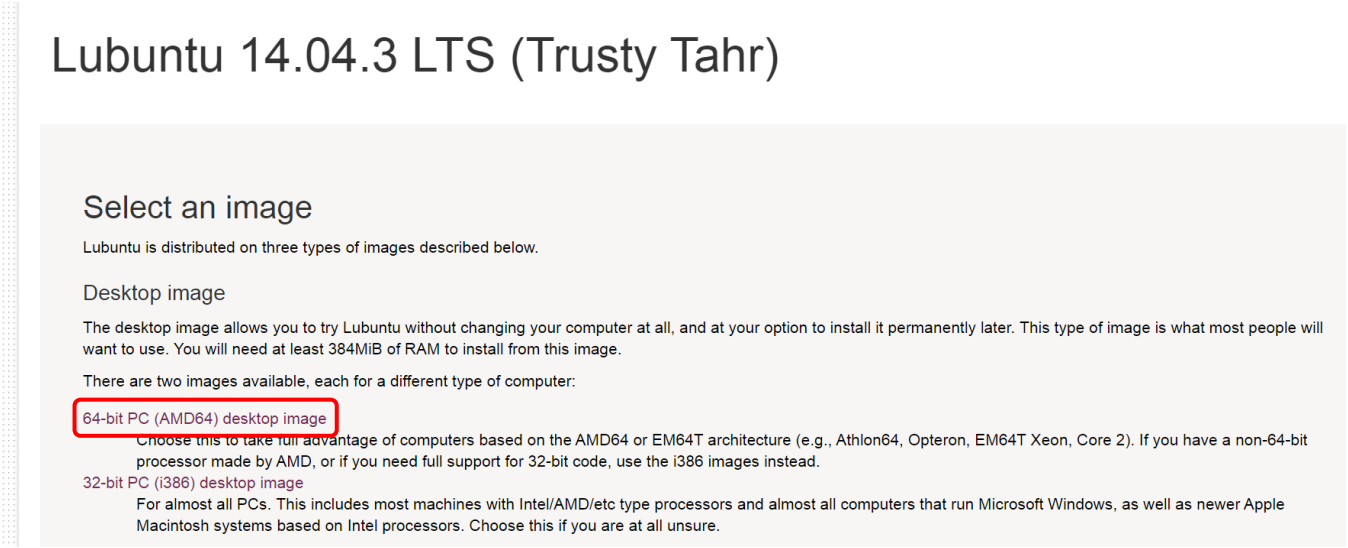
Lubuntu installation

This chapter explains how to install Lubuntu on UXA-90's internal computer, ex) ZBOX.

First, download Lubuntu iso file. Download the Lubuntu-14.04-LTS-x86-AMD64 version to install.

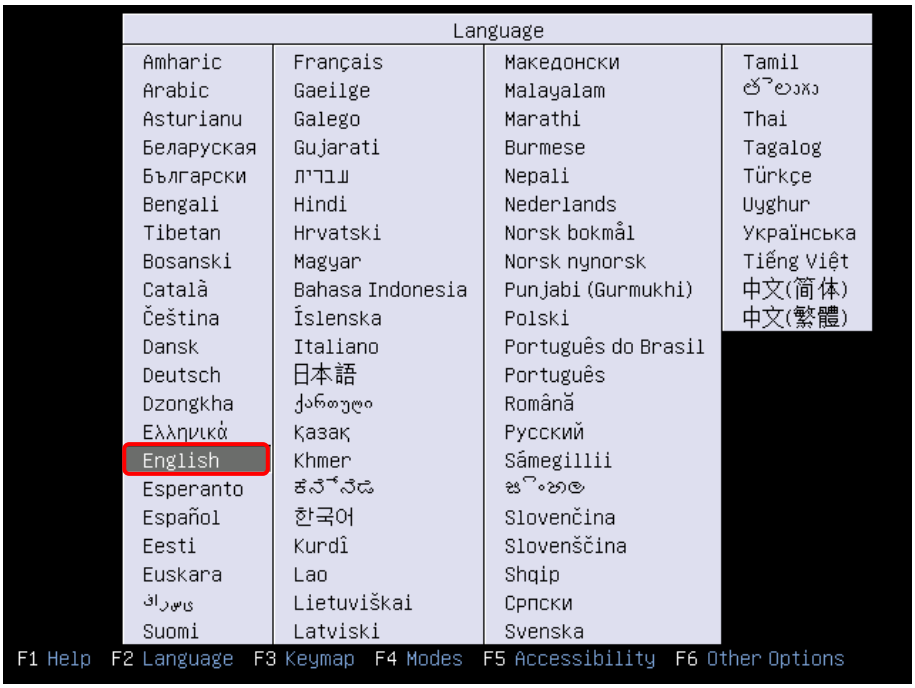
http://cdimage.ubuntu.com/lubuntu/releases/14.04/release/

Follow the link for free download. See <Figure 1-1> for instruction.



<Figure 1-1>

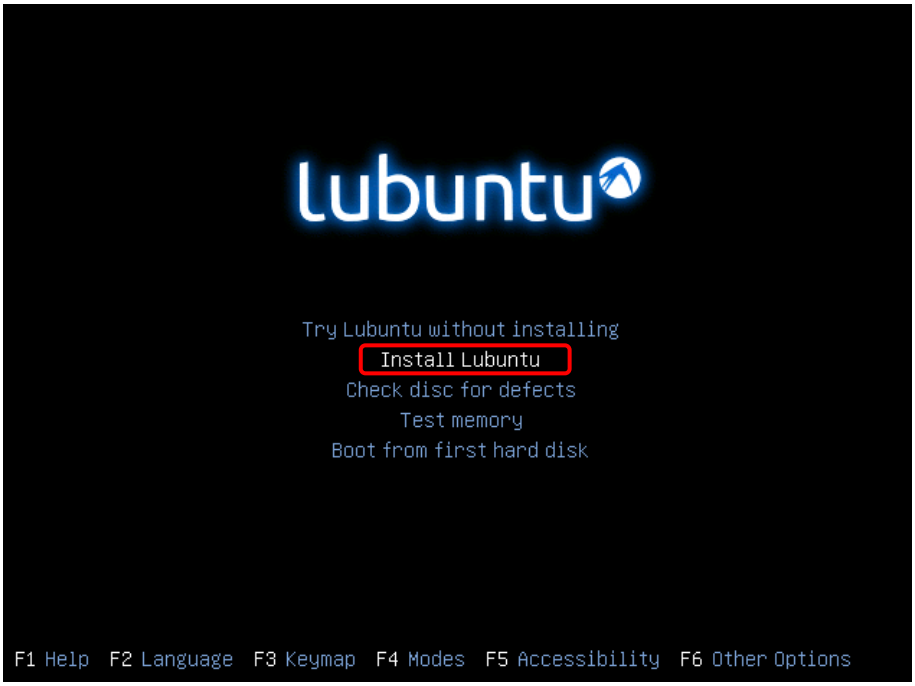
Use the iso file to install. Start the computer and you will see the screen as shown in <Figure 1-2>. Select the language as shown in the language in <Figure 1-3>. This manual will choose English.



<Figure 1-2>

After the language has been selected, the screen will look like <Figure 1-3>. You can immediately start 'Install Lubuntu' or check if there is enough memory by choosing 'memory test'.

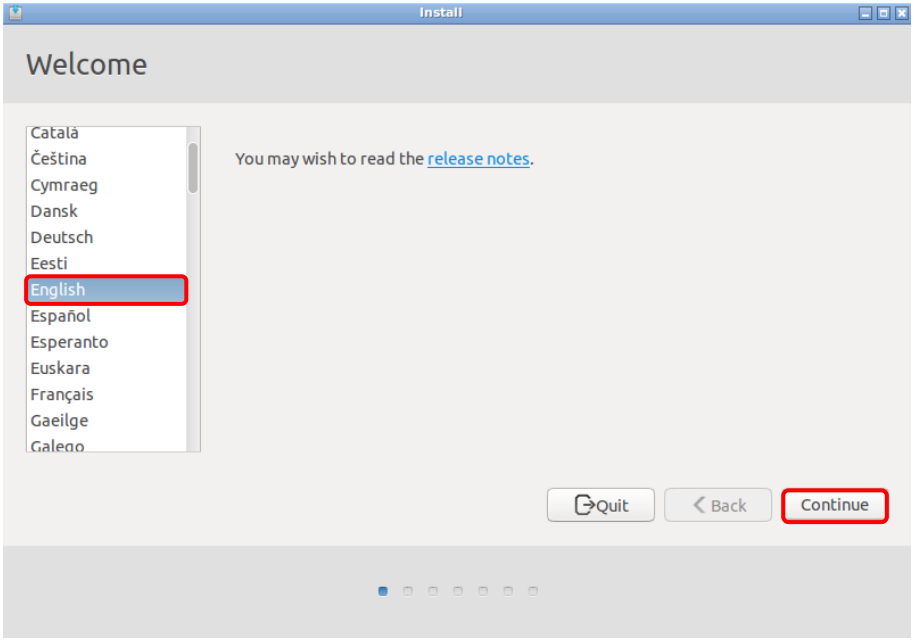
Choosing 'Install Lubuntu' will lead to the screen shown in <Figure 1-4> and then to installation.



<Figure 1-3>

<Figure 1-4>

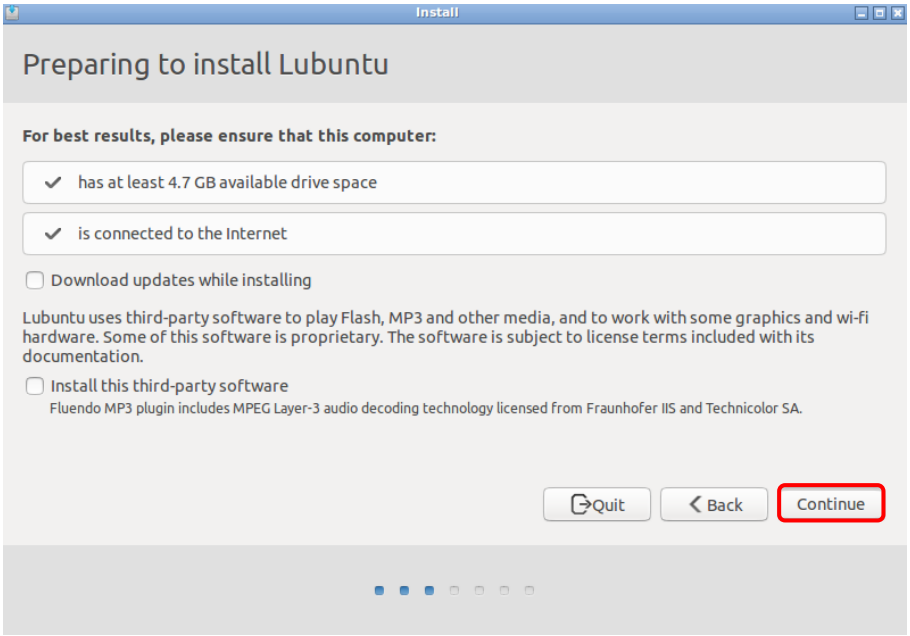
Now, setting and installation. First, select the language for the OS. It will set the default language. Select the language you are most comfortable with. Here, the manual will select English again. See <Figure 1-5> for instruction on language selection.



<Figure 1-5>

After the language has been selected, a window confirming the possibility of Lubuntu installation will appear. The drive should have at least 4.7GB available and the computer should be connected to internet.

Do not check 'Download update while installing' as we will update after the installation has been completed. See <Figure 1-6>.



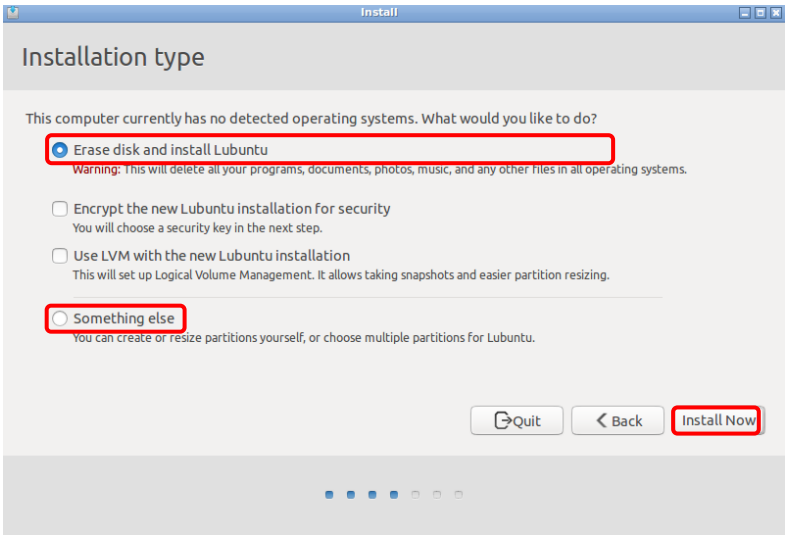
<Figure 1-6>

Next is selecting how to install Lubuntu OS on the disk.

<Figure 1-7> shows two options: 'Erase disk and install Lubuntu' and 'Something else'.

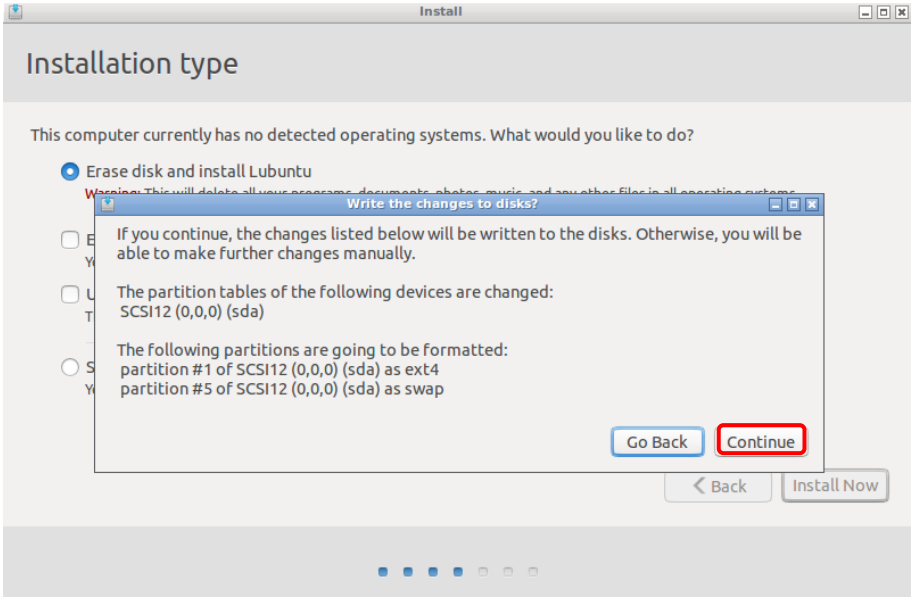
'Erase disk and install Lubuntu' automatically selects without creating new partition or adjusting the size of partition. If it is unnecessary to set a separate partition, it is recommended that you select the first option. If this option has been selected, you will see <Figure 1-7-1> on your screen.

'Something else' allows the user to assign partitions and swap memory space as desired. See <Figure 1-7-2>.



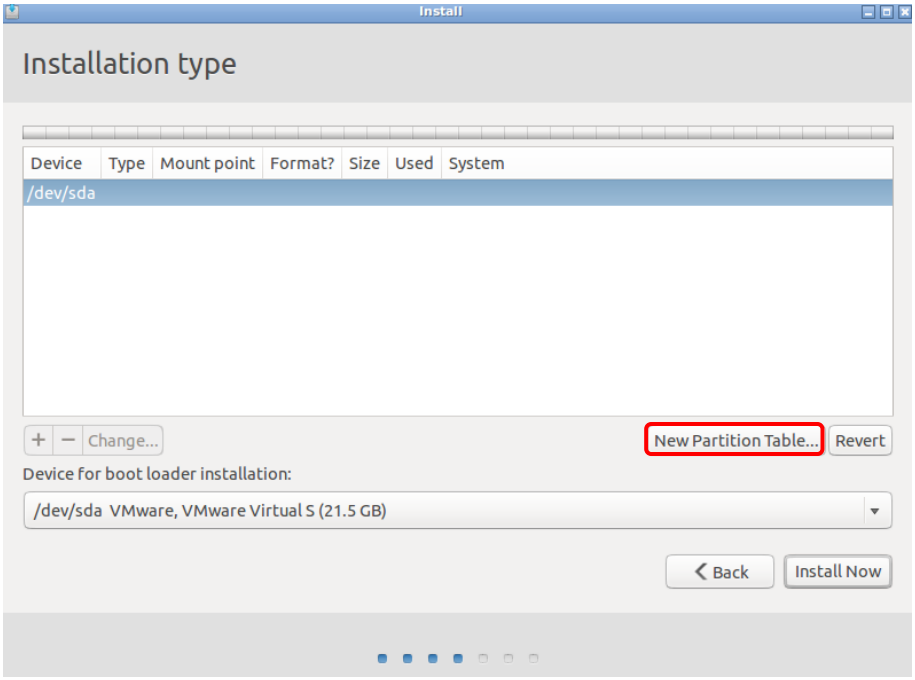
<Figure 1-7>

When 'Erase disk and install Lubuntu' is selected, the screen will look like <Figure 1-7-1>. It is to approve partition setting and format. Click 'Continue' to go to the next step.



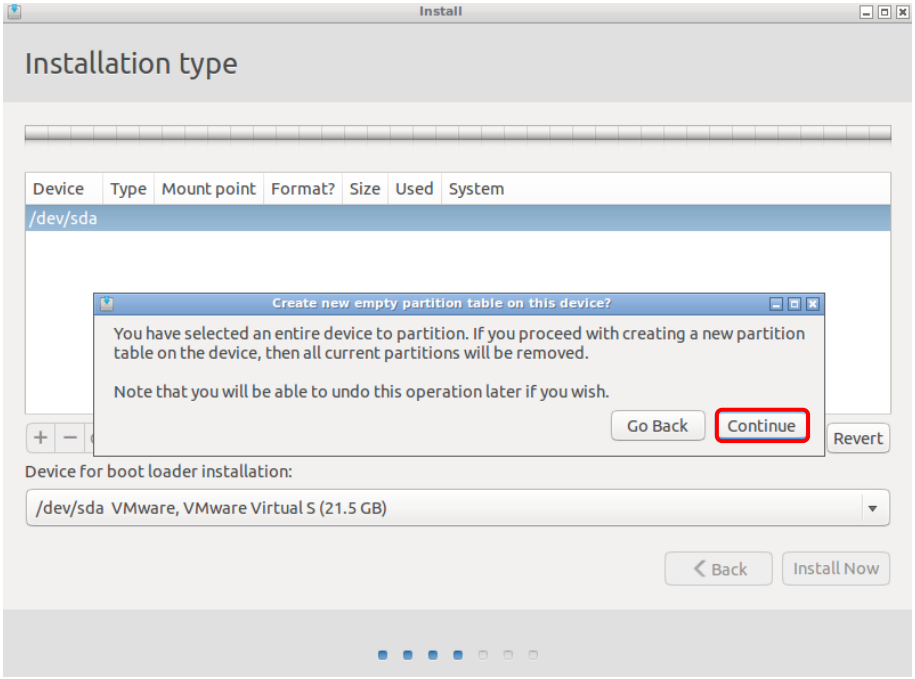
<Figure 1-7-1>

When ‘Something else’ is selected, the pop-up in <Figure 1-7-2> will appear. It is to divide the partition on the disk. Click ‘New Partition Table…’ to set partitions.



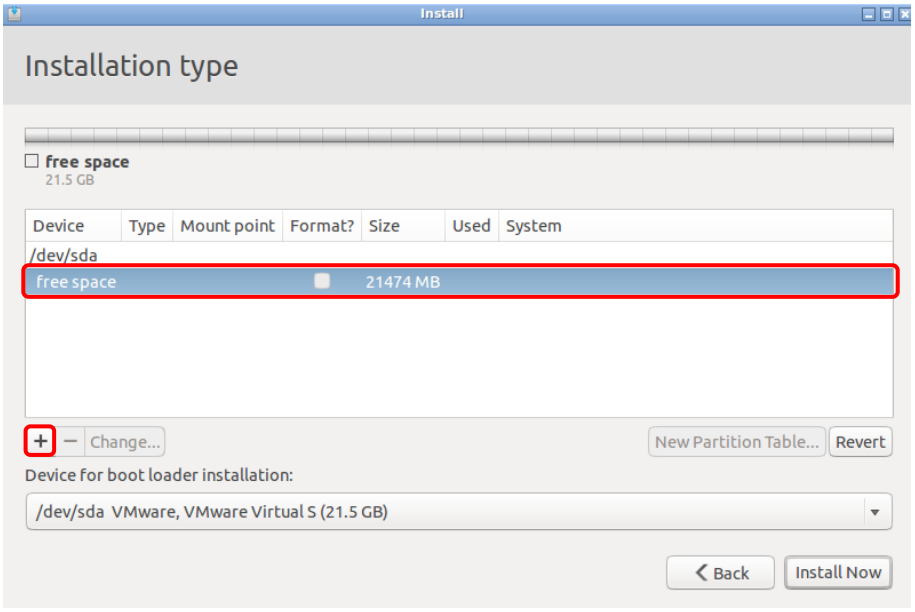
<Figure 1-7-2>

When ‘New Partition Table…’ is clicked the pop-up in <Figure 1-7-3> appears. Click ‘Continue’ to create a new partition table.



<Figure 1-7-3>

Select ‘free space’ and click ‘+’ button to enter the window to create partitions.

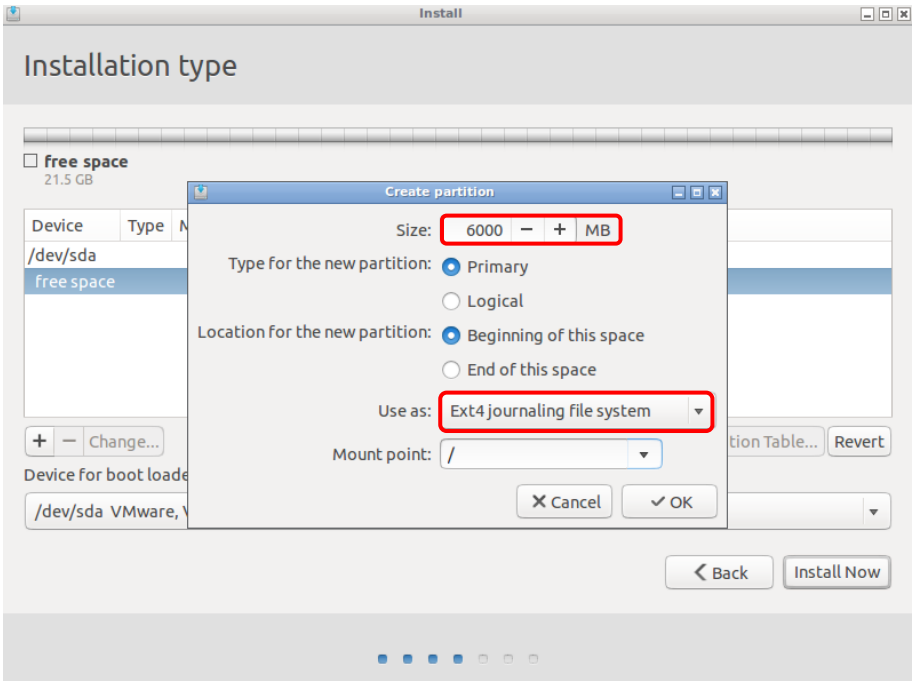


<Figure 1-7-4>

When ‘Create partition’ window appears, set the followings to create partitions. First, set the size and type. Size can be set as desired, but there should be 2 times the RAM available on PC (ZBOX). See the details in the next image.

Select ‘EXT4 journaling file system’ for ‘Use as’ to create a space to save data.

Click ‘OK’ when finished.

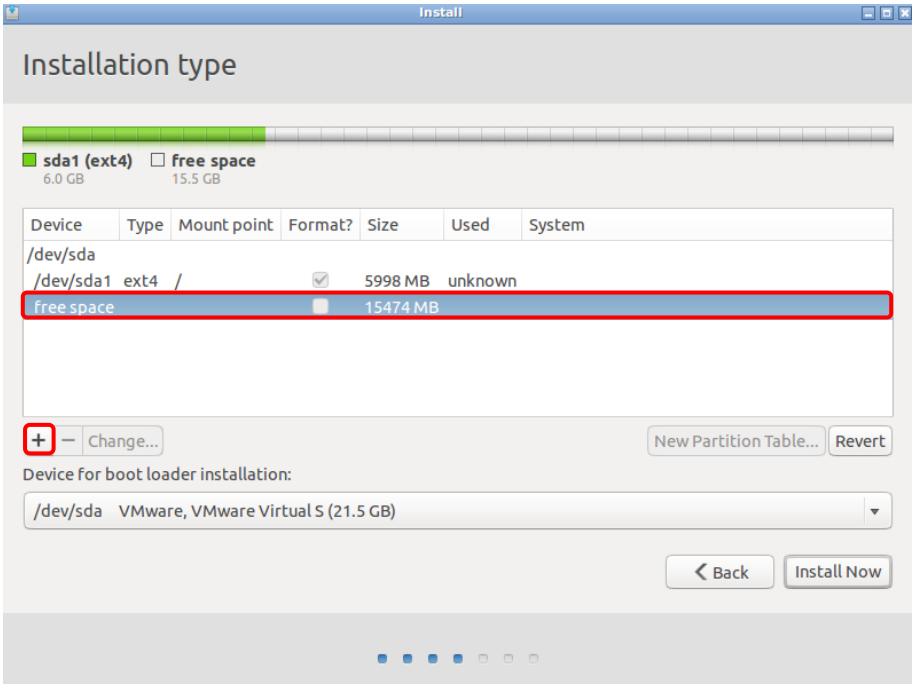


<Figure 1-7-5>

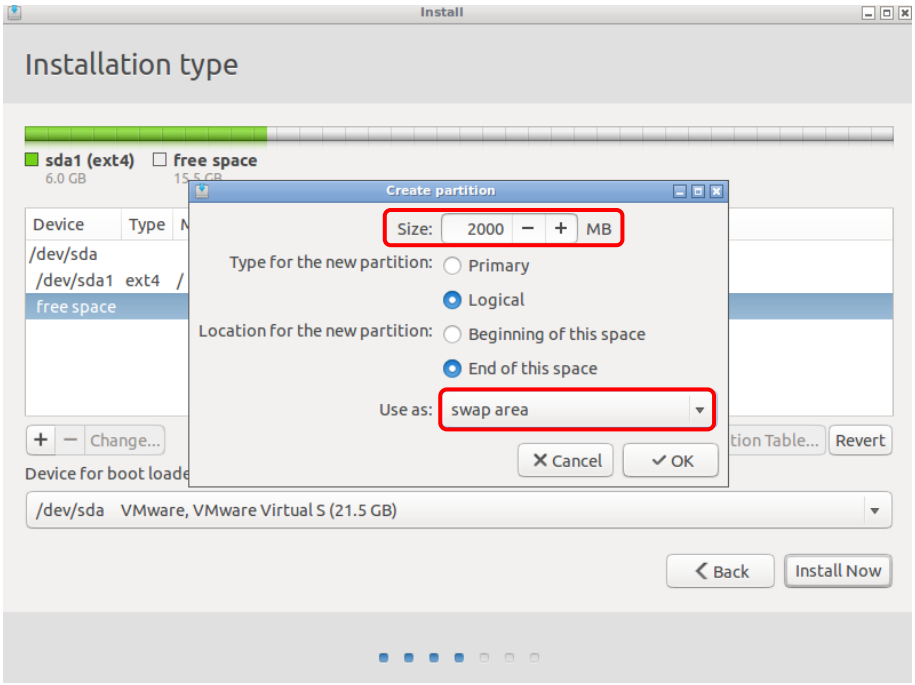
Now, create the swap memory space.

Unlike Windows, Linux shuts down if the memory space is full. To prevent this from happening, use hard disk as memory. And this space is called swap memory. To set aside a spare space, assign twice the size of actual memory (RAM).

As shown below, assign 2GB for the swap space.



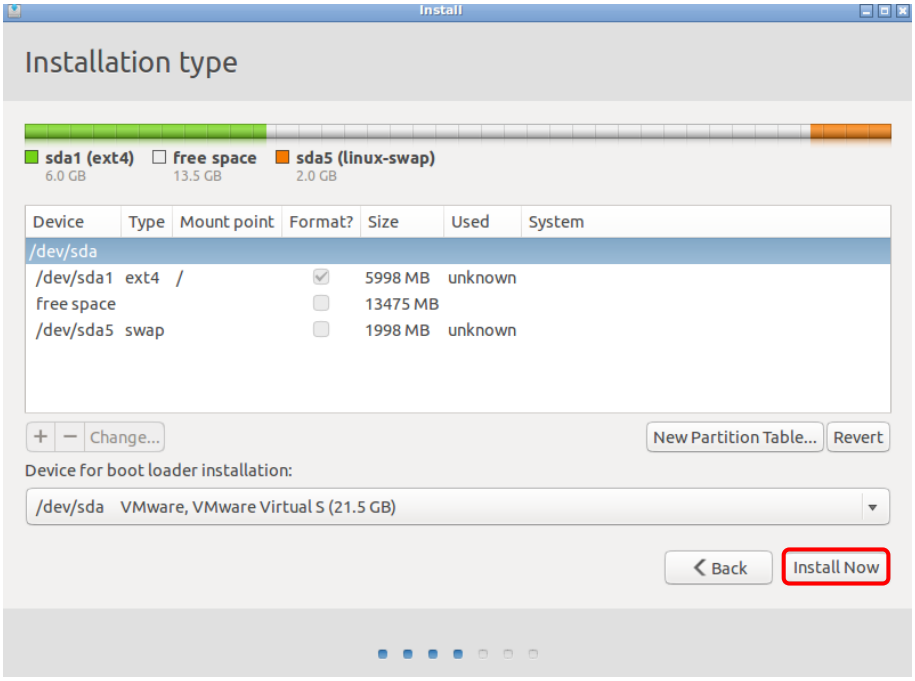
<Figure 1-7-6>



<Figure 1-7-7>

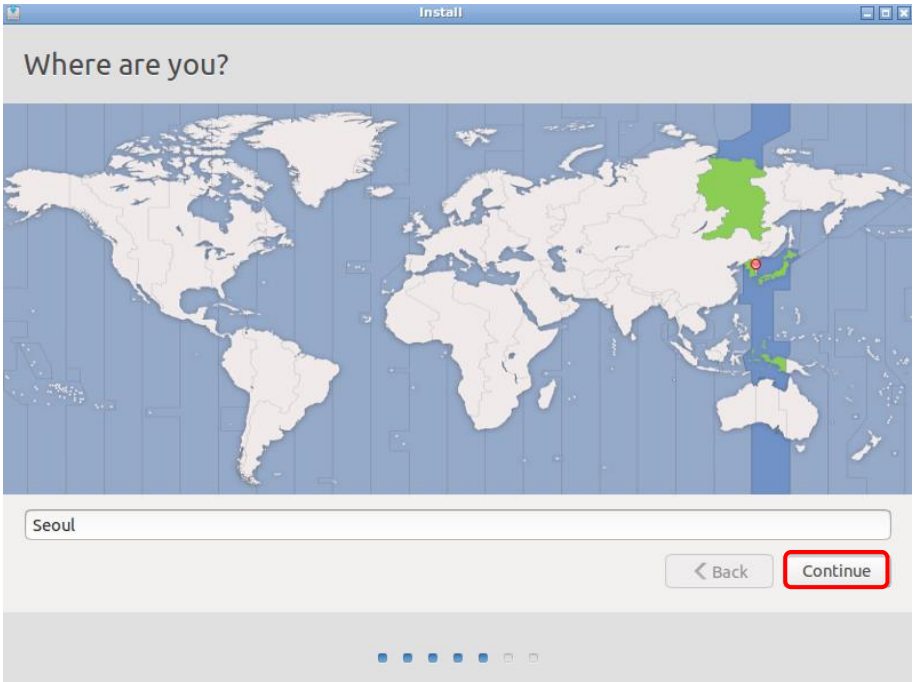
Once memory setting is done, <Figure 1-7-8> appears on the screen. In the free space shown in <Figure 1-7-8>, you can make a new partition or leave as is and edit later.

Once finished, start installation by clicking 'Install Now'.



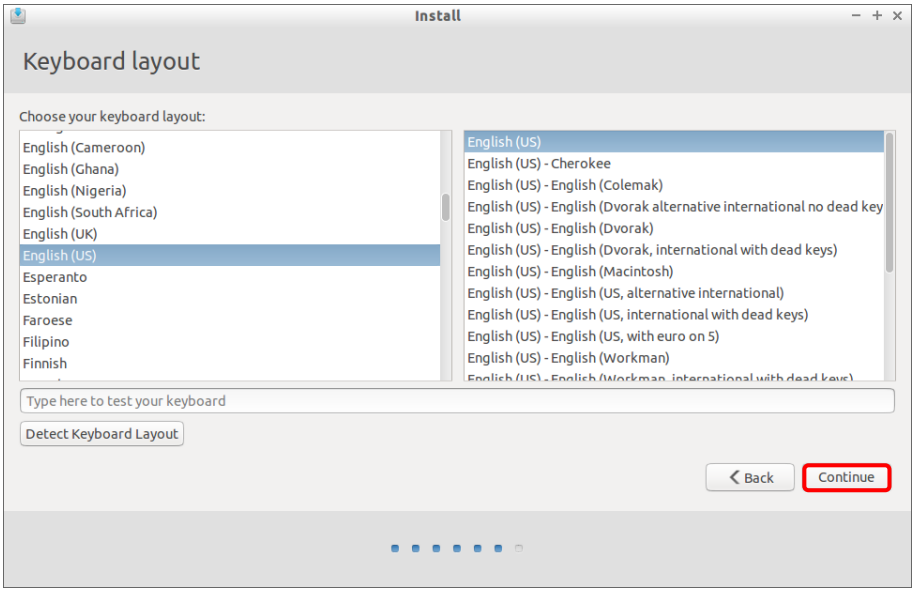
<Figure 1-7-8>

Once memory setting is done, set the location to set the time and date. See <Figure 1-8>.



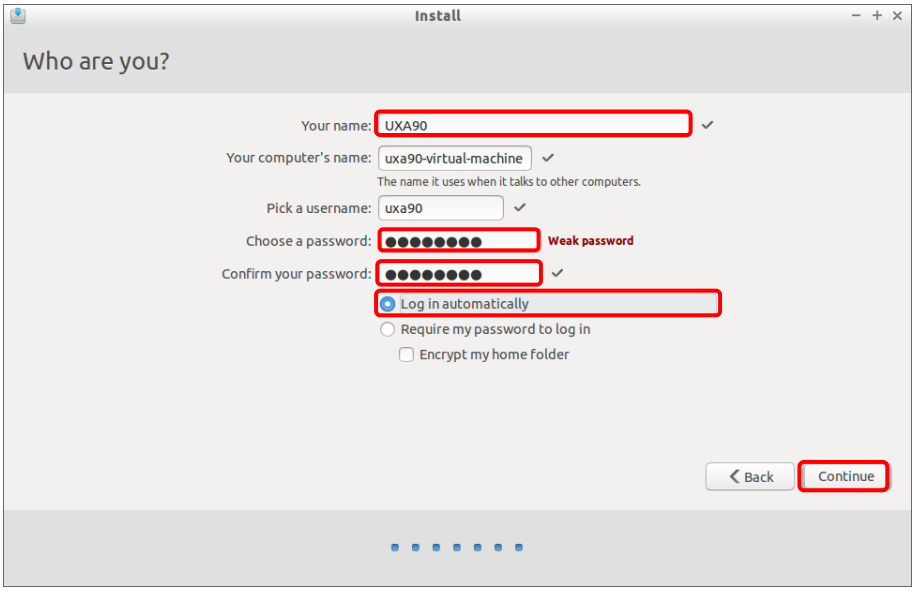
<Figure 1-8>

Next, set the keyboard layout. Test the keyboard and choose the layout.



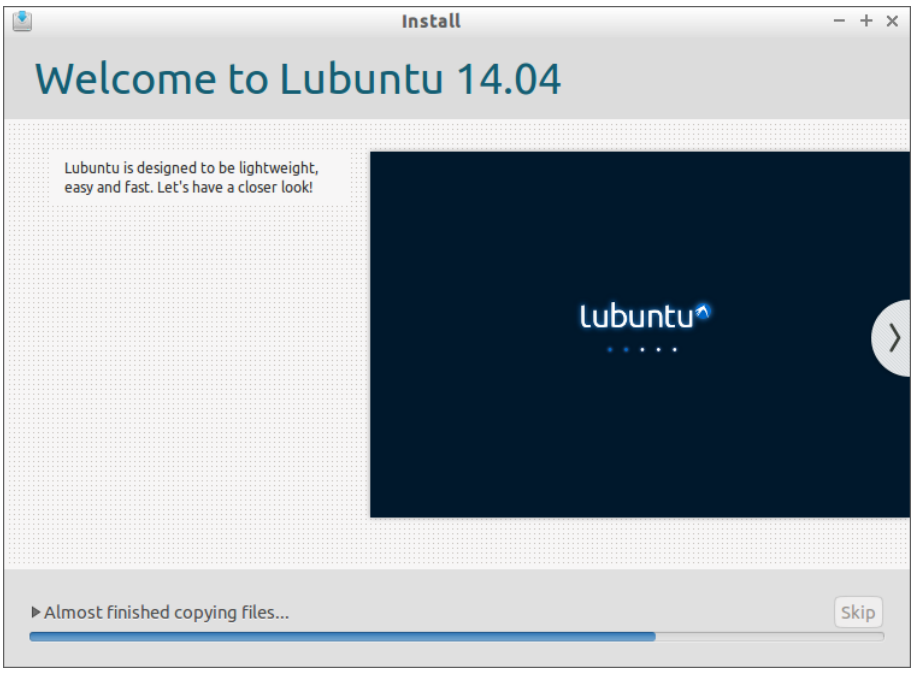
<Figure 1-9>

Lastly, assign the user. Assign a username and a password. Password will be used frequently in Lubuntu, so make sure to remember it well. Also, checking 'Log in automatically' will allow auto-log in when booting.

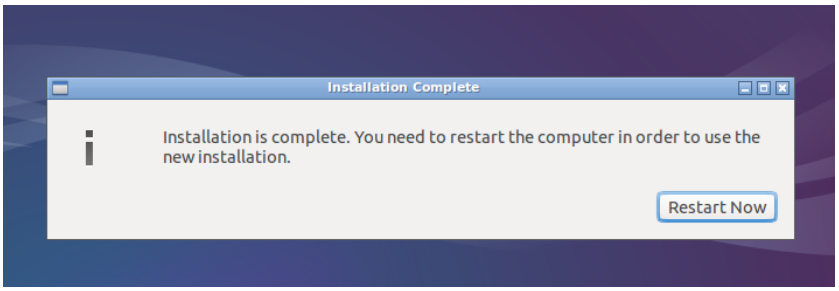


<Figure 1-10>

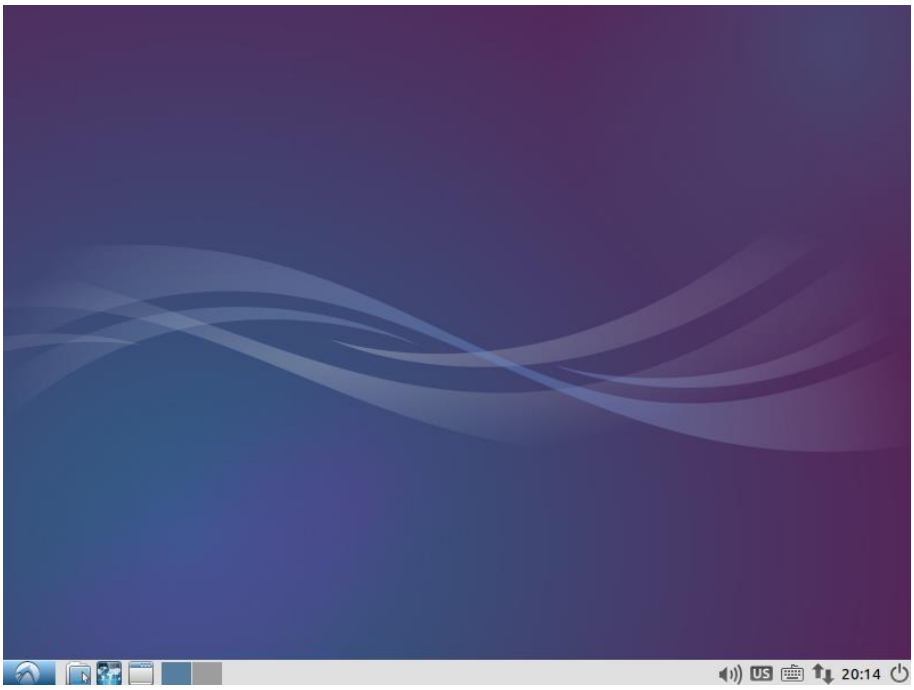
Once all setting has been done, the installation begins as shown in <Figure 1-11>. When completed, press the Restart Now button as shown in <Figure 1-12>. After rebooting, you will see the Lubuntu background as in <Figure 1-13>. Lubuntu has been successfully installed.



<Figure 1-11>



<Figure 1-12>



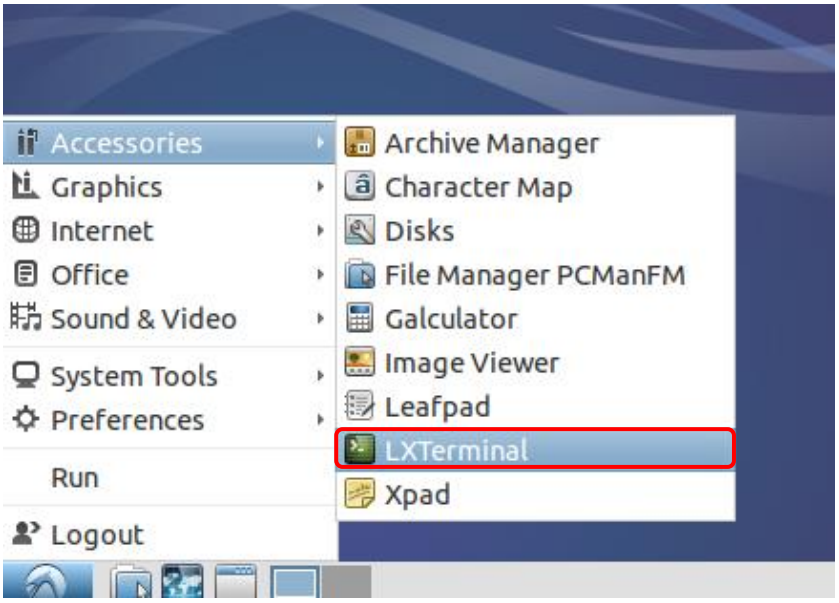
<Figure 1-13>

1. Lubuntu initial setting

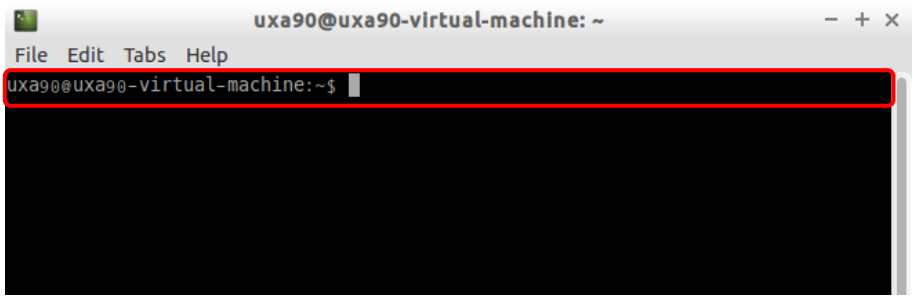
This chapter is on Lubuntu initial setting and environment setting.

First, run the terminal. As in <Figure 2-1>, you can first run 'LXTerminal' or press 'Ctrl' + 'Alt' + 't' to start. Once terminal is run, see that the program starts as shown in <Figure 2-2>.

Terminal allows the user to use a command to start an action. As <Figure 2-2>, you will see 'uxa90@uxa90-virtual-machine:~$'. 'uxa90@uxa90-virtual-machine' is the information on current user and the device, and '~$' refers to the user authorization and terminal's location. '~' is the home location of the system which is '/home/uxa90', and '$' refers to a general user. Here, root user authority (similar as admin in Windows) is described with '#'.



<Figure 2-1>

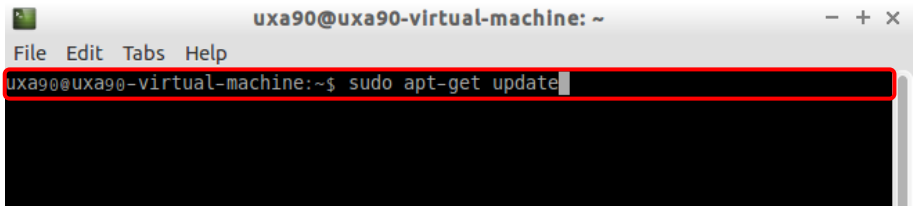


<Figure 2-2>

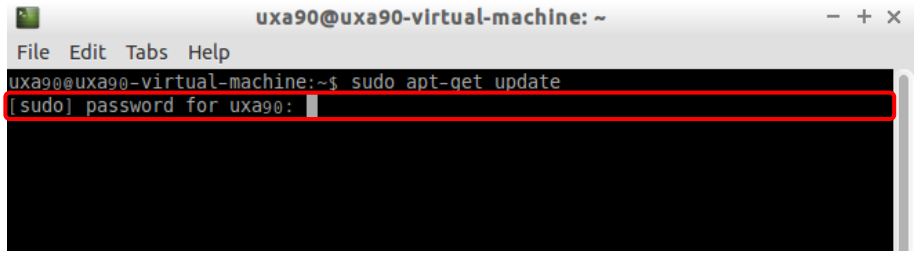
Using the command below in the terminal as shown in <Figure 2-3>, the part that needs an update is automatically updated. Usually the update is regarding the package.

$ sudo apt-get update

Sudo command is a command that temporarily borrows the root authority. To update the package, the package information needs to be modified, and it requires root authority. Therefore, it will require the password as shown in <Figure 2-4>. Enter the password created in Chapter1 and press 'Enter'.

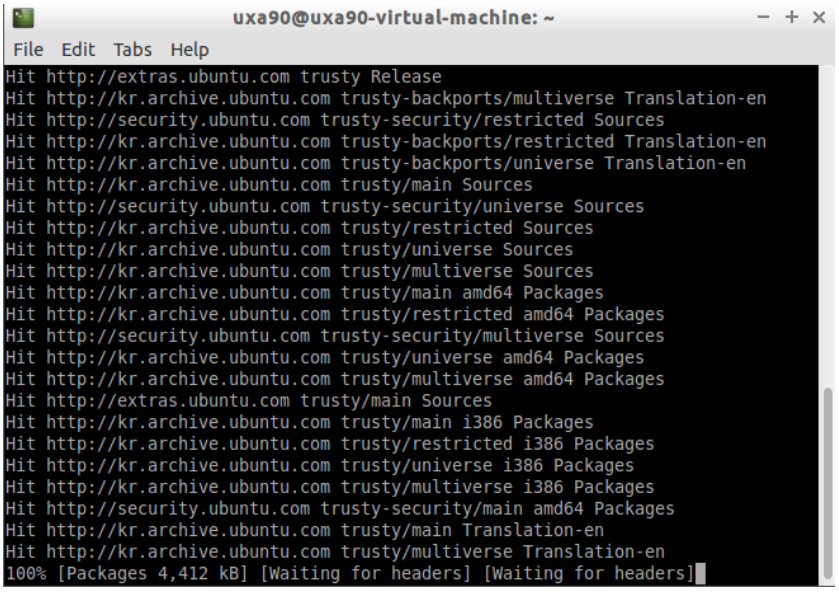


<Figure 2-3>



<Figure 2-4>

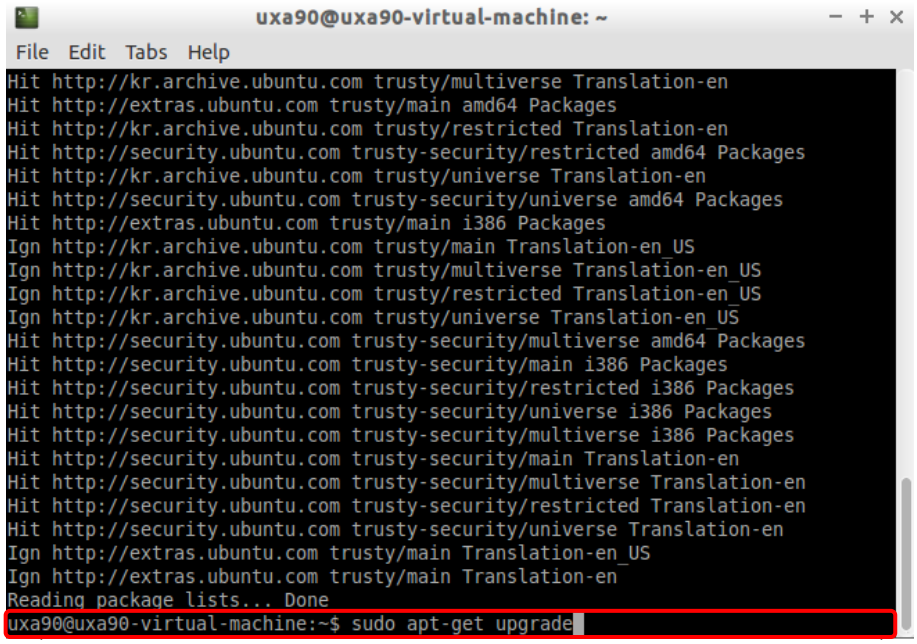
Once the password has been entered, the package information is automatically entered as in <Figure 2-5>, and those needing an update are automatically updated.



<Figure 2-5>

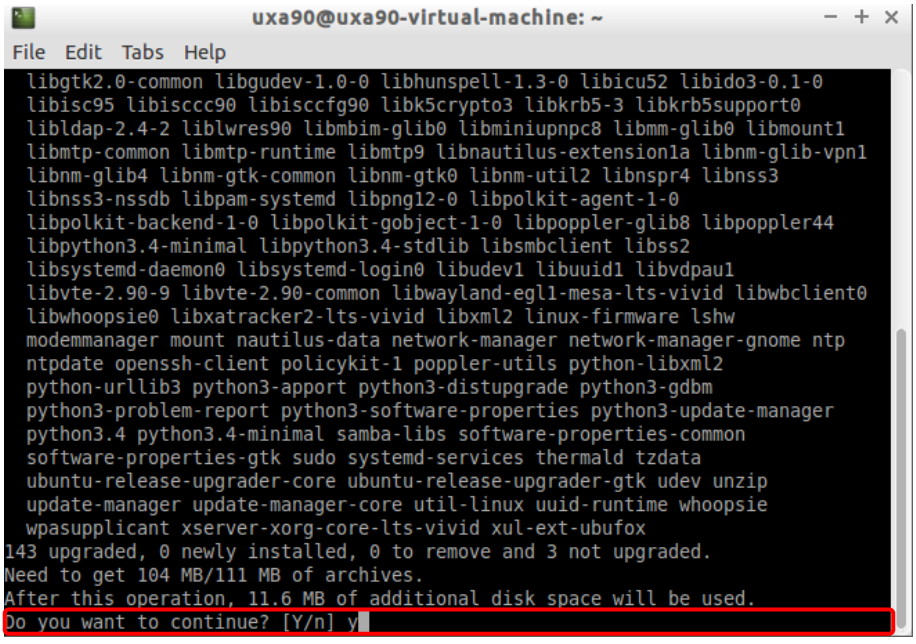
Once package update has been finished, run the system upgrade. Package upgrade is always coupled with updates. The command is:

$ sudo apt-get upgrade



<Figure 2-6>

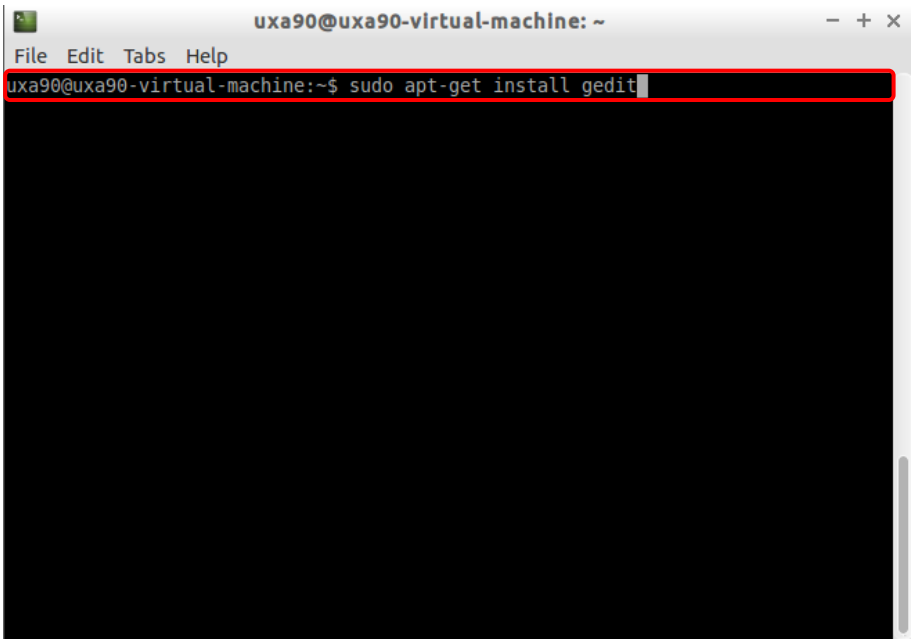
When you enter the command as shown in <Figure 2-6>, a list of package that needs an upgrade appears as in <Figure 2-7>, and it asks for permission. Enter 'y' to continue with the package upgrade and installation.



<Figure 2-7>

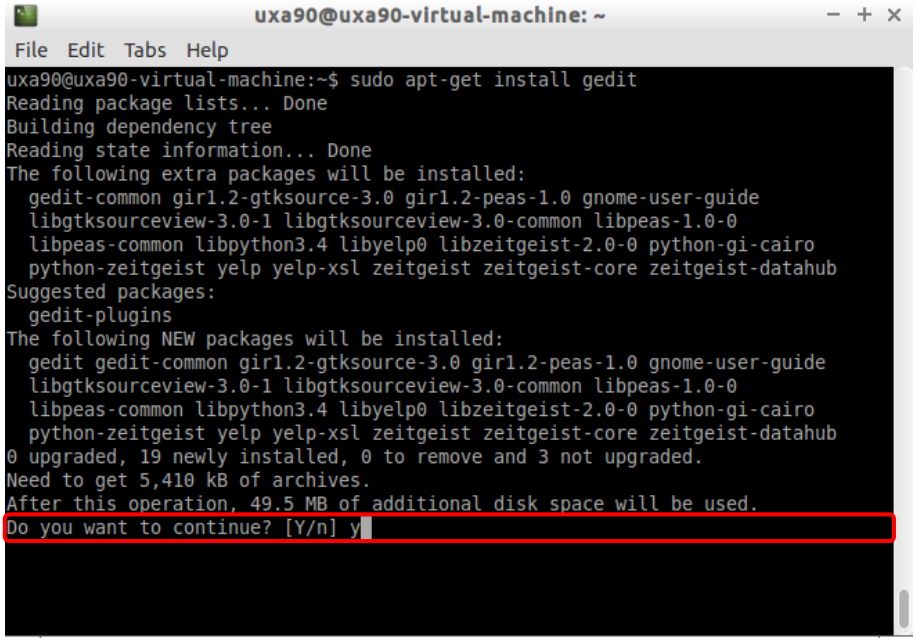
Once the installation has been completed, the initial setting has been done. Next, use the 'apt-get' command to install the person. You can easily install the package if you know the name. Try installing 'gedit' package. 'gedit' is a text file application like Notes in Windows. Use this command:

$ sudo apt-get install gedit



<Figure 2-8>

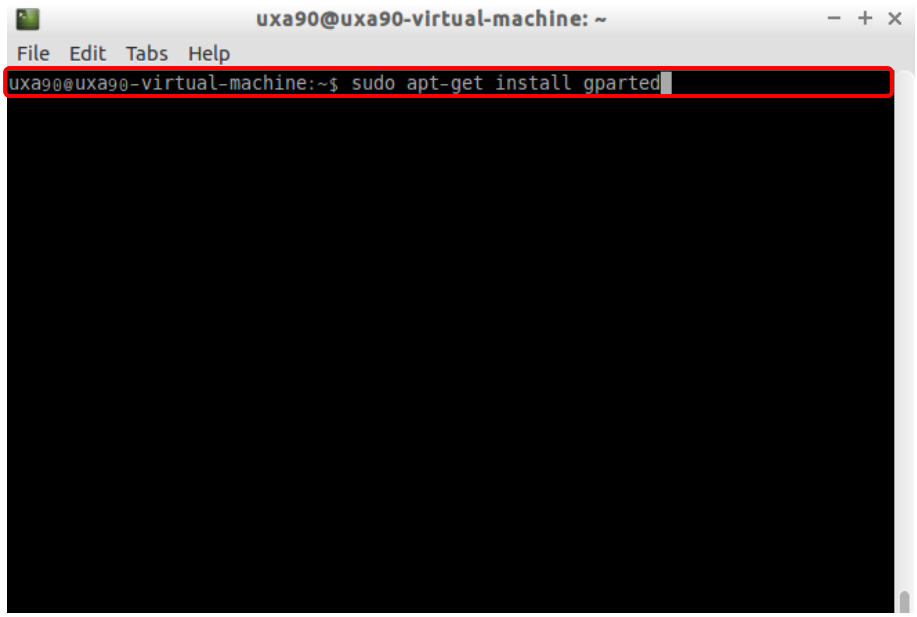
Installation requires permission just like for the upgrade. Enter 'y' to accept.



<Figure 2-9>

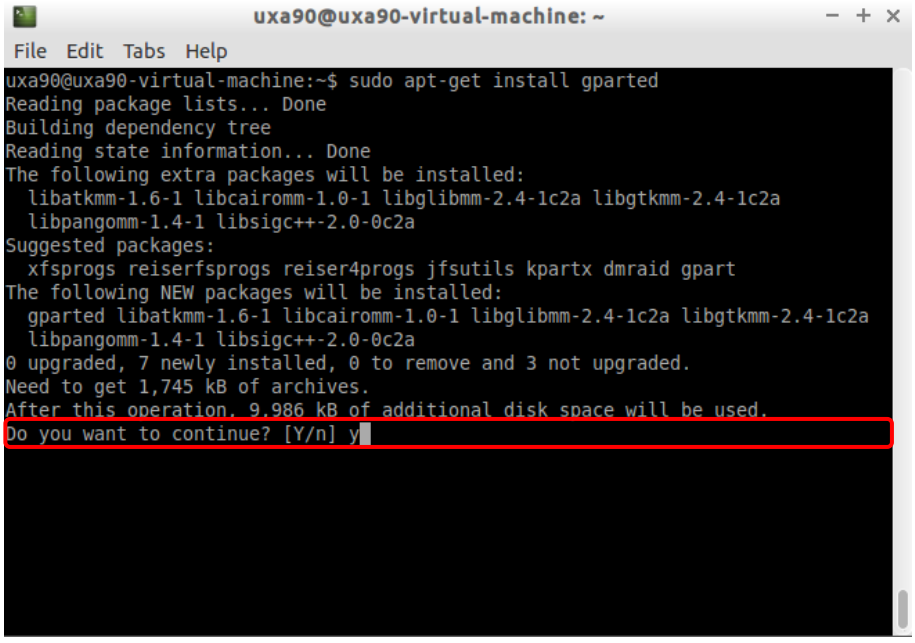
Once 'gedit' installation is done, try installing another application. The application 'gparted' manages disk space. Use this command:

$ sudo apt-get install gparted



<Figure 2-10>

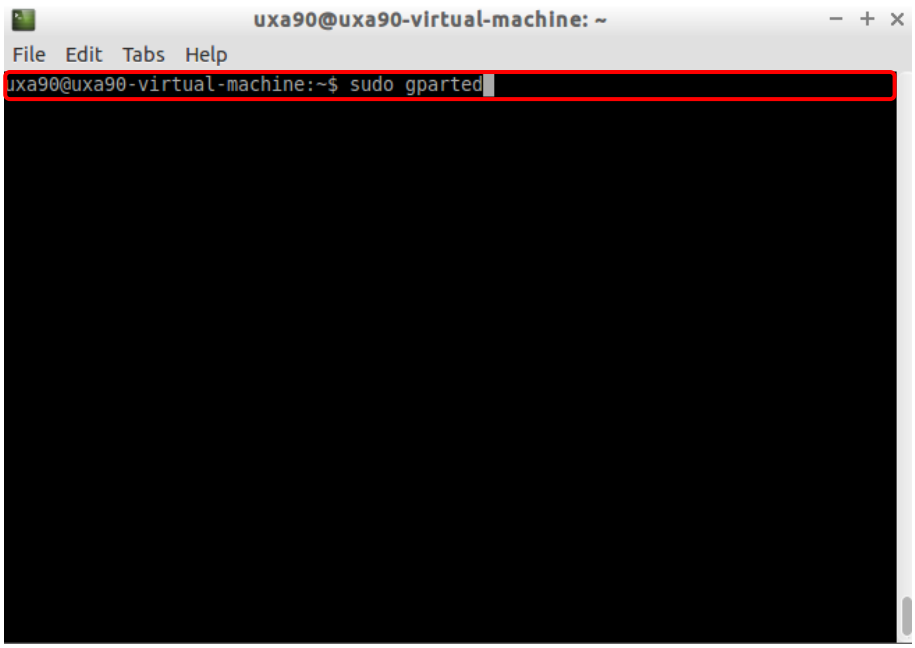
It also requires permission. Enter 'y' to continue.



<Figure 2-11>

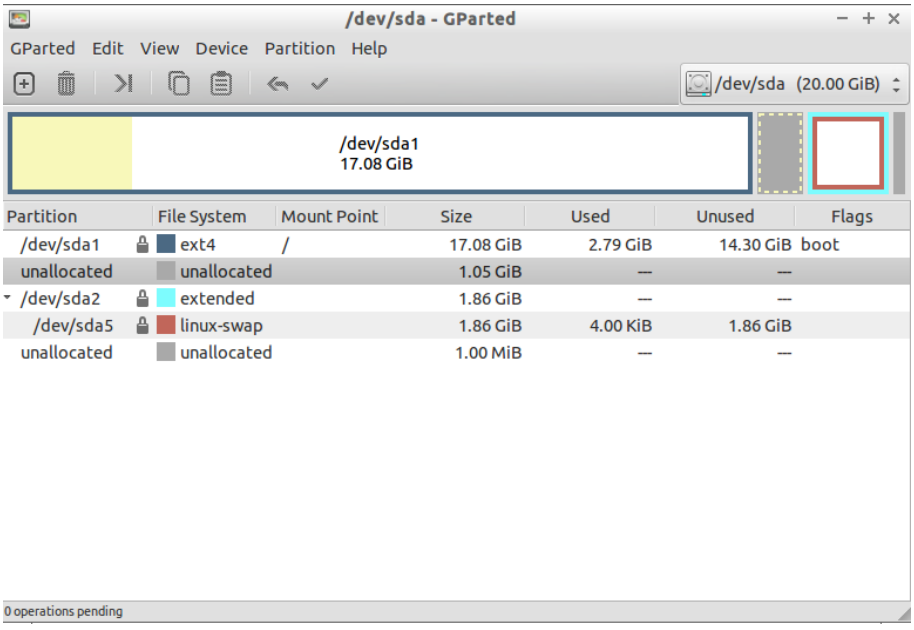
Once completed, run the program. 'gparted' needs to read the disk data, so it requires root authority. Enter the command below to run.

$ sudo gparted



<Figure 2-12>

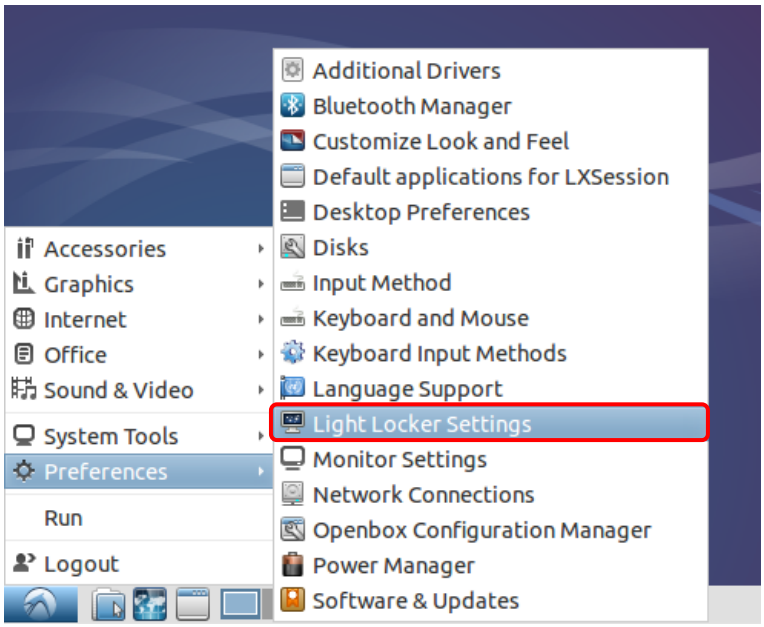
'Gparted' shows disk and partition size and information, and it also allows modifying them. Refer to the program manual for detailed information.



<Figure 2-13>

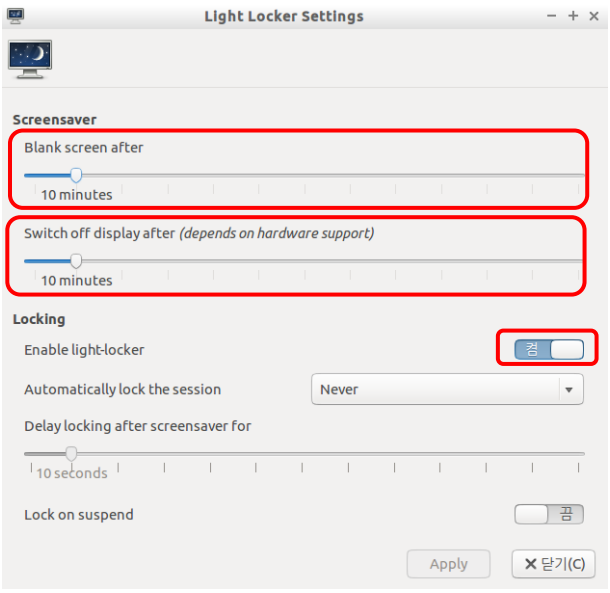
Lastly, set the screensaver. In default, Ubuntu assigns a short time before entering the screensaver mode, so to prevent inconvenience later, set the time now.

Start Light Locker Settings as shown in <Figure 2-14>.

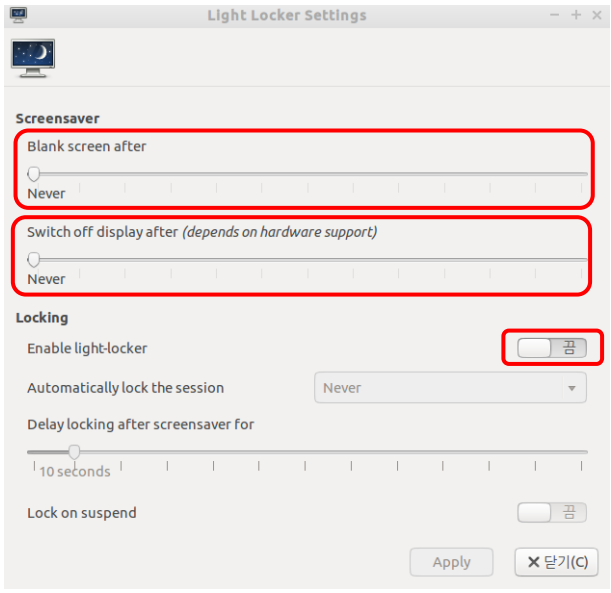


<Figure 2-14>

The window in <Figure 2-15> will appear. Change the setting as shown in <Figure 2-16> and turn off the screensaver. Change the time as desired.



<Figure 2-15>



<Figure 2-16>

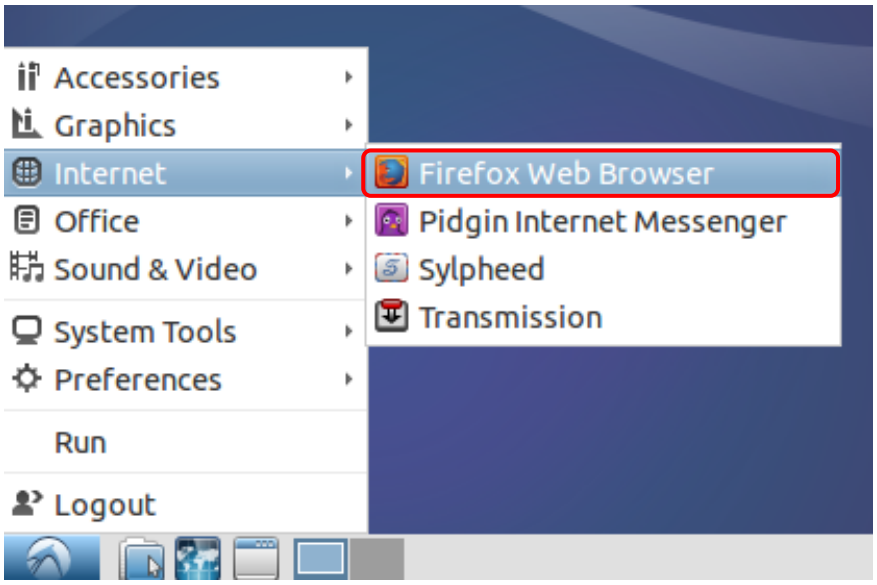
2. ROS installation

This chapter shows how to install ROS Indogo on Lubuntu 14.04.

ROS is different from other OS's. It offers key functions required for robot application software in a library type. Visit 'http://www.ros.org/wiki/' for more information.

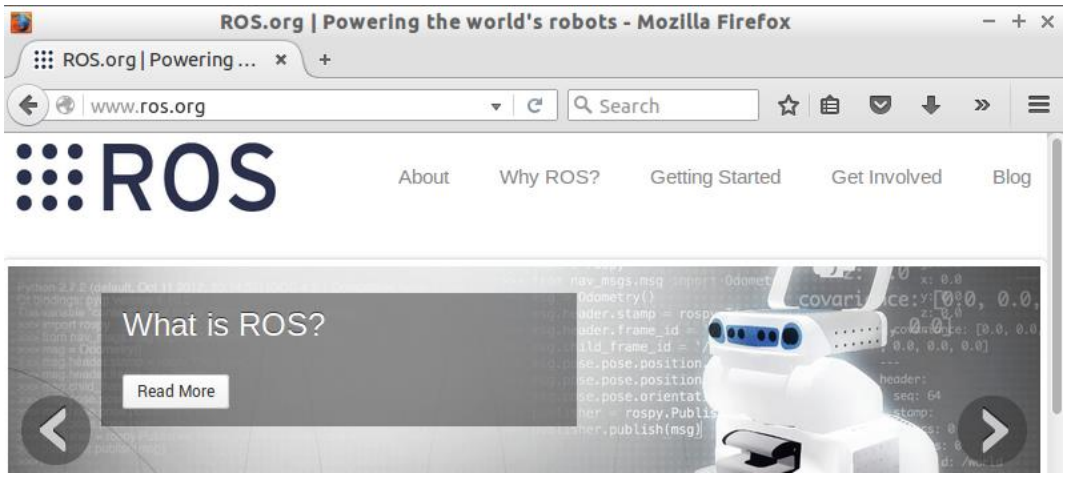
Since its release of ROS 1.0 in 2010, it has been releasing updates consistently. This manual uses Indigo Igloo version introduced in 2014.

First, start the web browser. Firefox Web Browser is pre-installed in Lubuntu. As shown in <Figure 3-1>, launch the web browser.



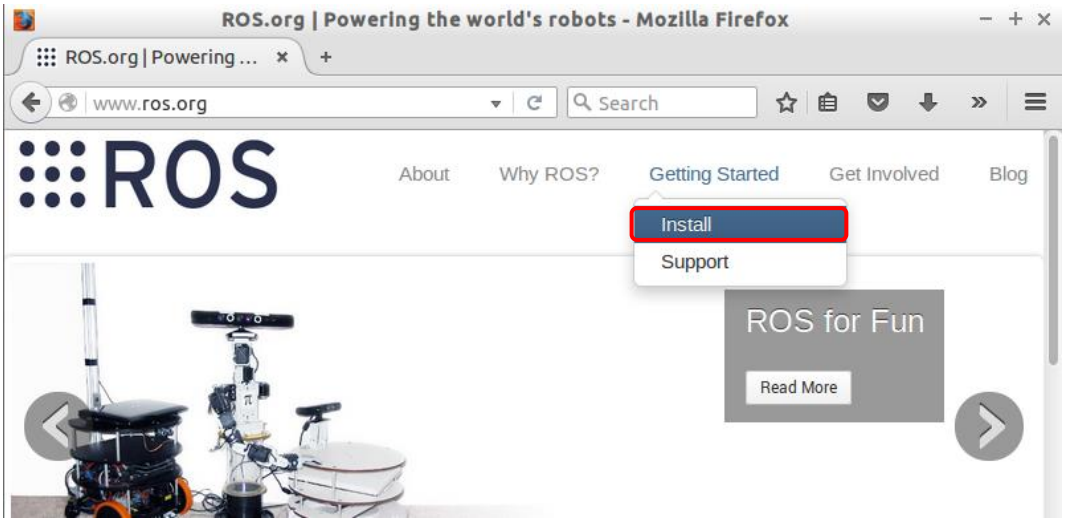
<Figure 3-1>

Connect to internet and enter 'http://www.ros.org'. You will see the website shown in <Figure 3-2>.



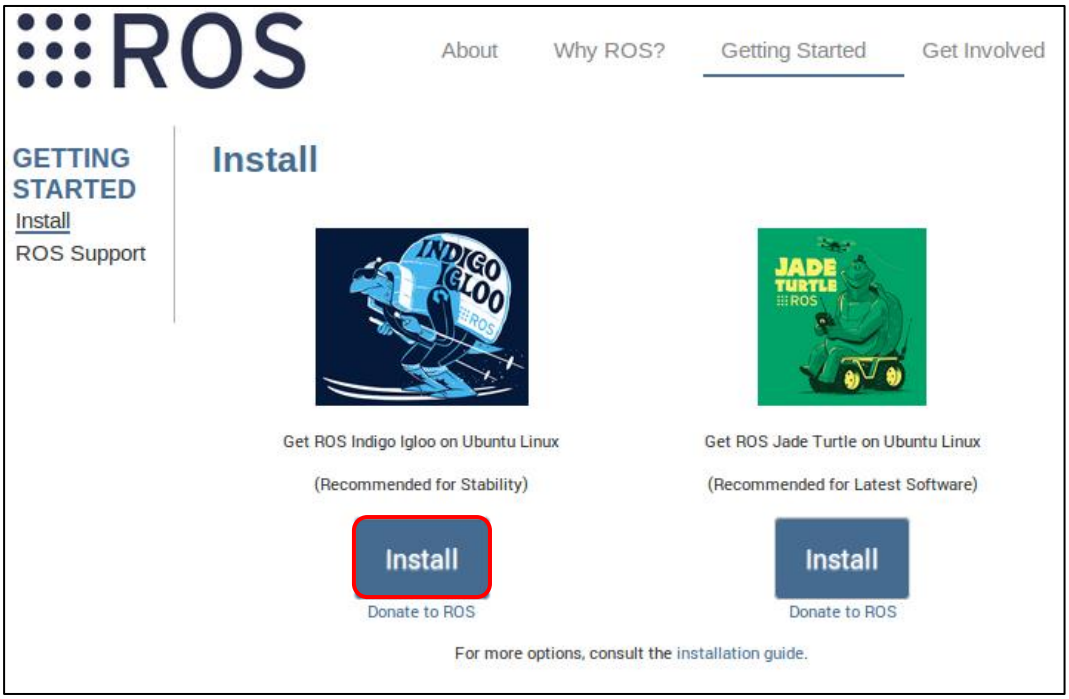
<Figure 3-2>

Once you enter, go to Install as shown in <Figure 3-3>.



<Figure 3-3>

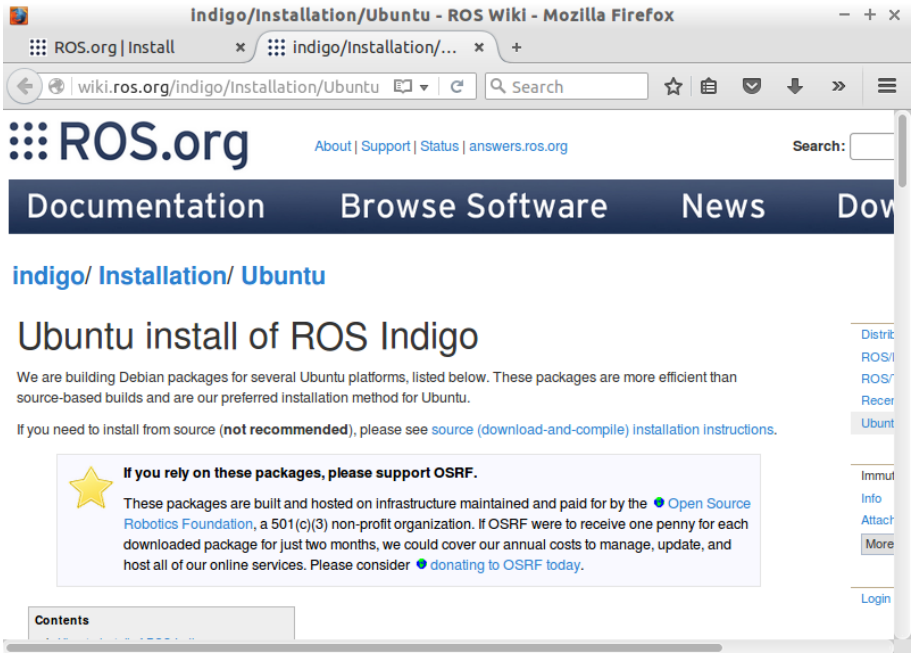
If the page shown in <Figure 3-4> appears, choose the version and click 'Install'. In this case, choose 'Install' under 'Indigo Igloo'.



<Figure 3-4>

Follow the order above, then you will see the page as in <Figure 3-5>. Follow the instruction on the page to finish the ROS installation.

Scroll down to start from the first step: '1. Installation'.



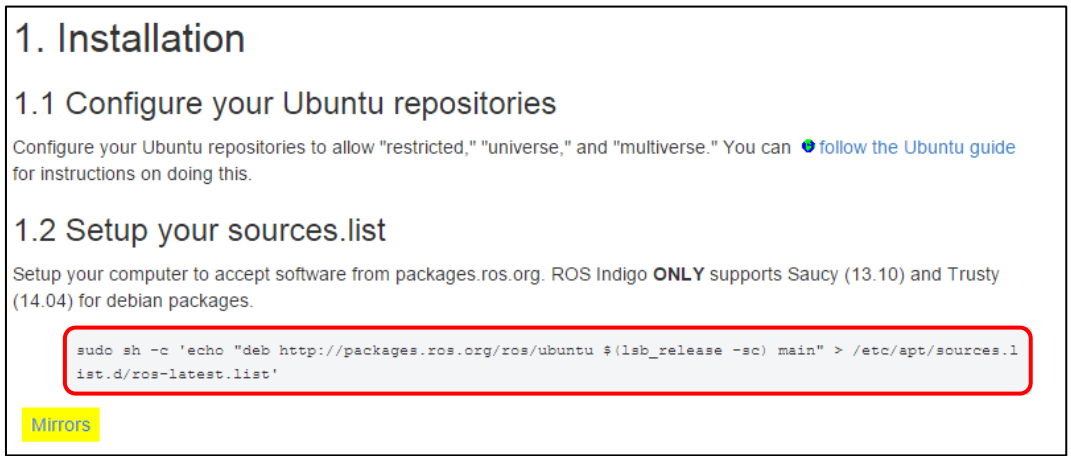
<Figure 3-5>

First, '1.1 Configure your Ubuntu repositories' sets the software repository of Lubuntu. This step can be skipped.

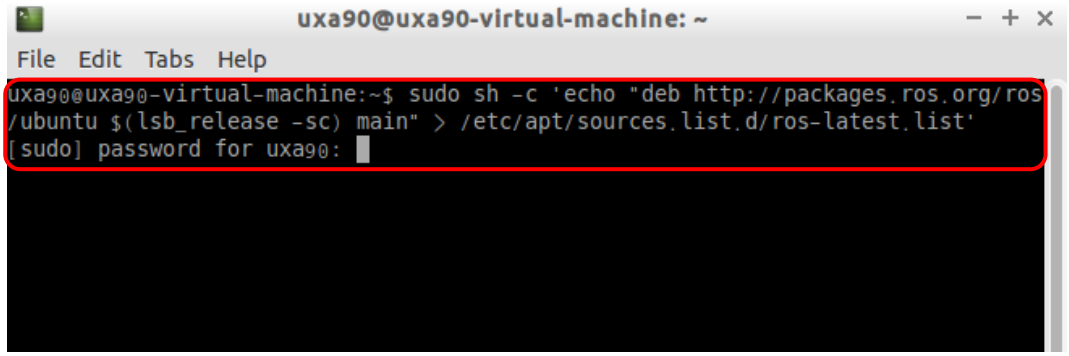
Go on to '1.2 Setup your sources.list'. Here, you can add a ROS repository to download a ROS-related package. Copy the part shown in <Figure 3- 6> to run the terminal and enter as shown in <Figure 3-7>.

Enter the password.

```
$ sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'
```



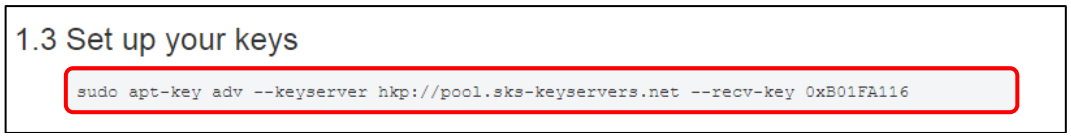
<Figure 3-6>



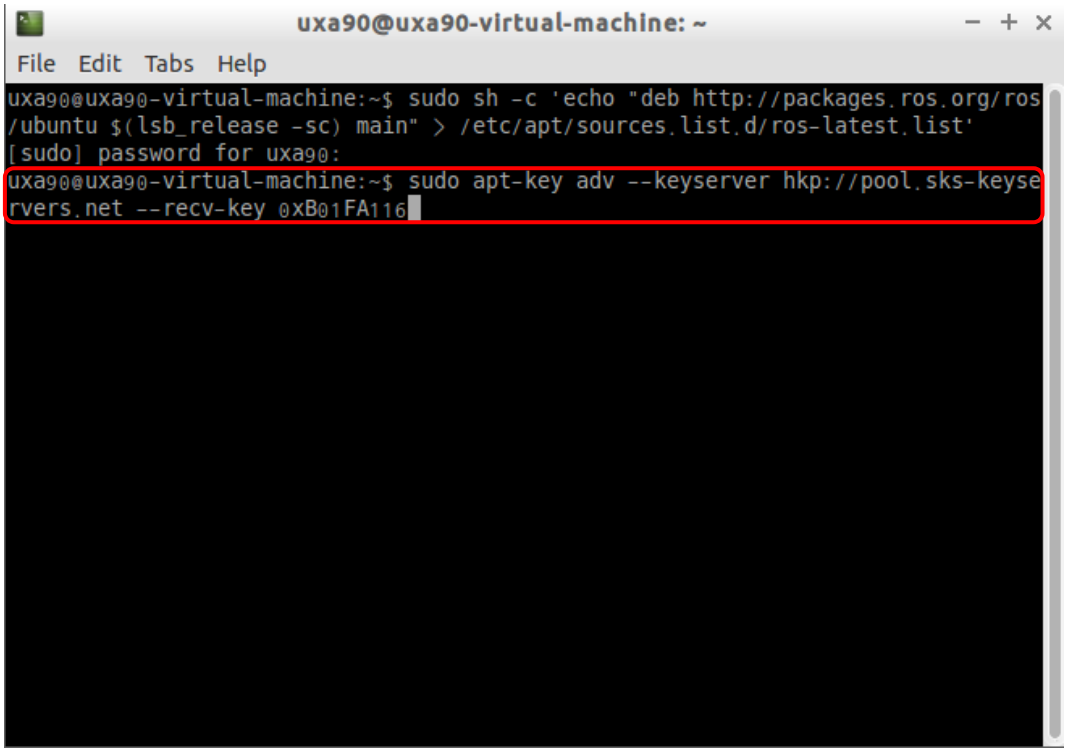
<Figure 3-7>

Next is '1.3 Set up your keys'. It is to add a key to download a ROS package. Copy the part in <Figure 3-8> and paste into the terminal.

$ sudo apt-key adv --keyserver hkp://pool.sks-keyservers.net --recv-key 0xB01FA116



<Figure 3-8>

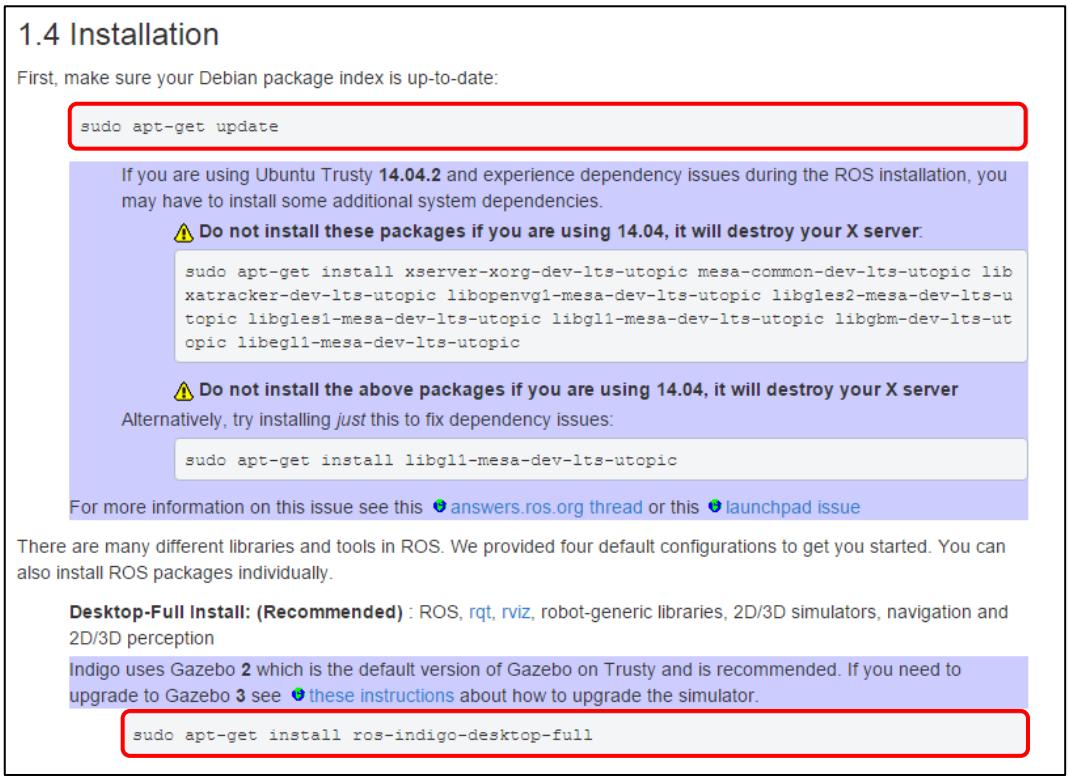


<Figure 3-9>

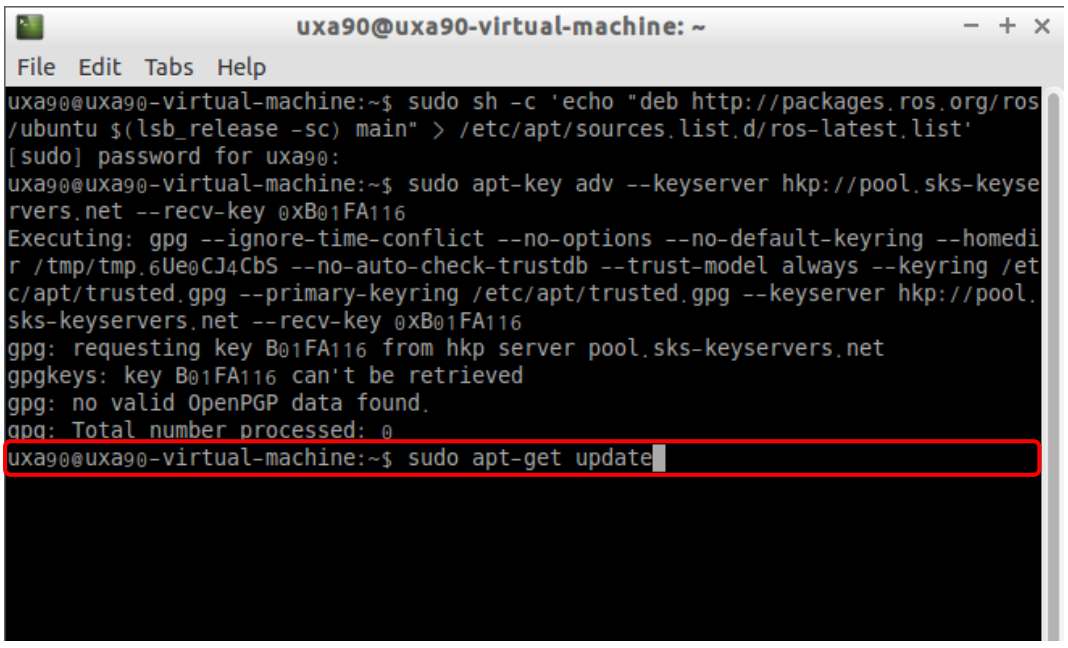
Next is '1.4 Installation'. It is a process of installing a ROS-related package. You may skip this step if you are using Lubuntu 14.04 version.

Use the 'apt-get' command to start the update.

$ sudo apt-get update



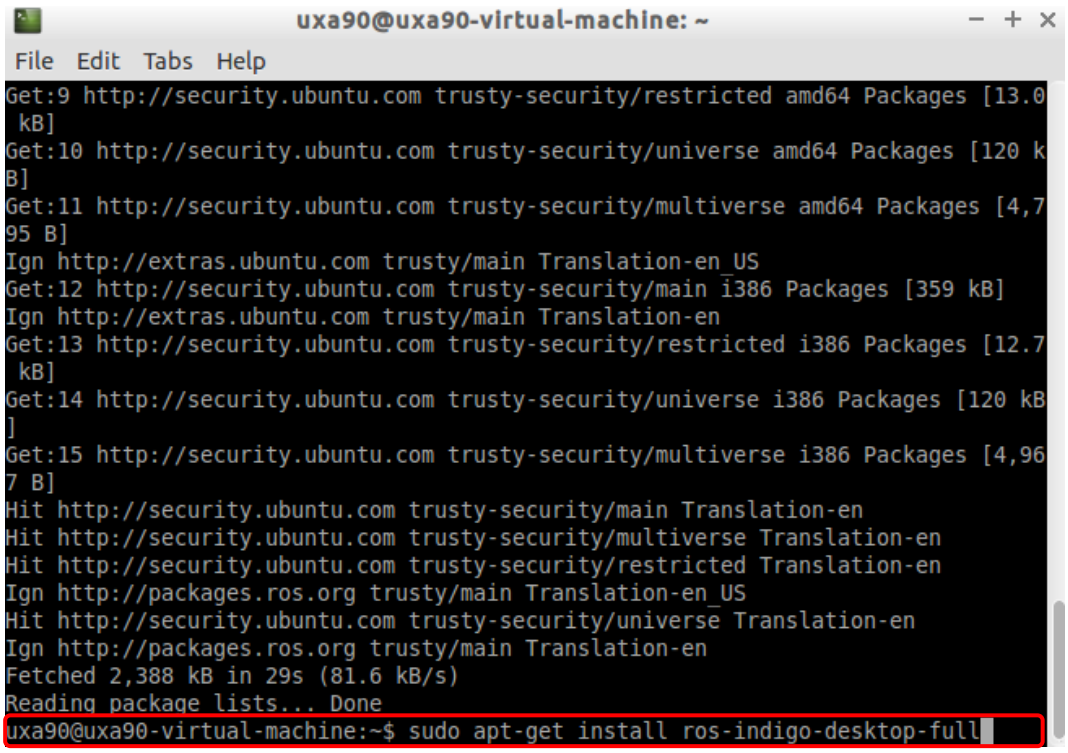
<Figure 3-10>



<Figure 3-11>

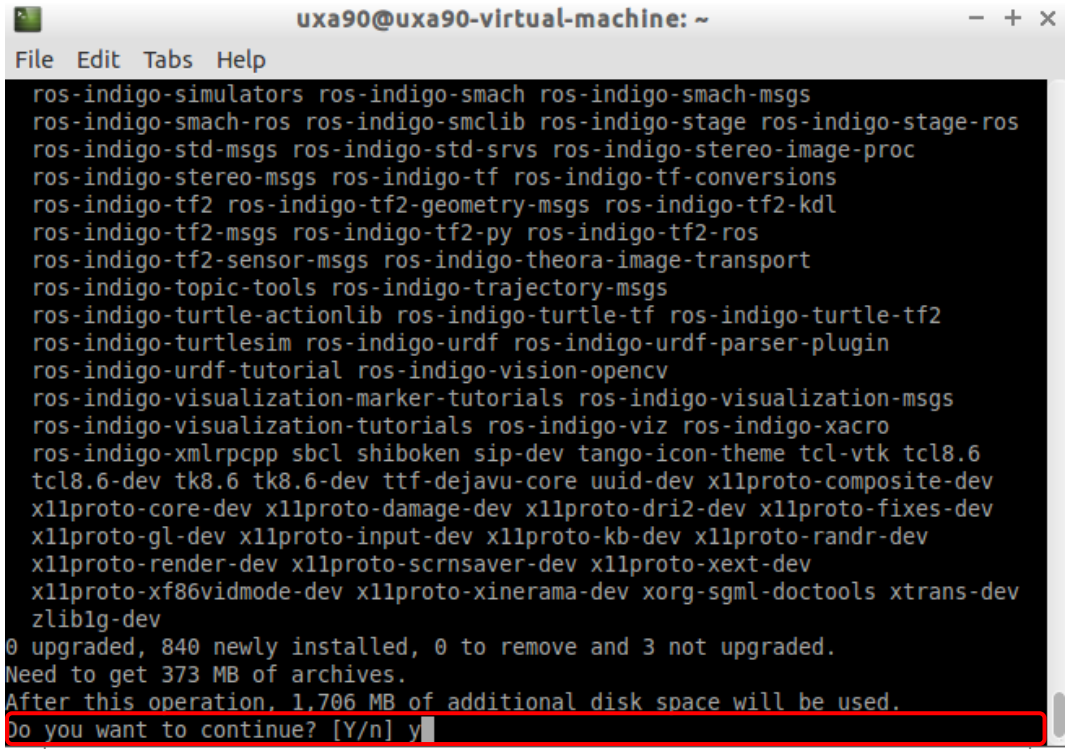
After the update, use the command below to install the entire ROS package.
Use the command on the bottom of <Figure 3-10>.

$ sudo apt-get install ros-indigo-desktop-full



<Figure 3-12>

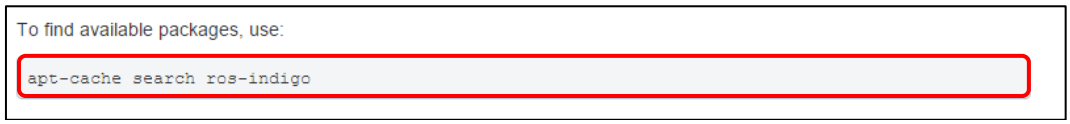
It requires permission. Enter 'y'.



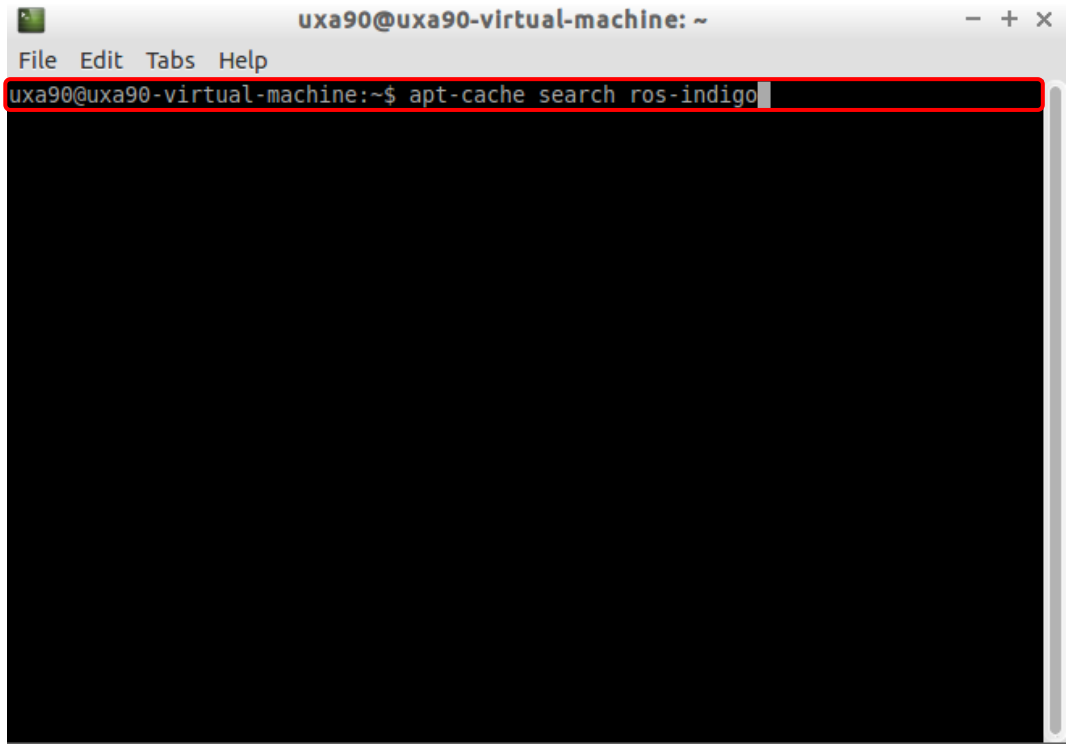
<Figure3-13>

Once finished, enter the command in <Figure 3-14>. If the ROS-Indigo package name appears, then it means the installation has been completed.

$ apt-cache search ros-indigo



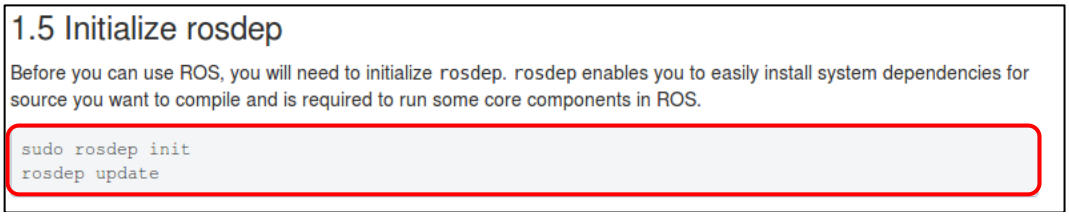
<Figure 3-14>



<Figure 3-15>

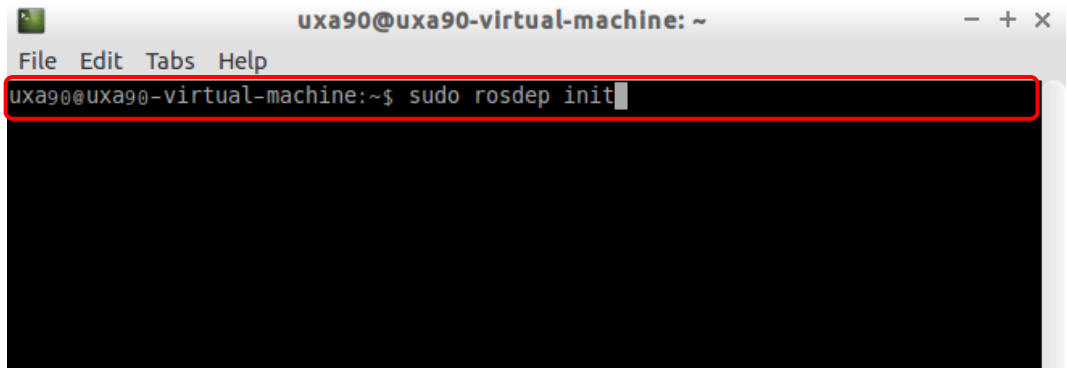
After installation, rosdep needs to be initialized before using ROS. Rosdep helps with easy installation of dependent package when using ROS components or compiling packages.

Refer to <Figure 3-16> and continue on to '1.5 Initialize rosdep'.



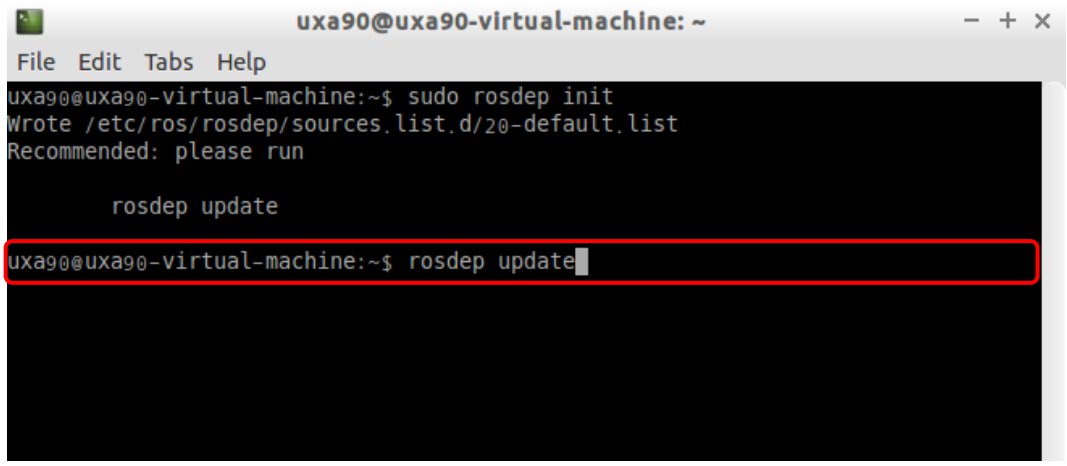
<Figure 3-16>

$ sudo rosdep init



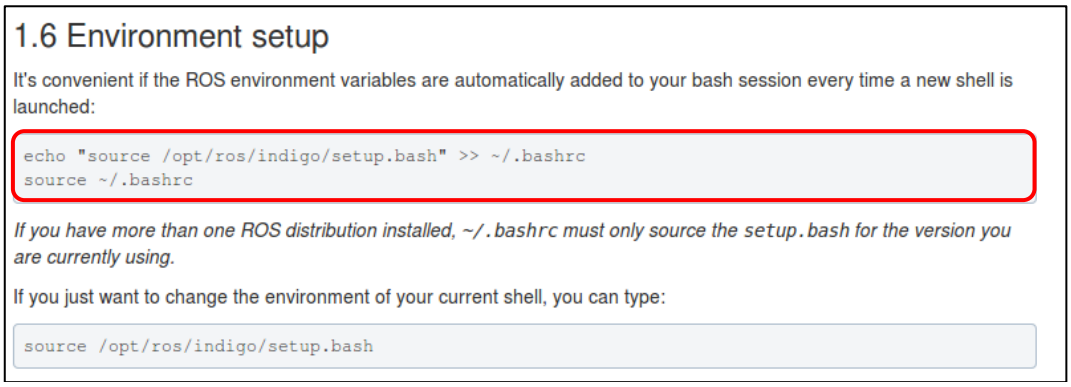
<Figure 3-17>

$ rosdep update



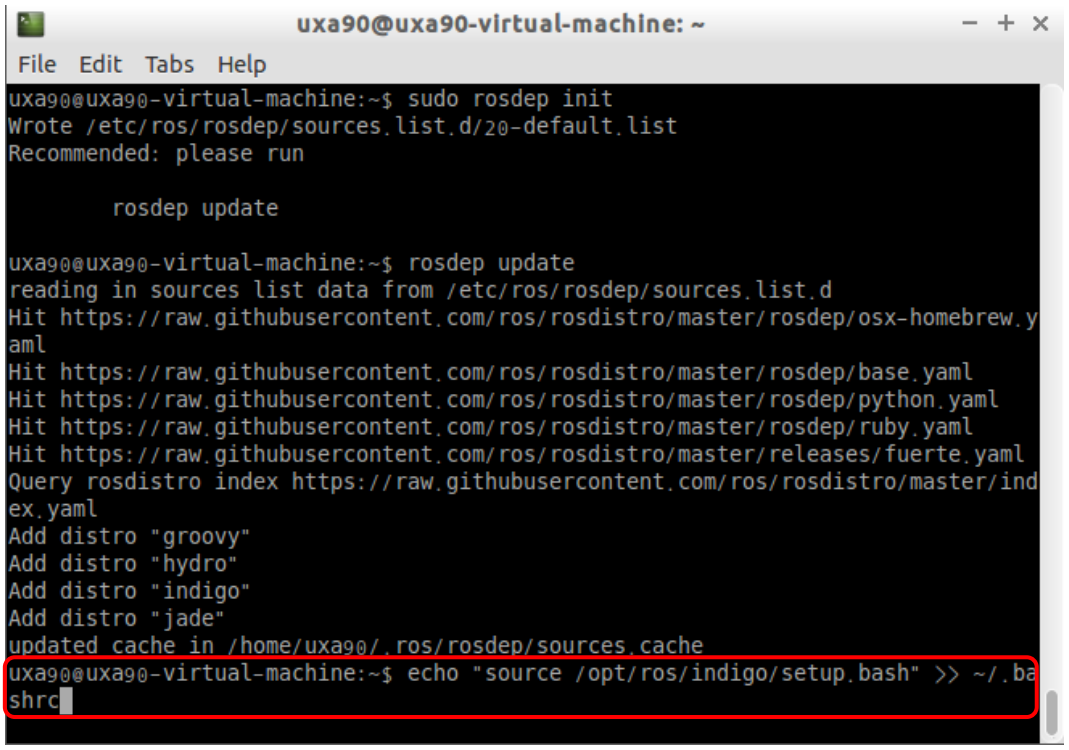
<Figure 3-18>

Next, let's set ros by changing 'bashrc' to read a specific setting file every time the terminal is opened. See <Figure 3-19>.



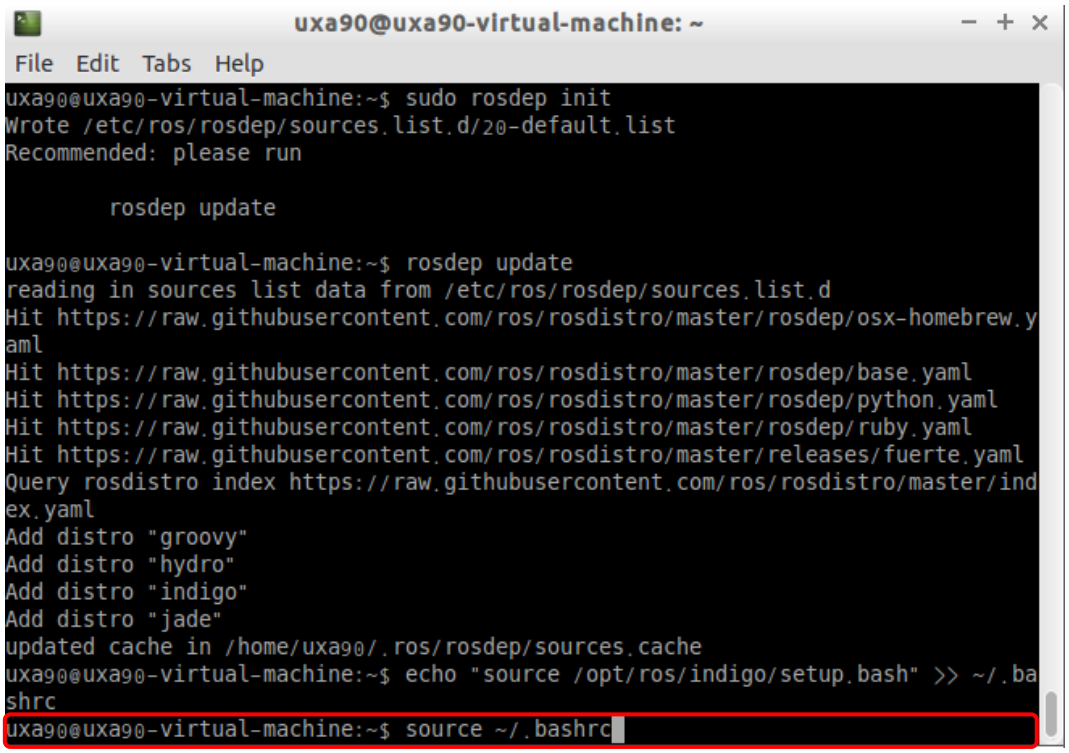
<Figure 3-19>

$ echo "source /opt/ros/indigo/setup.bash" >> ~/.bashrc



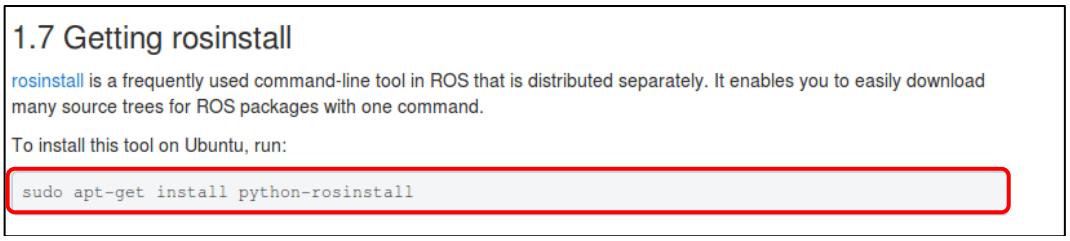
<Figure 3-20>

$ source ~/.bashrc



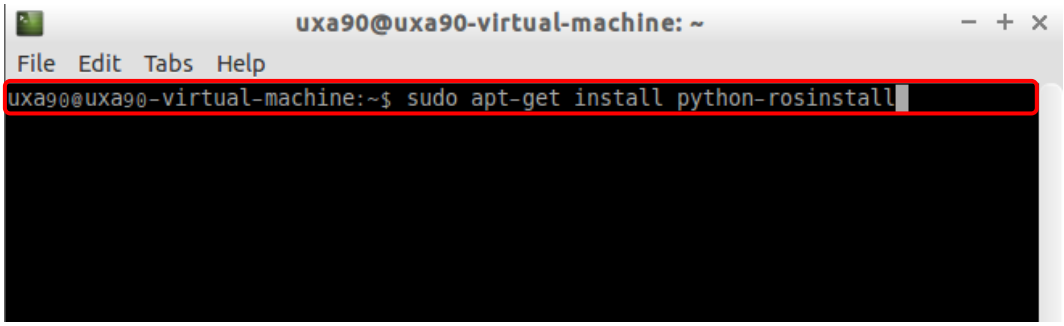
<Figure 3-21>

Lastly, let's install various ROS packages. Use the command in <Figure 3-22>.



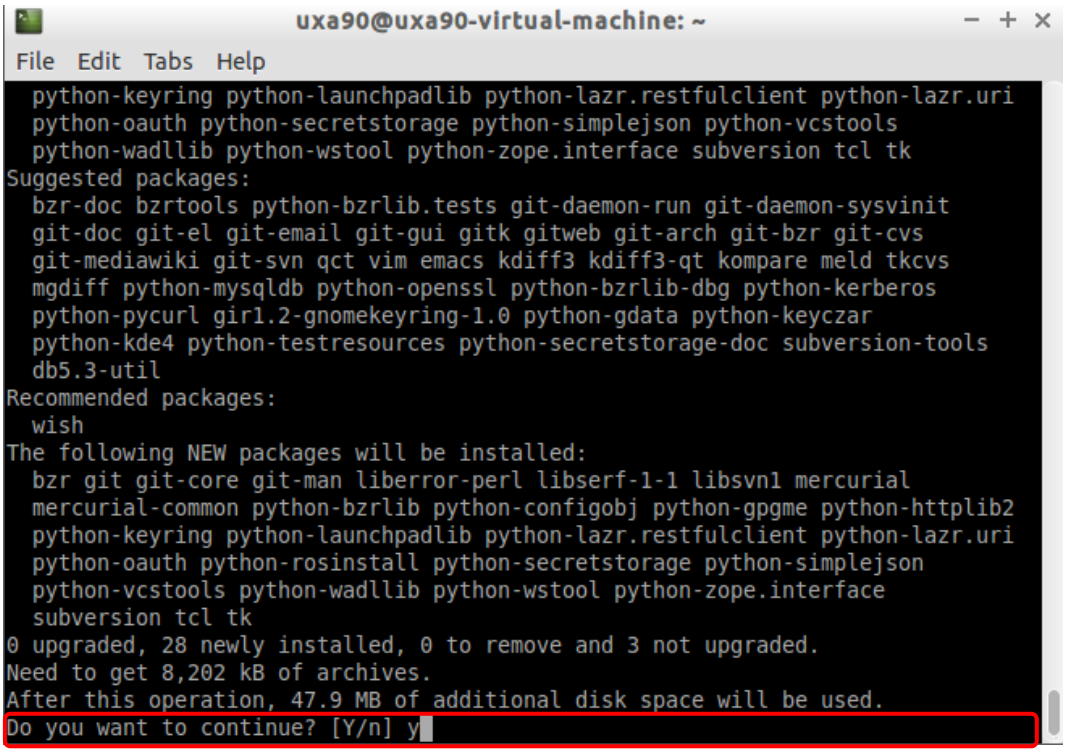
<Figure 3-22>

$ sudo apt-get install python-rosinstall



<Figure 3-23>

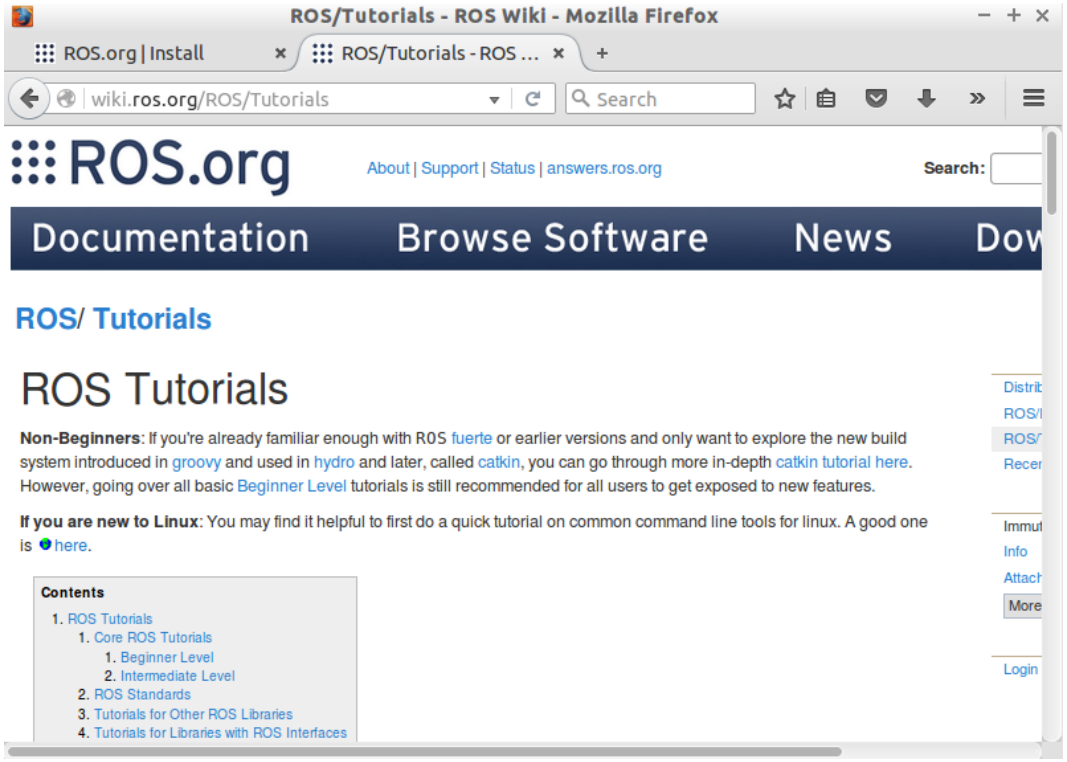
Permission required. Enter 'y'.



<Figure 3-24>

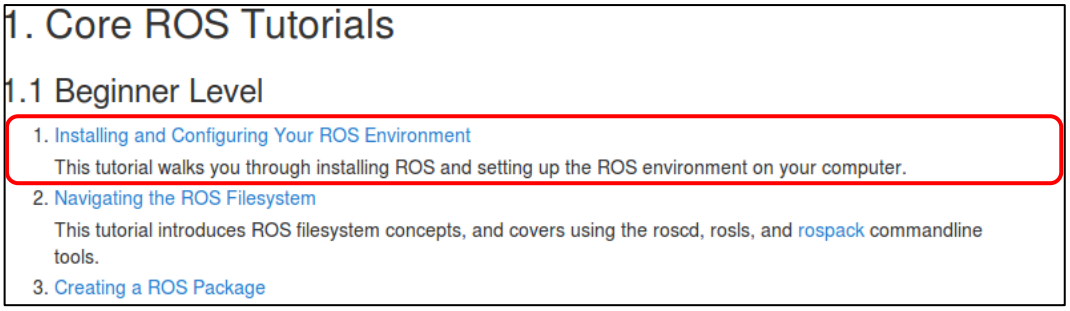
Installation has been completed. Move on to '2. Tutorials' to modify configuration.

Enter the setting page and you will see the page that looks like <Figure 3-25>. This page describes the configuration and tutorials.



<Figure 3-25>

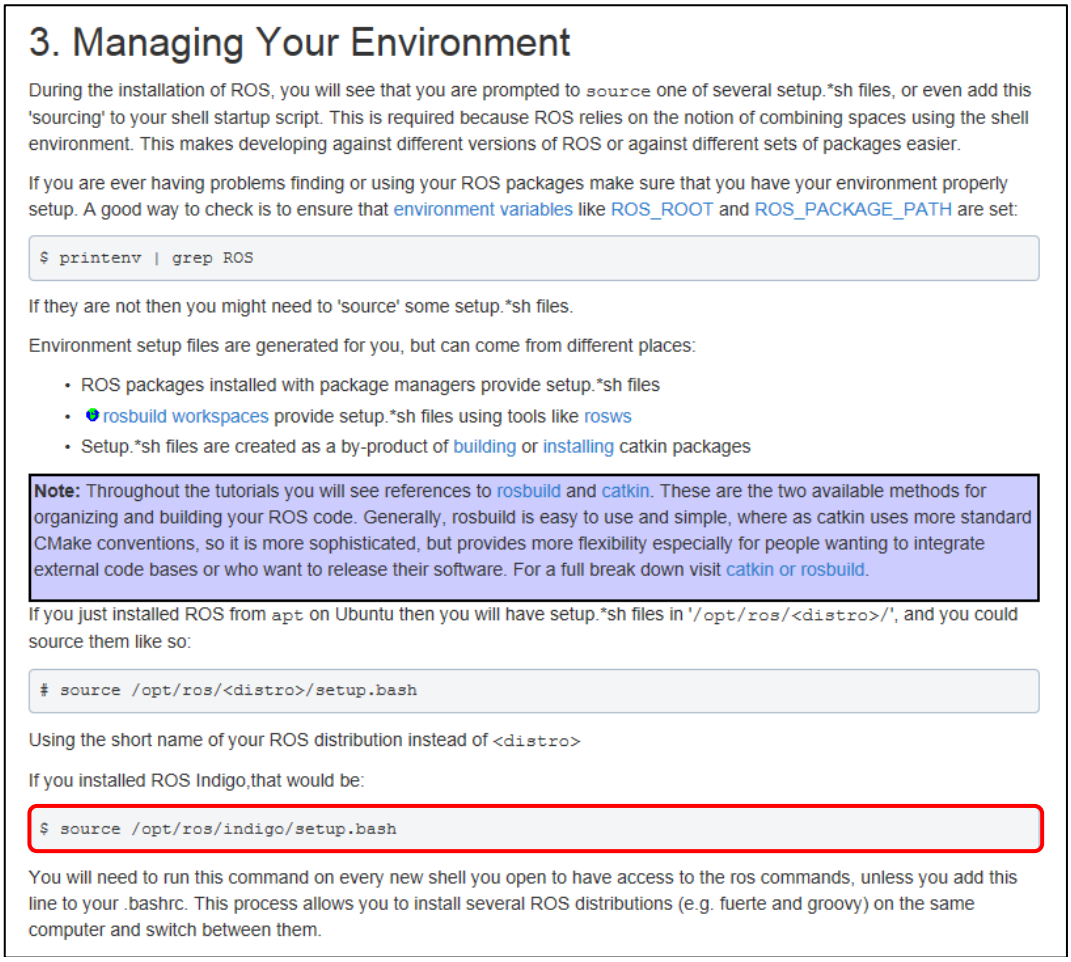
First click '1. Installing and Configuring Your ROS Environment' as shown in <Figure 3-26> and start ROS configuration.



<Figure 3-26>

Skip everything and start from '3 Managing Your Environment'. In this step, you will open the configuration file.

Refer to <Figure 3-27> and enter into the terminal screen.



3. Managing Your Environment

During the installation of ROS, you will see that you are prompted to `source` one of several setup.*sh files, or even add this 'sourcing' to your shell startup script. This is required because ROS relies on the notion of combining spaces using the shell environment. This makes developing against different versions of ROS or against different sets of packages easier.

If you are ever having problems finding or using your ROS packages make sure that you have your environment properly setup. A good way to check is to ensure that environment variables like ROS_ROOT and ROS_PACKAGE_PATH are set:

```
$ printenv | grep ROS
```

If they are not then you might need to 'source' some setup.*sh files.

Environment setup files are generated for you, but can come from different places:

- ROS packages installed with package managers provide setup.*sh files
- rosbuild workspaces provide setup.*sh files using tools like rosws
- Setup.*sh files are created as a by-product of building or installing catkin packages

Note: Throughout the tutorials you will see references to rosbuild and catkin. These are the two available methods for organizing and building your ROS code. Generally, rosbuild is easy to use and simple, where as catkin uses more standard CMake conventions, so it is more sophisticated, but provides more flexibility especially for people wanting to integrate external code bases or who want to release their software. For a full break down visit catkin or rosbuild.

If you just installed ROS from `apt` on Ubuntu then you will have setup.*sh files in '`/opt/ros/<distro>/`', and you could source them like so:

```
# source /opt/ros/<distro>/setup.bash
```

Using the short name of your ROS distribution instead of `<distro>`

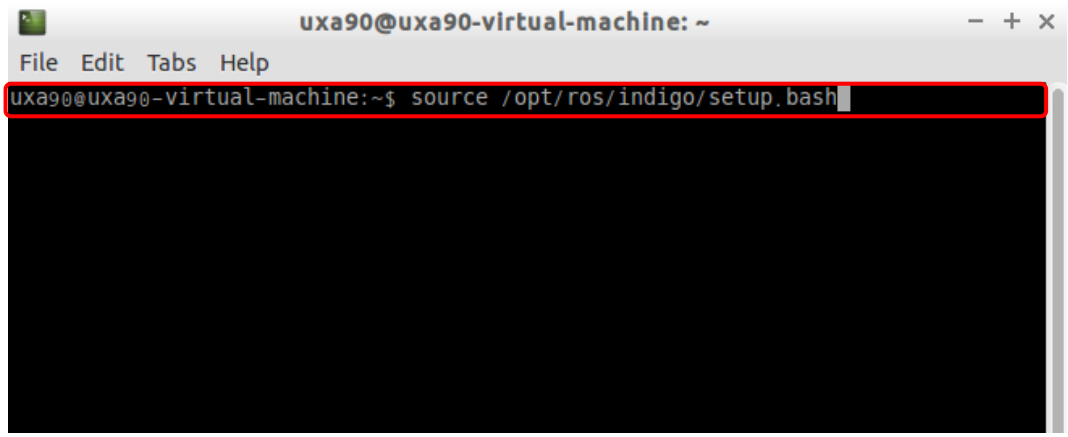
If you installed ROS Indigo,that would be:

```
$ source /opt/ros/indigo/setup.bash
```

You will need to run this command on every new shell you open to have access to the ros commands, unless you add this line to your .bashrc. This process allows you to install several ROS distributions (e.g. fuerte and groovy) on the same computer and switch between them.

<Figure 3-27>

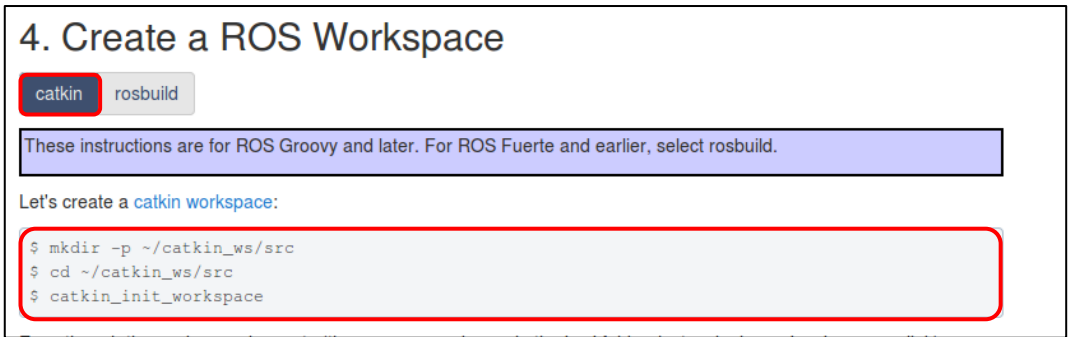
$ source /opt/ros/indigo/setup.bash



<Figure 3-28>

Next is '4 Create a ROS Workspace'. This step is to create a workspace. The two options - 'catkin' and 'rosbuild' – are a ROS-specialized build system. This manual will use the 'catkin' build system.

First, create a workspace as shown in <Figure 3-27>.

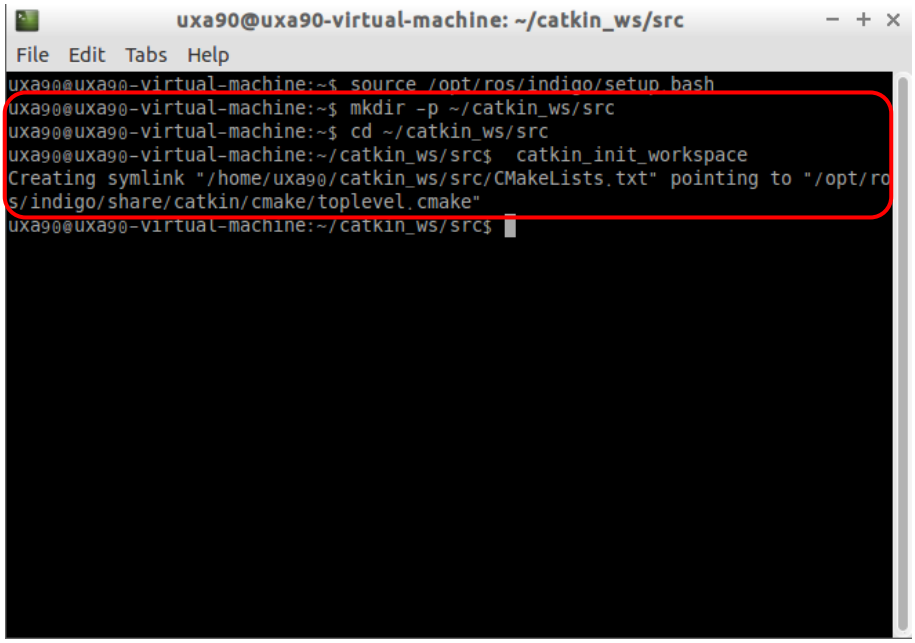


<Figure 3-27>

$ mkdir -p ~/catkin_ws/src

$ cd ~/catkin_ws/src

$ catkin_init_workspace



<Figure 3-28>

Next, build the workspace you have created. See <Figure 3-29>.

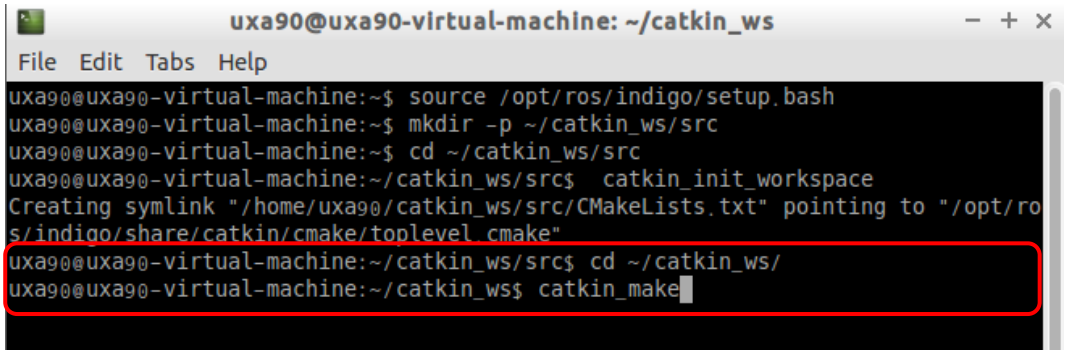
Even though the workspace is empty (there are no packages in the 'src' folder, just a single `CMakeLists.txt` link) you can still "build" the workspace:

```
$ cd ~/catkin_ws/
$ catkin_make
```

<Figure 3-29>

$ cd ~/catkin_ws/

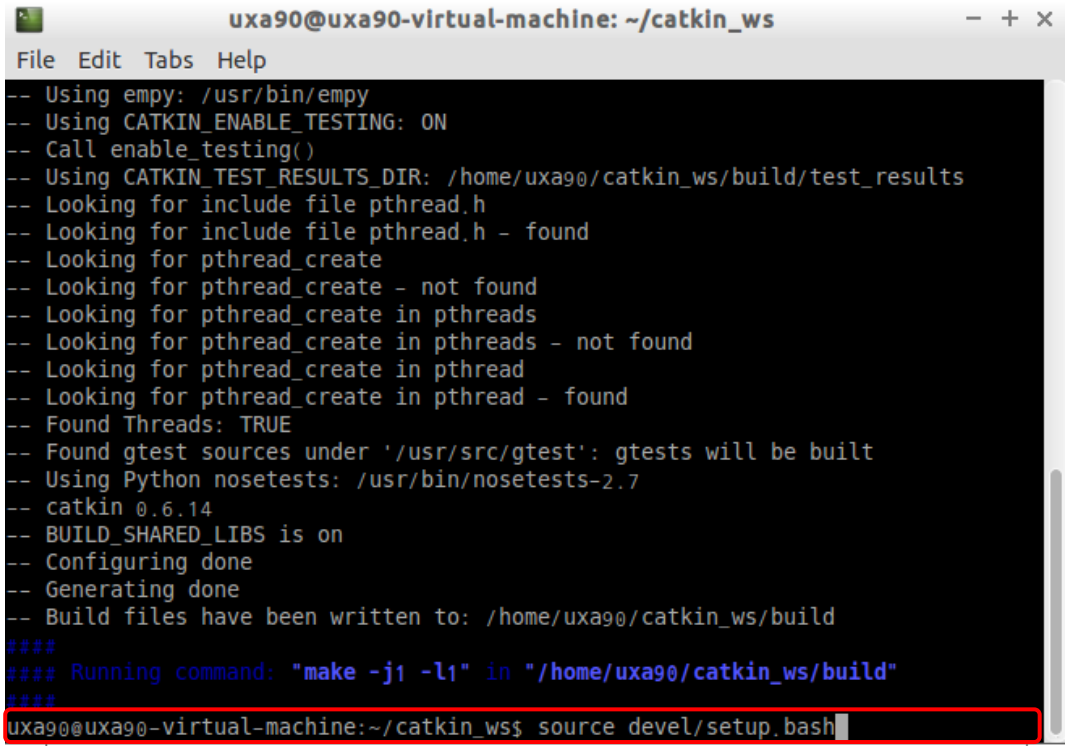
$ catkin_make



<Figure 3-30>

Lastly, source the bash file to finish configuration.

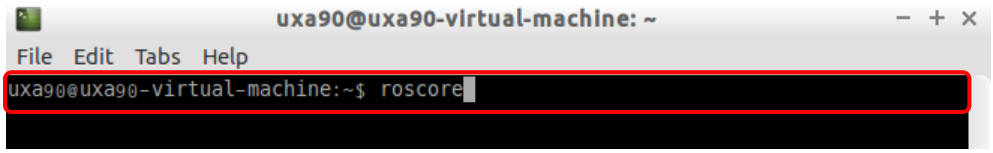
$ source devel/setup.bash



<Figure 3-31>

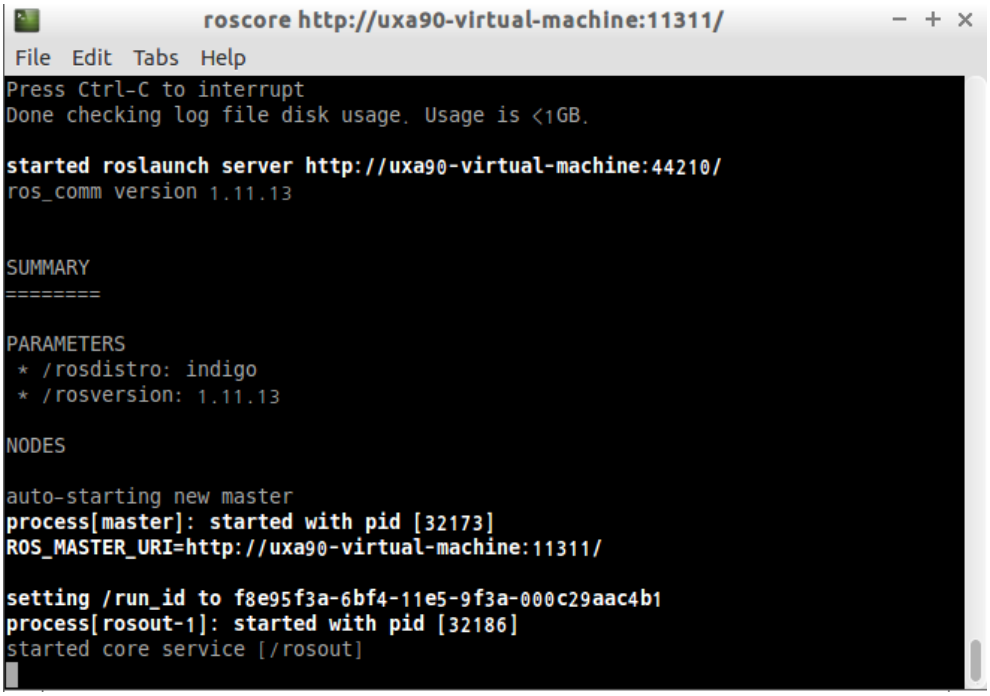
After installation, run the installation test. Run a simple example, 'turtlesim'. This example is a package that moves a turtle-shaped robot shown on the screen with the keyboard. First, run roscore, which controls the ros system.

$ roscore



<Figure 3-32>

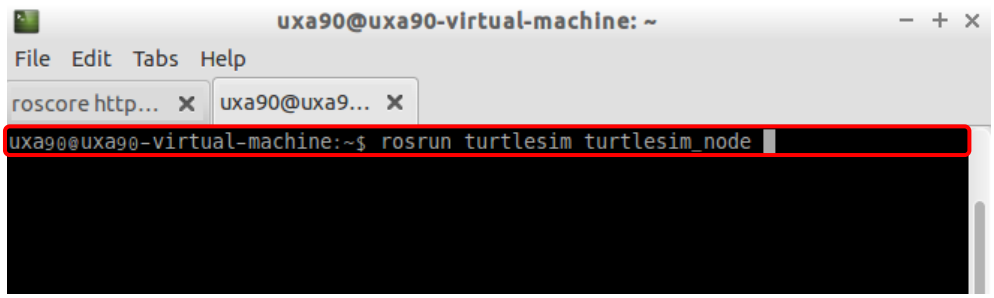
If you see the message as in <Figure 3-33>, it means the installation was successful.



<Figure 3-33>

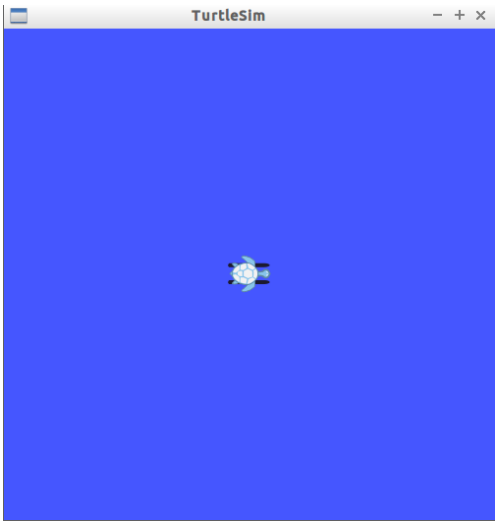
Next, open a new window and run 'turtlesim_node'. See <Figure 3-34>.

$ rosrun turtlesim turtlesim_node



<Figure 3-34>

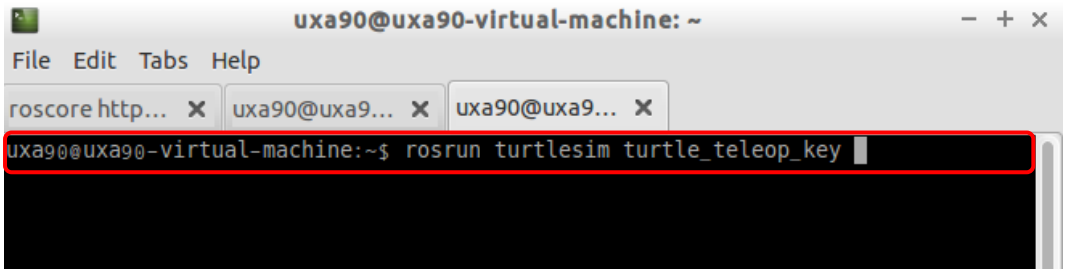
Enter the command, and you will see the simulation screen as shown in <Figure 3-35>.



<Figure 3-35>

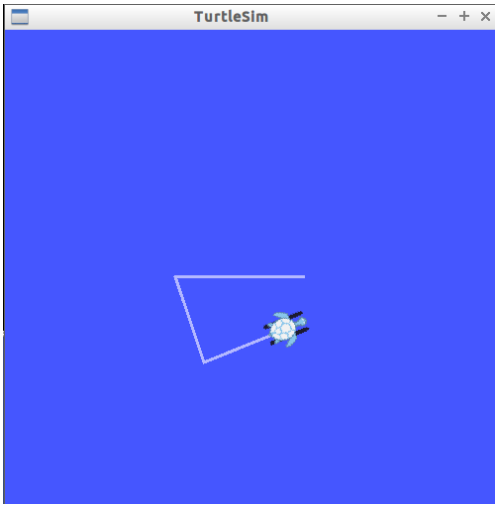
Next, open a new terminal screen and run 'turtle_teleop_key'.

$ rosrun turtlesim turtle_teleop_key



<Figure 3-36>

Once the package is running, you will be able to control the turtle with the keyboard. If it works well as shown in <Figure 3-37>, it means the installation was successful.



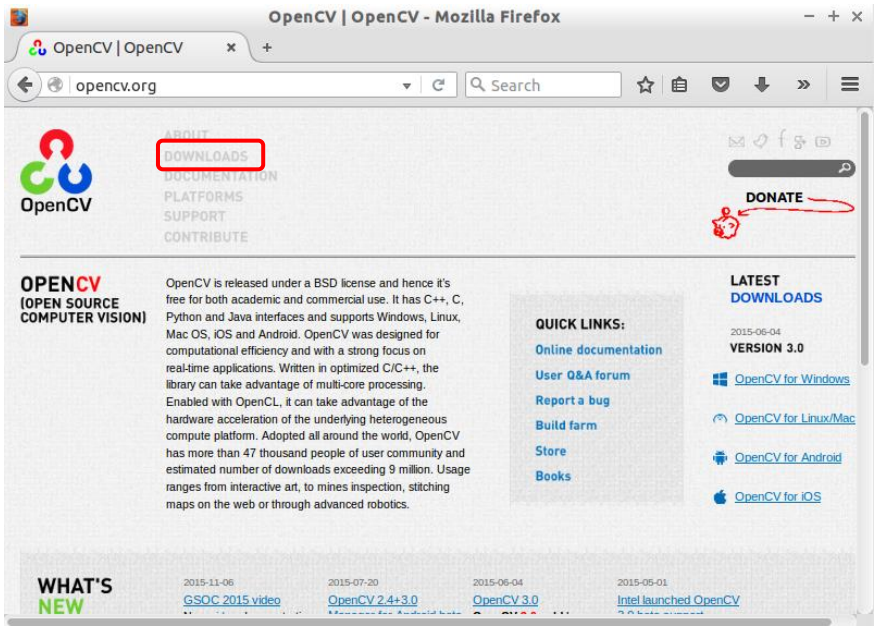
<Figure 3-37>

3. Open CV installation

This chapter will explain the OpenCV installation process.

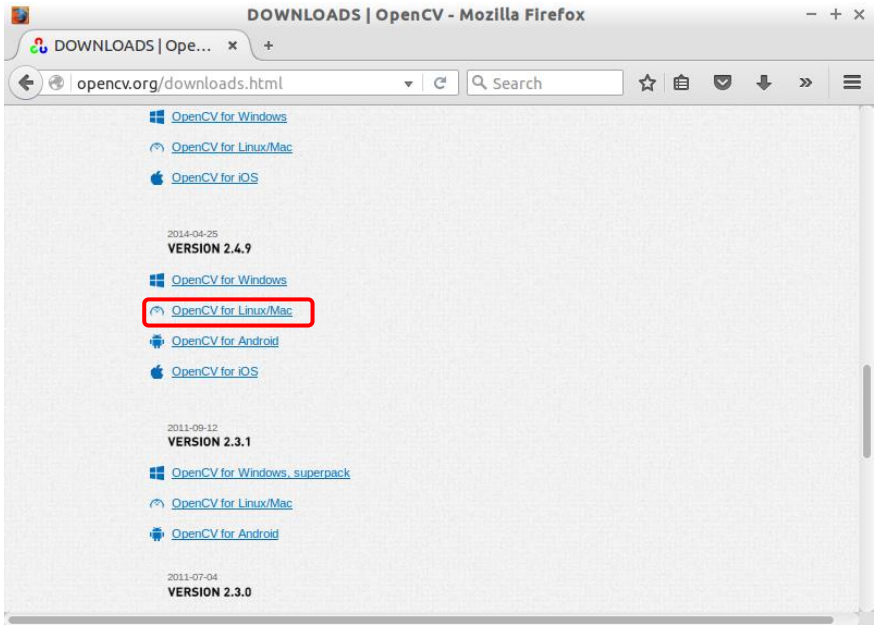
OpenCV is a shorter version of Open Computer Vision, and it is an open source computer vision C library. This manual will describe the installation process of OpenCV 2.4.9 ver.

First, run the internet browser and to go the OpenCV website, http://opencv.org. Go to 'Downloads' on the home page as shown in <Figure 4-1>.



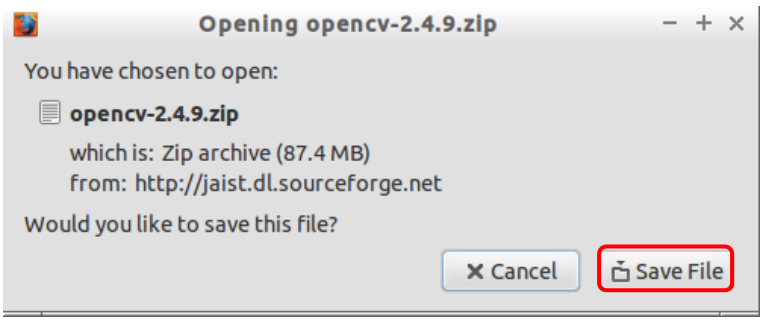
<Figure 4-1>

In 'Downloads', scroll down and click 'OpenCV for Linux/Mac' under VERSION 2.4.9.



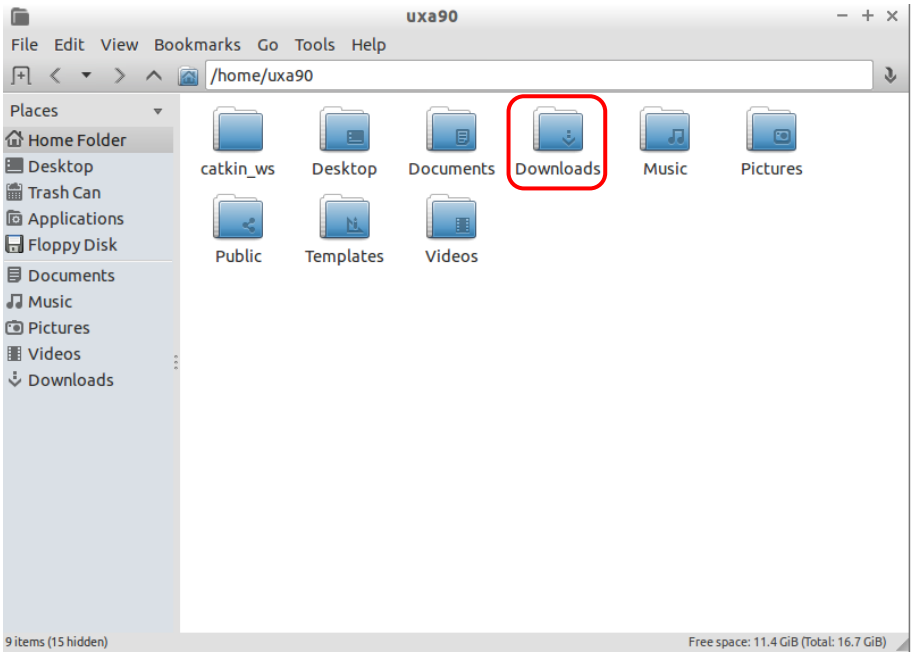
<Figure 4-2>

Wait about 5 seconds, and you will see a download screen. Click 'Save file' and start the download.



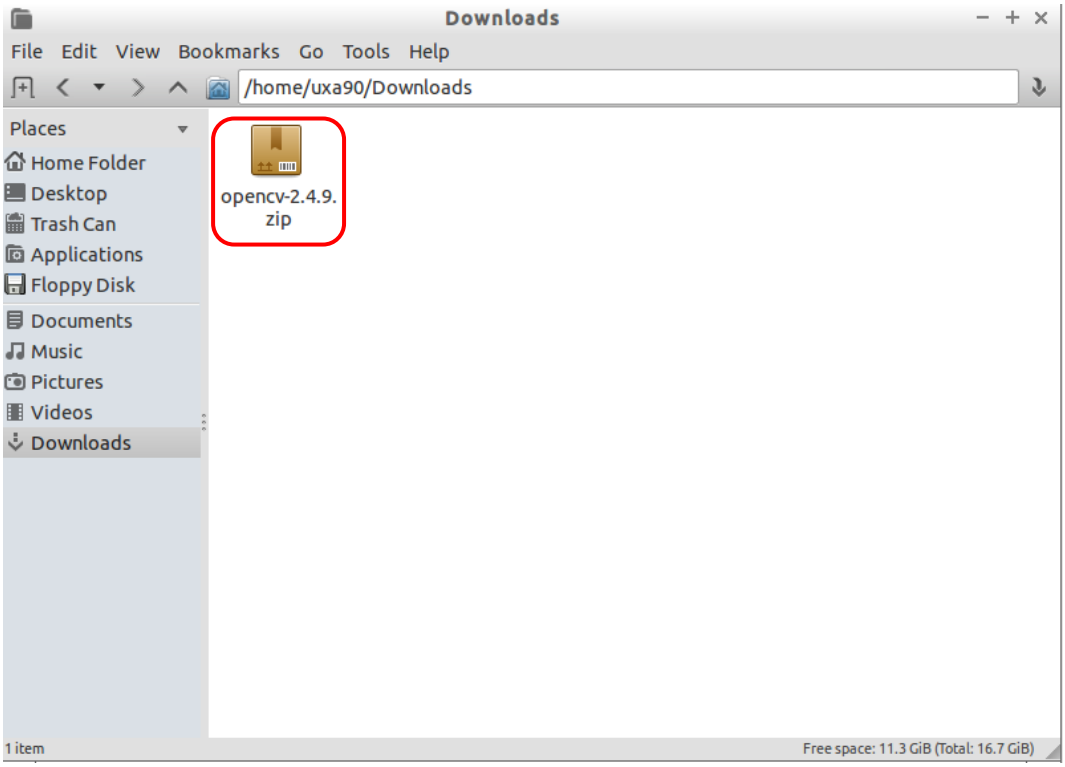
<Figure 4-3>

Once the file is downloaded, go to the download folder and unzip the file. The download folder is located in '/home/uxa90'.



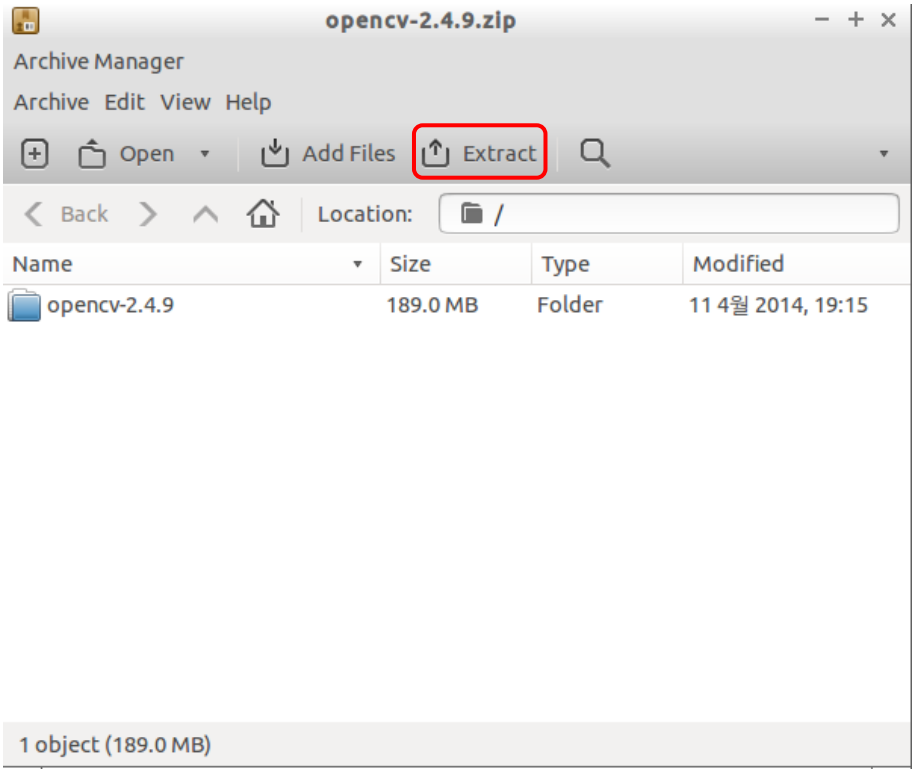
<Figure 4-4>

Double click the file in <Figure 4-5> to run 'Archive Manager'.



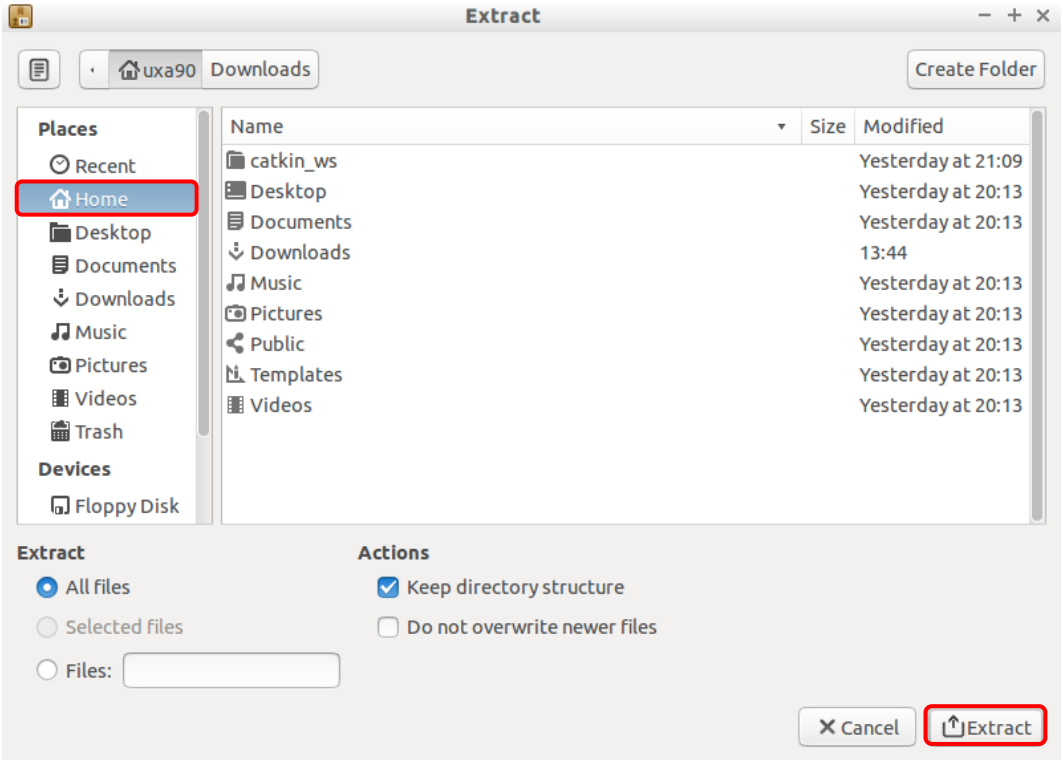
<Figure 4-5>

When 'Archive Manager' starts, unzip the file. Click 'Extract' as shown in <Figure 4-6>.



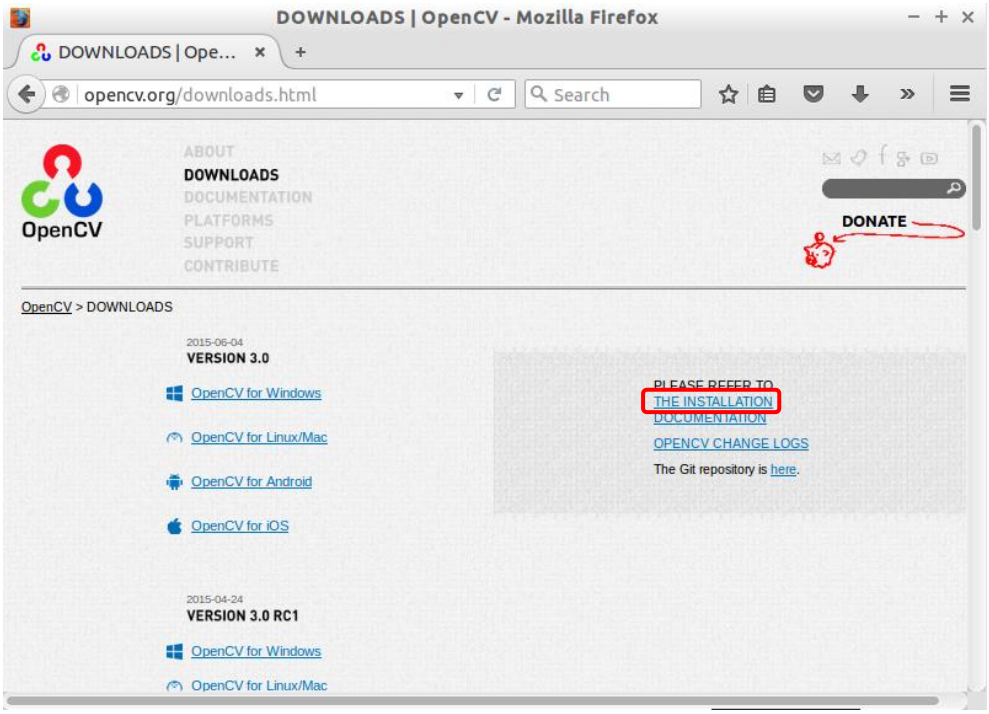
<Figure 4-6>

When you click 'Extract', you have to assign the location, and in this case, choose 'home'. See <Figure 4-7>.



<Figure 4-7>

After the file was extracted, start the installation. If you go back to the web browser, you will see 'THE INSTALLATION' as shown in <Figure 4-8>. Use the guideline here for easier installation.

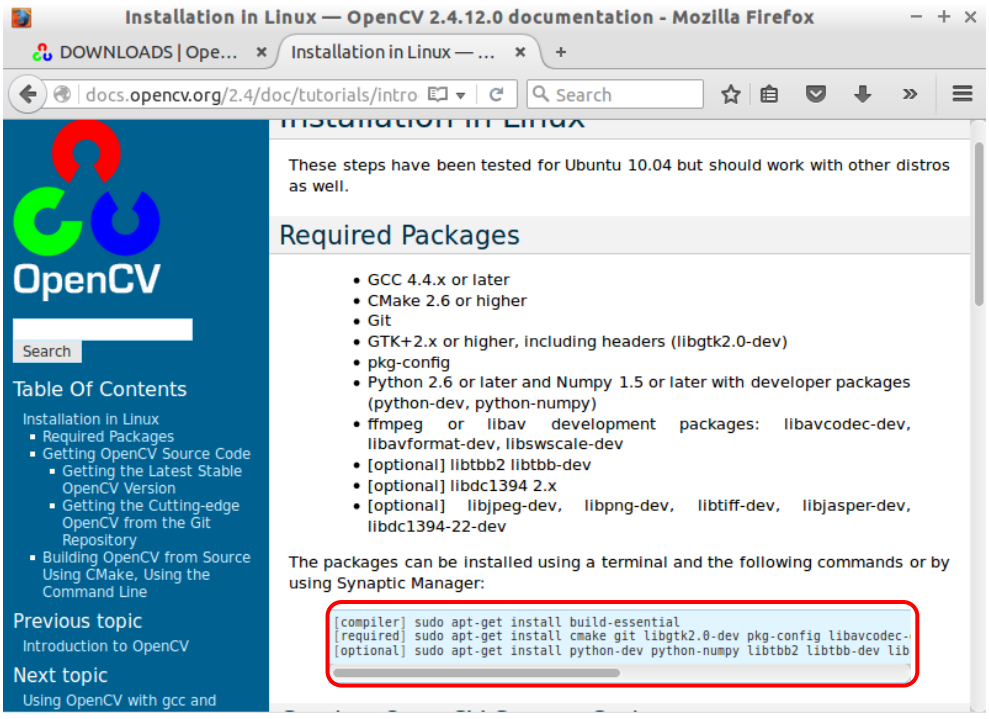


<Figure 4-8>

When you see the screen that looks like <Figure 4-9>, select 'Installation in Linux' to begin installation.

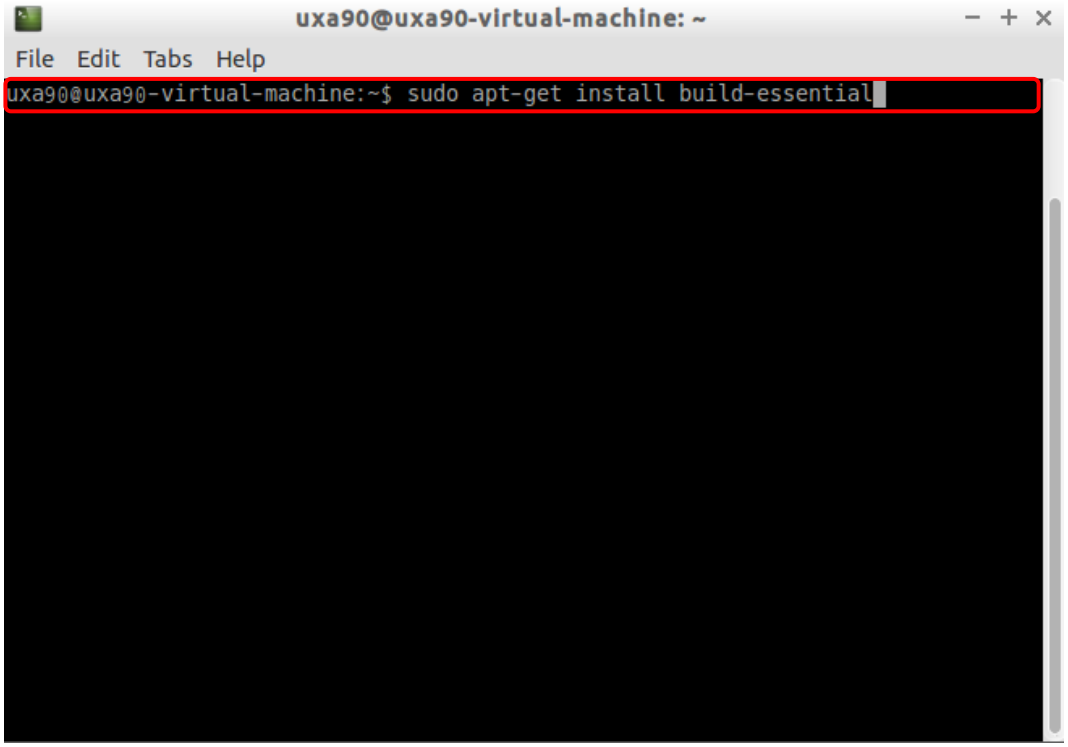
<Figure 4-9>

Follow the installation direction in order. First, start with 'Required Packages'. It is already installed, but just to confirm, reinstall.



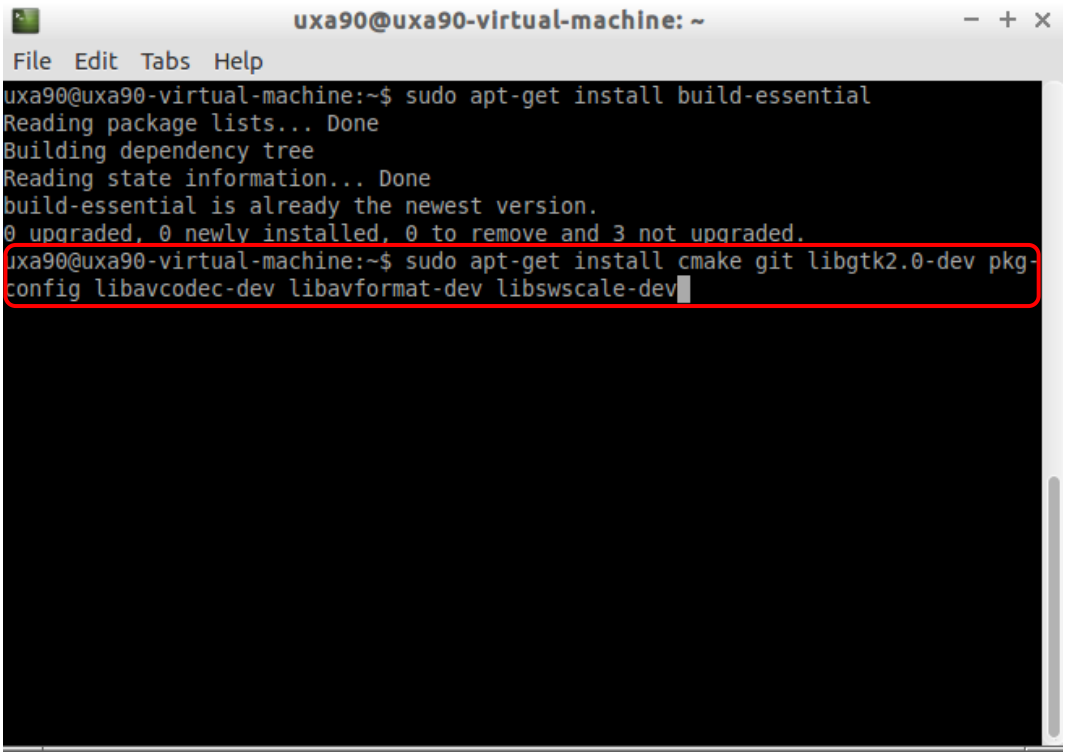
<Figure 4-10>

$ sudo apt-get install build-essential



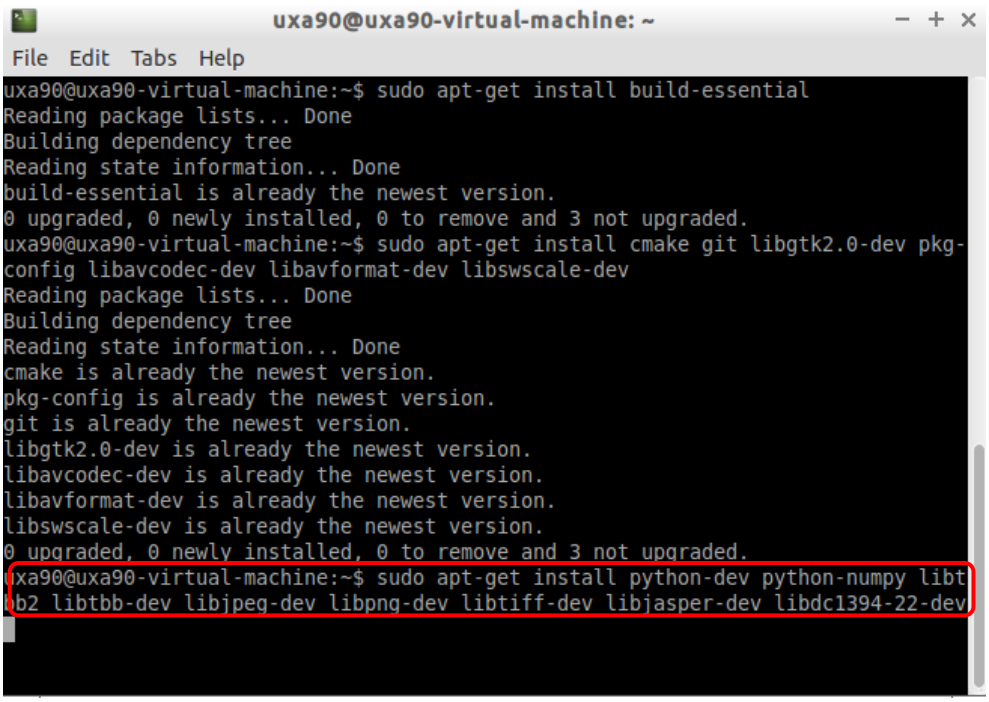
<Figure 4-11>

$ sudo apt-get install cmake git libgtk2.0-dev pkg-config libavcodec-dev libavformat-dev libswscale-dev



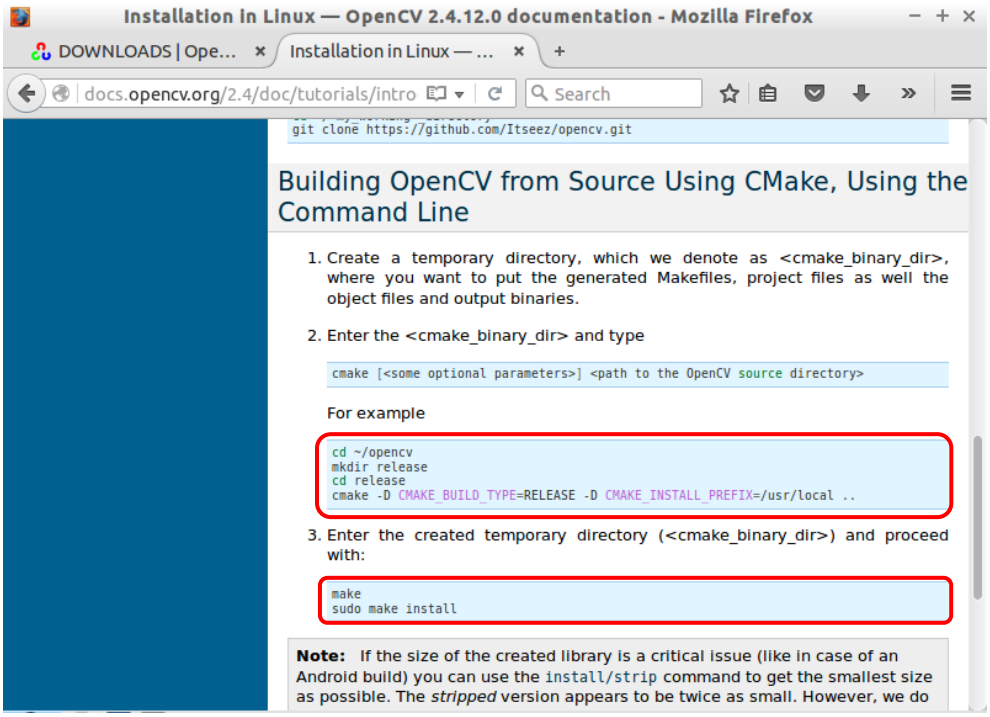
<Figure 4-12>

$ sudo apt-get install python-dev python-numpy libtbb2 libtbb-dev libjpeg-dev libpng-dev libtiff-dev libjasper-dev libdc1394-22-dev



<Figure 4-13>

After you are done with 'Required Packages', go on to the very bottom to 'Building OpenCV from Source Using CMake, Using the Command Line'. Here, you will start the installation.



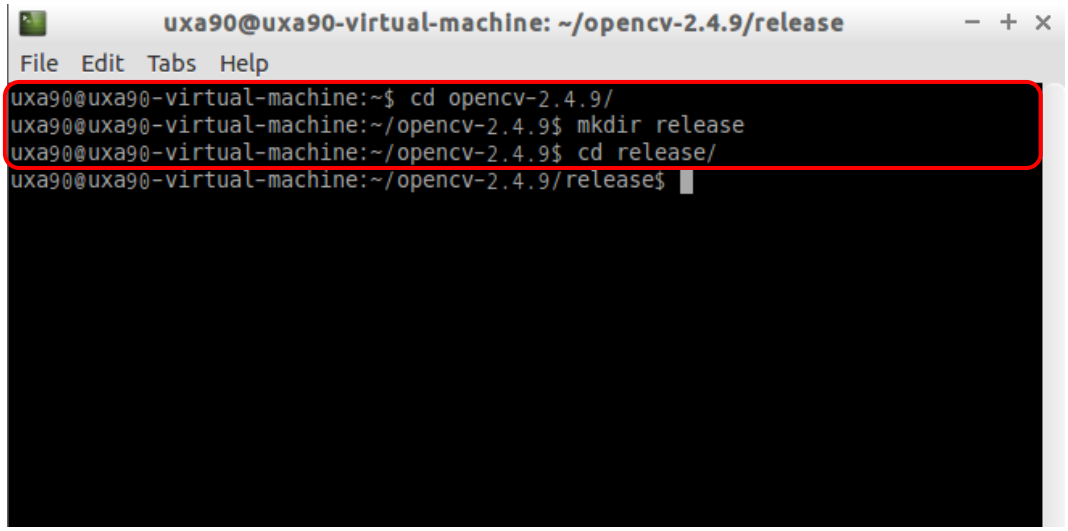
<Figure 4- 14>

First, change the location and create a folder 'release'. Enter the commands below.

$ cd ~/opencv

$ mkdir release

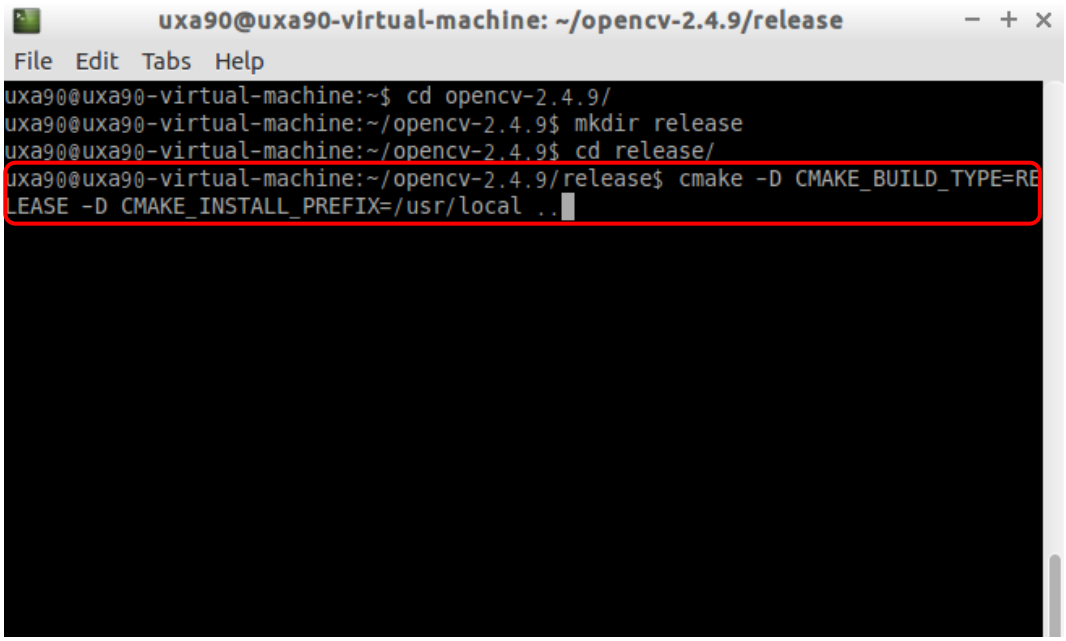
$ cd release



<Figure 4-15>

Set the configuration to assign the location and type for CMake.

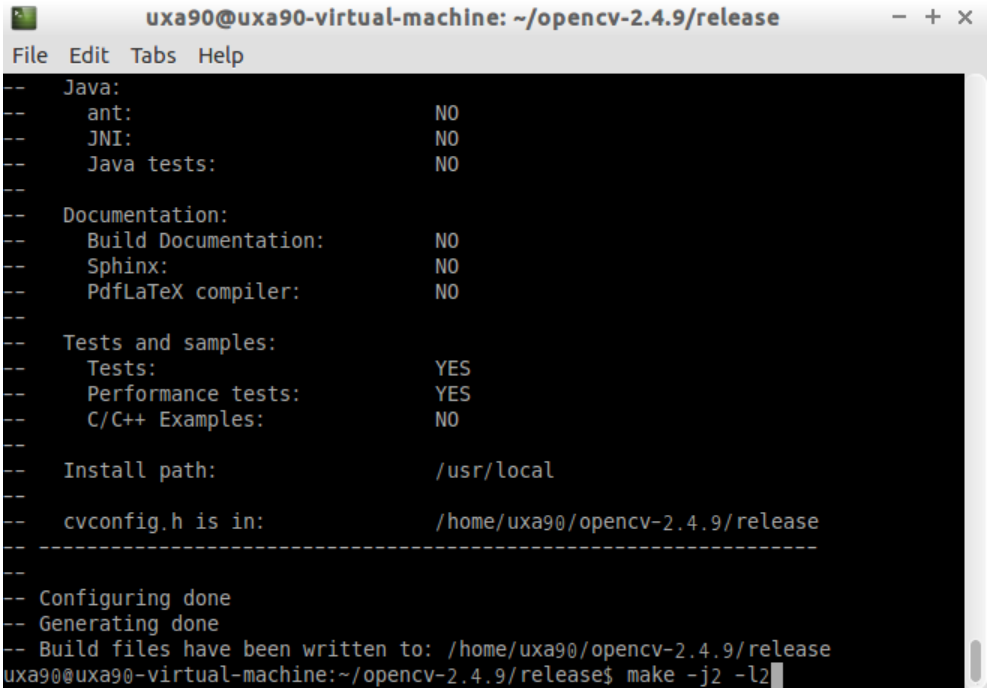
$ cmake -D CMAKE_BUILD_TYPE=RELEASE -D CMAKE_INSTALL_PREFIX=/usr/local ..



<Figure 4-16>

Once the installation is completed, make.

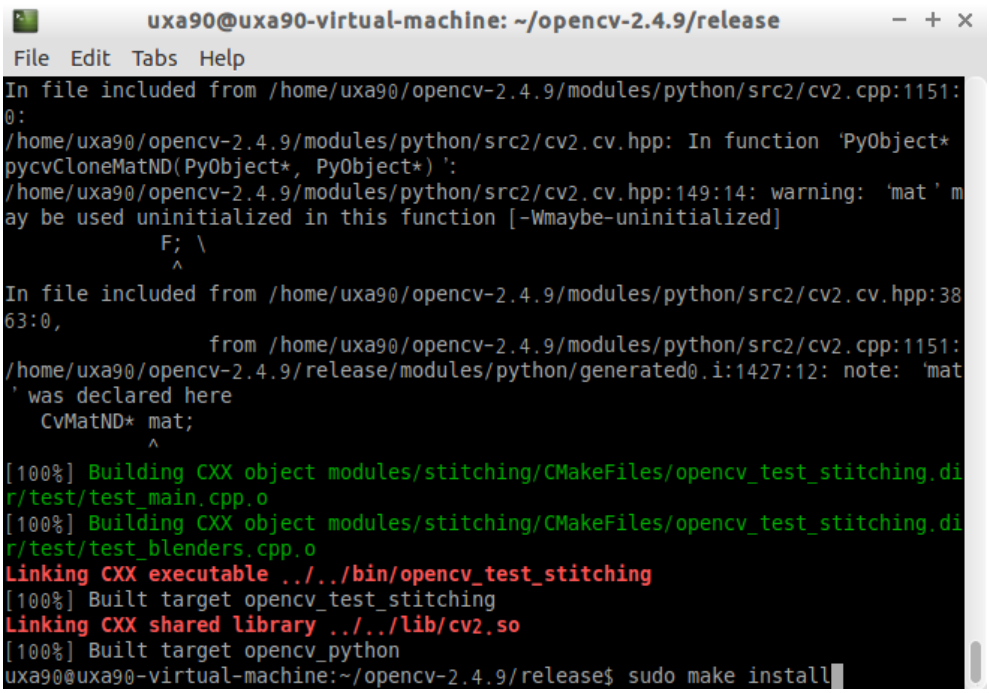
$ make –j2 –l2



<Figure 4-17>

After make, make install to complete installation.

$ sudo make install



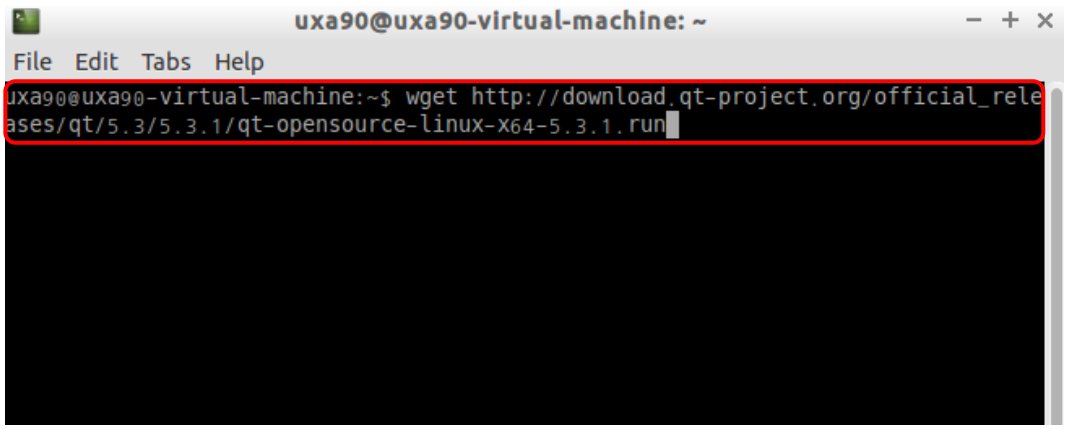
<Figure 4-18>

4. QT installation

This chapter will explain the Qt installation process, as well as how to connect Qt and ROS.

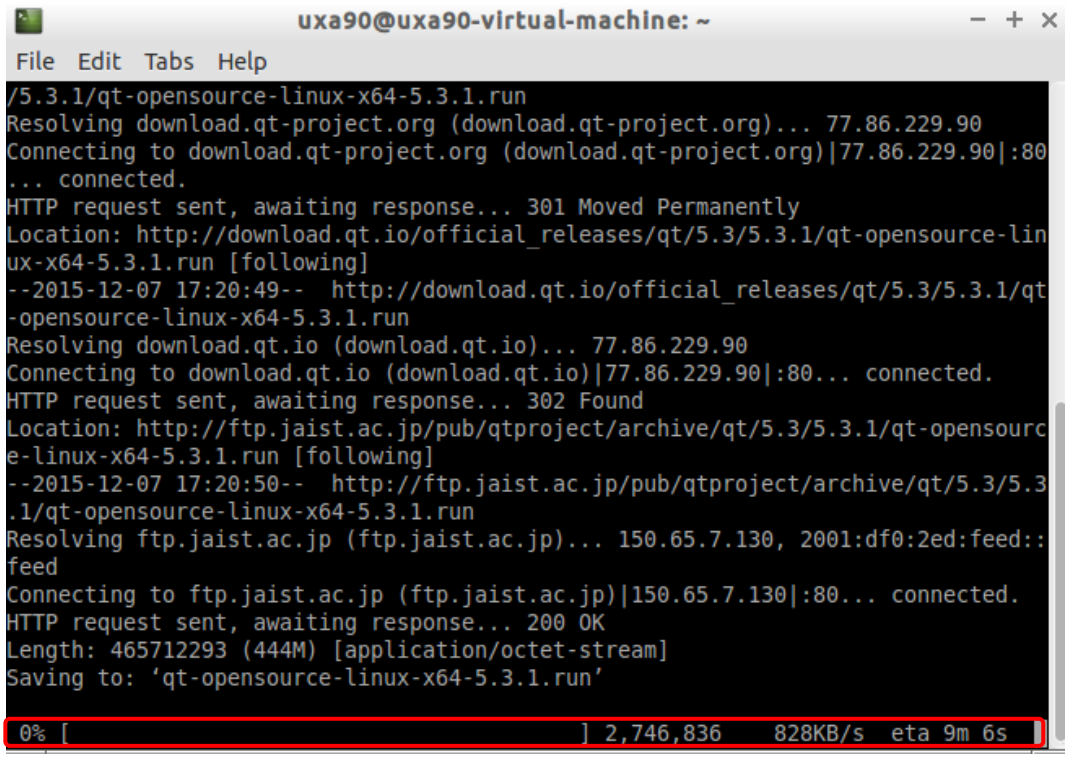
First, download the Qt file from the Qt Downloads page. Download the Qt-5.3.1 version. Run the terminal as shown in <Figure 5-1> and enter the commands.

$ wget http://download.qt-project.org/official_releases/qt/5.3/5.3.1/qt-opensource-linux-x64-5.3.1.run



<Figure 5-1>

Then the file starts to download as shown in <Figure 5-2>.

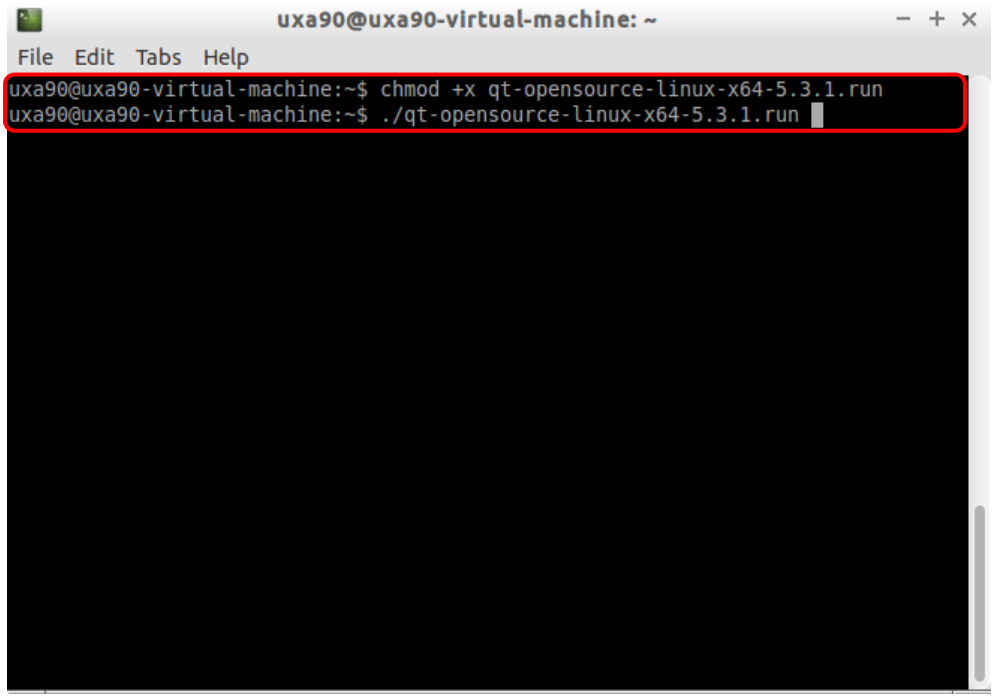


<Figure 5-2>

Once the file is downloaded, add the authority and enter the command that commands to run the file. See <Figure 5-3>.

$ chmod +x qt-opensource-linux-x64-5.3.1.run

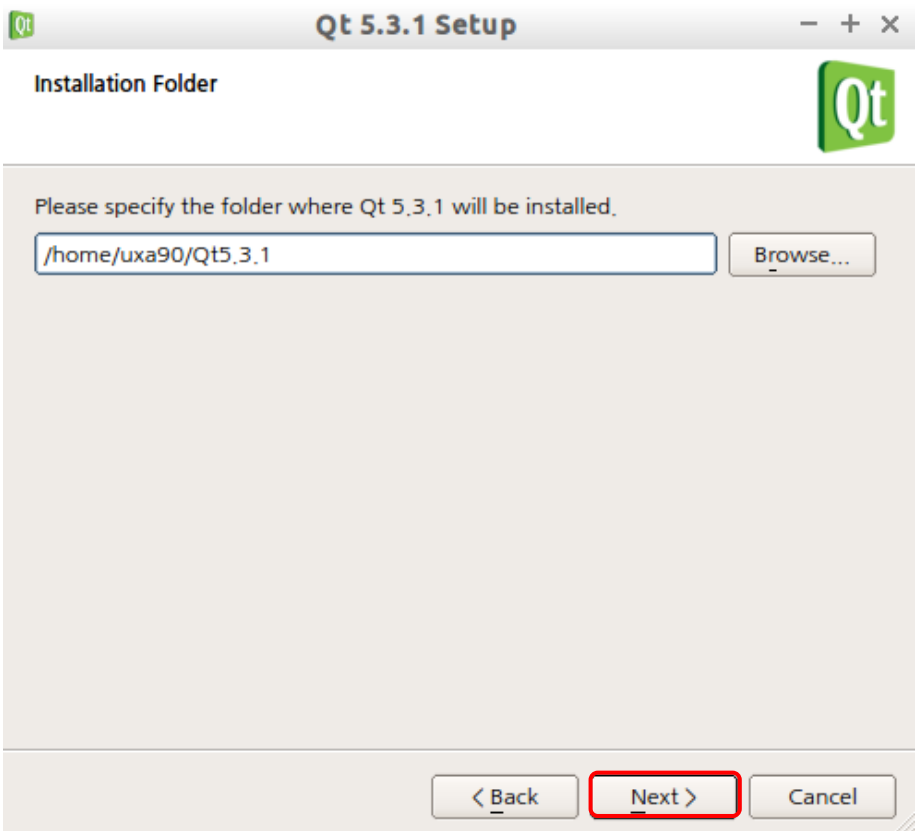
$ ./qt-opensource-linux-x64-5.3.1.run



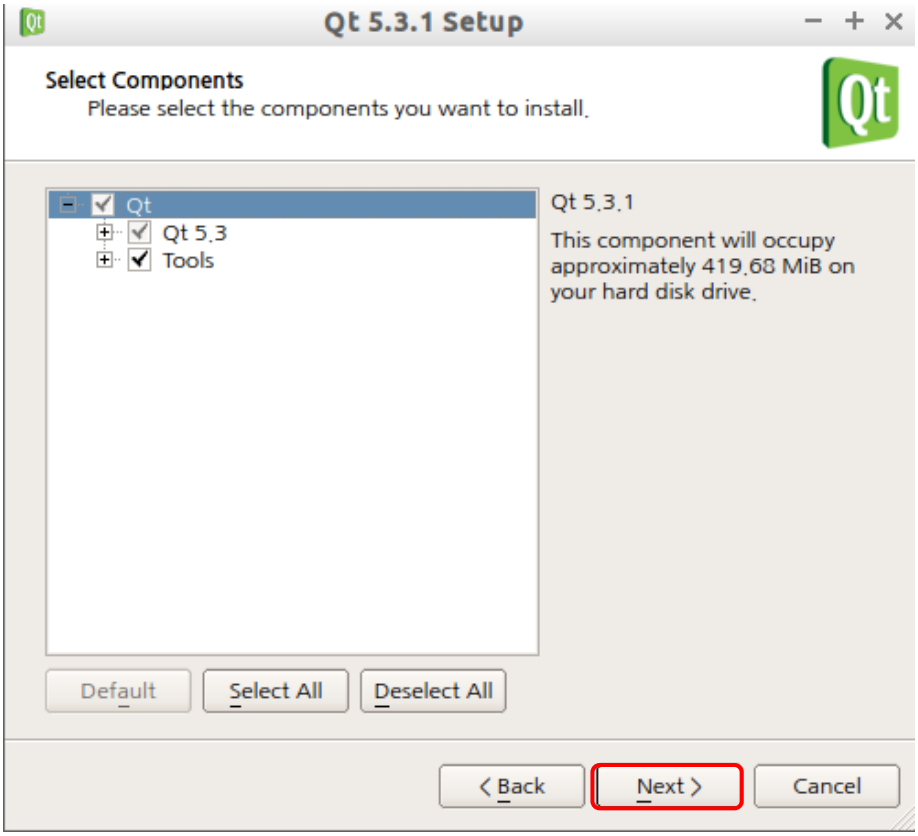
<Figure 5-3>

The installation program begins as in <Figure 5-4>. Follow the direction to complete installation.

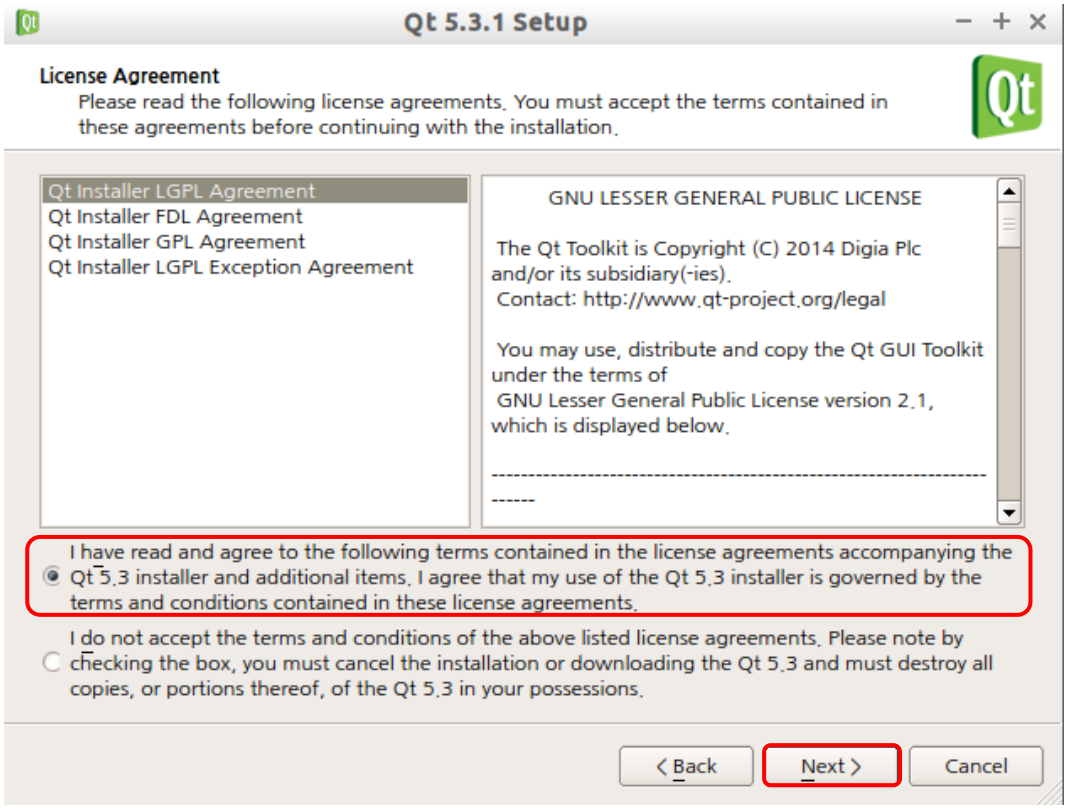
<Figure 5-4>



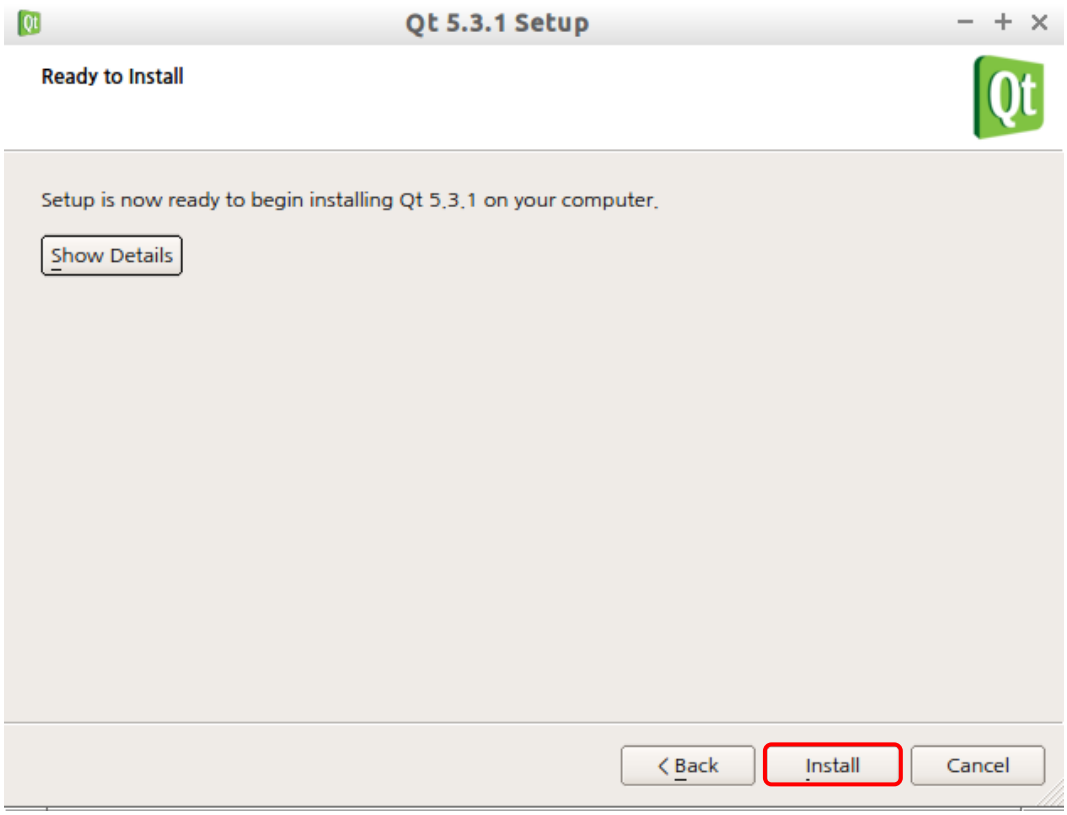
<Figure 5-5>



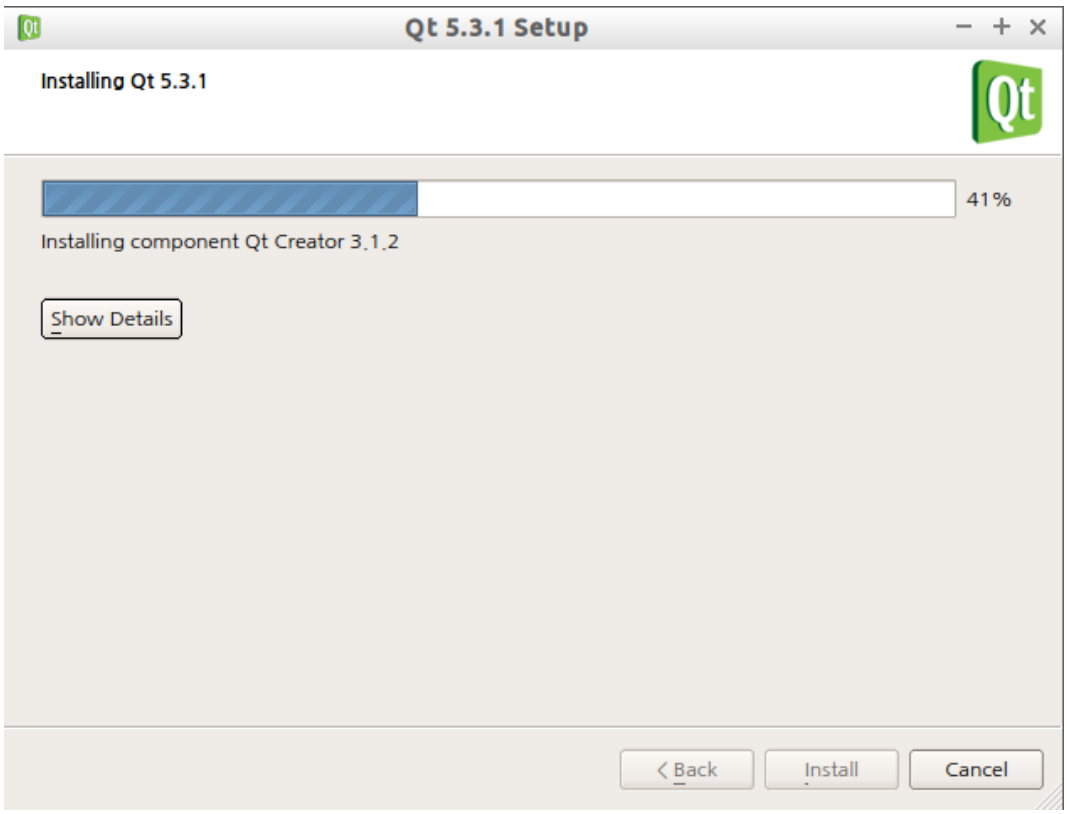
<Figure 5-6>



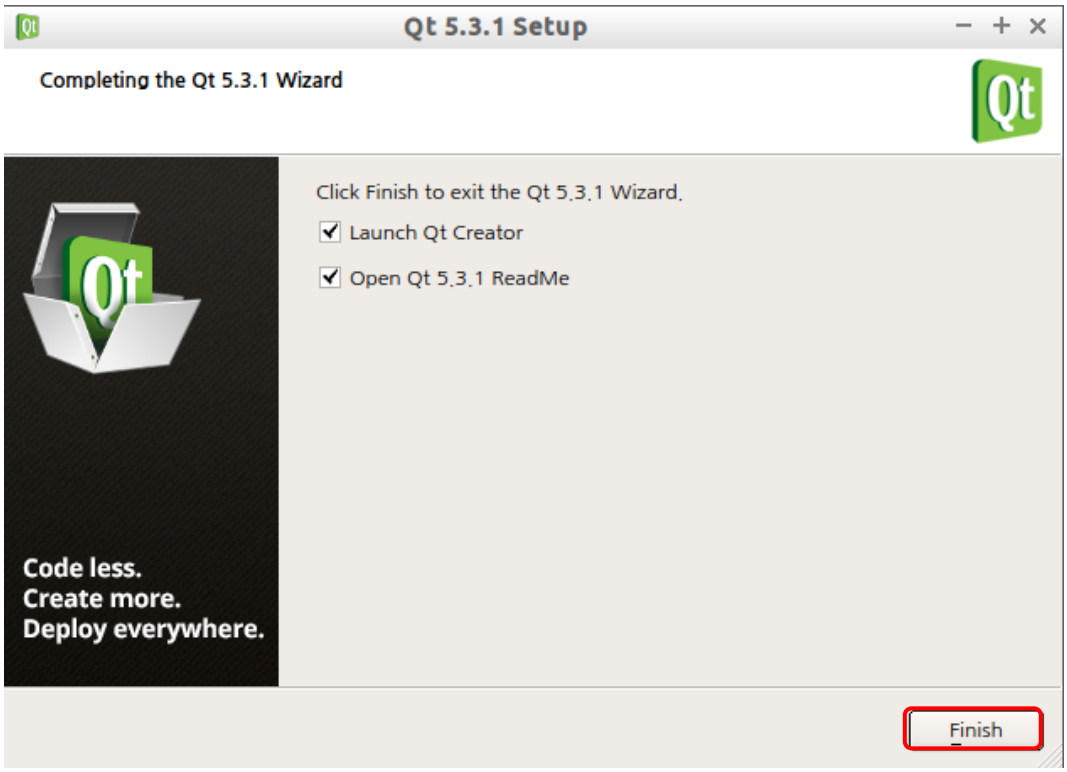
<Figure 5-7>



<Figure 5-8>

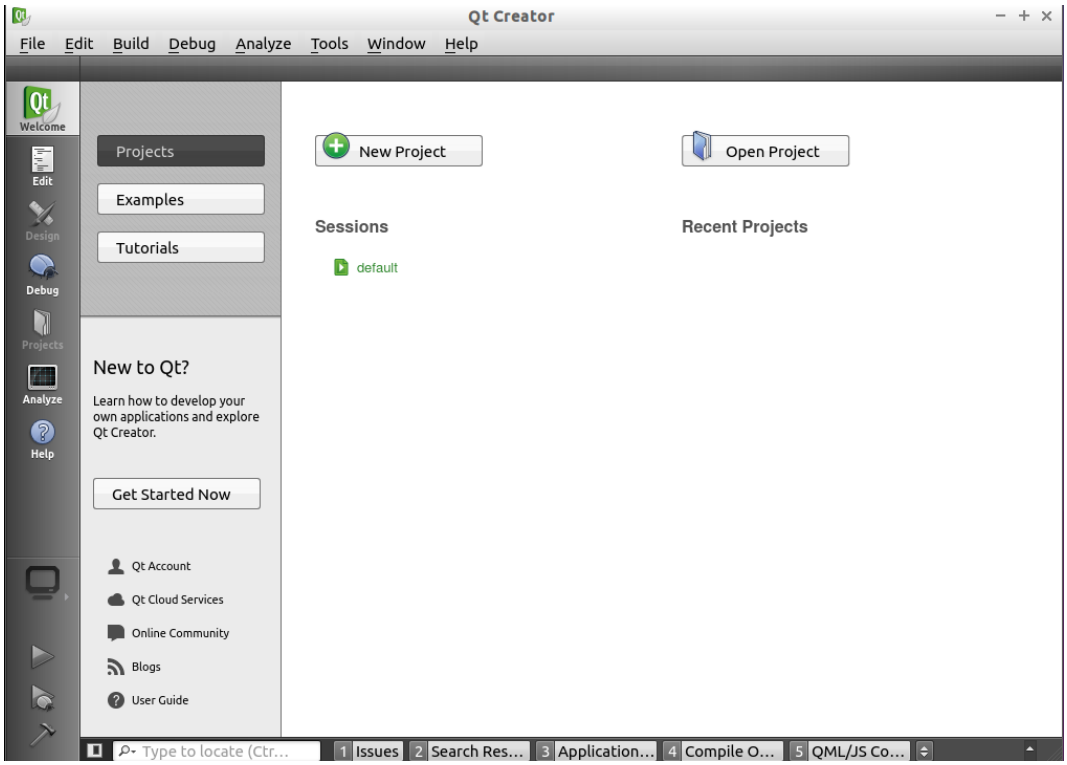


<Figure 5-9>



<Figure 5-10>

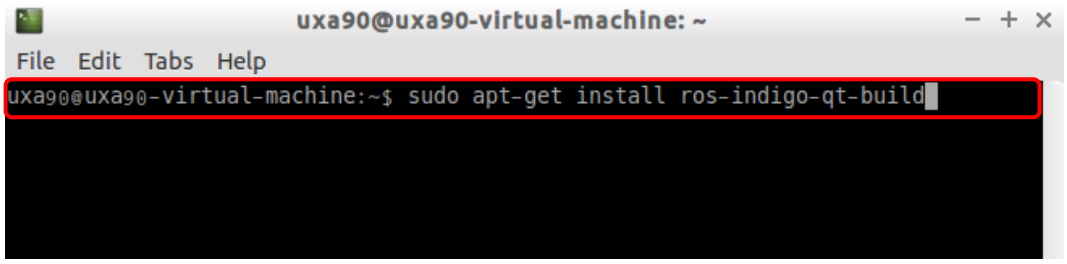
Once the installation has been completed, you will see 'Qt Creator' as shown in <Figure 5-11>. Now, the installation has been finished. Let's try to build a Qt with ROS catkin.



<Figure 5-11>

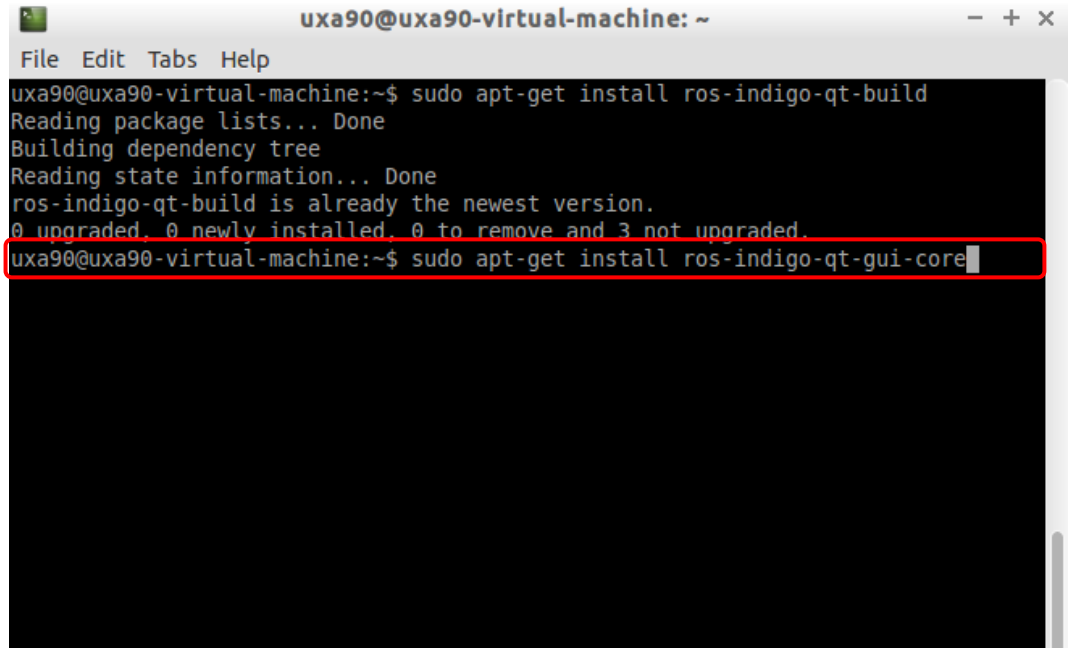
First, open the terminal and install Qt ROS-related packages. There are 4 in total.

$ sudo apt-get install ros-indigo-qt-build



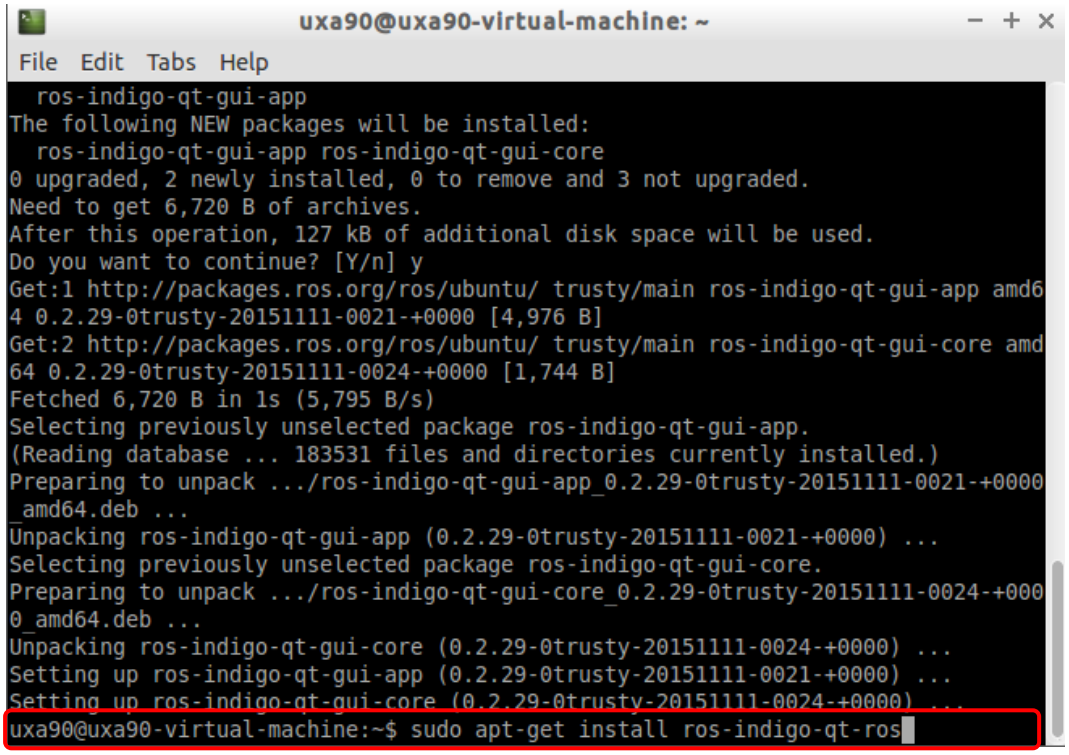
<Figure 5-12>

$ sudo apt-get install ros-indigo-qt-gui-core



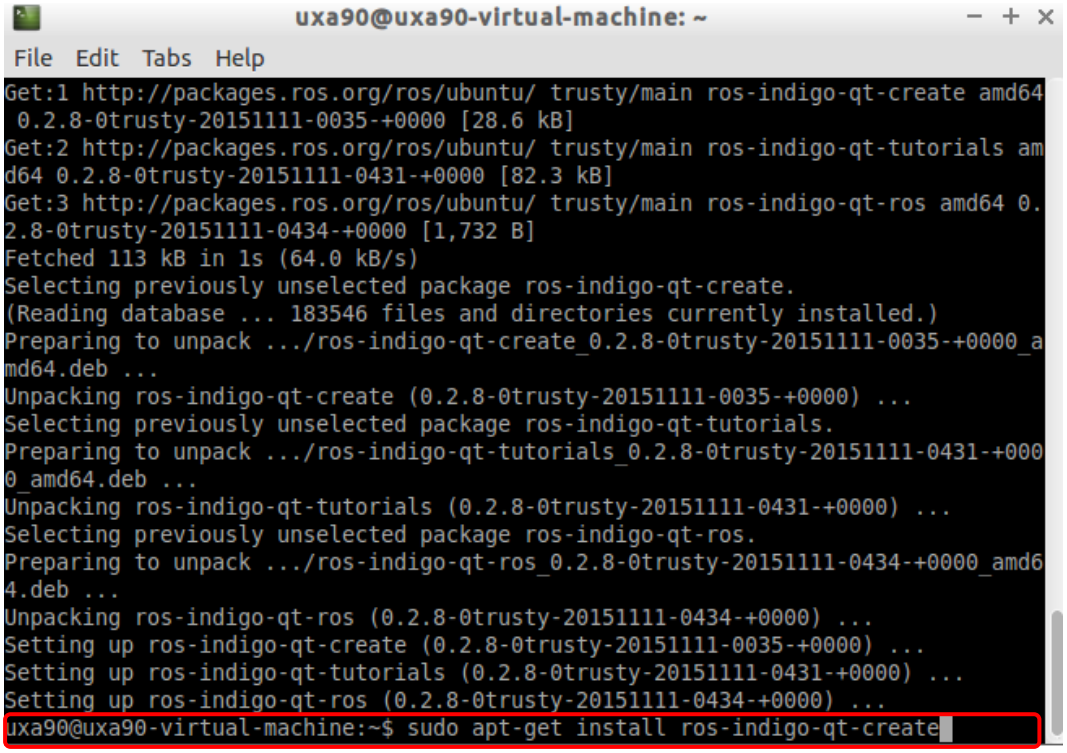
<Figure 5-13>

$ sudo apt-get install ros-indigo-qt-ros



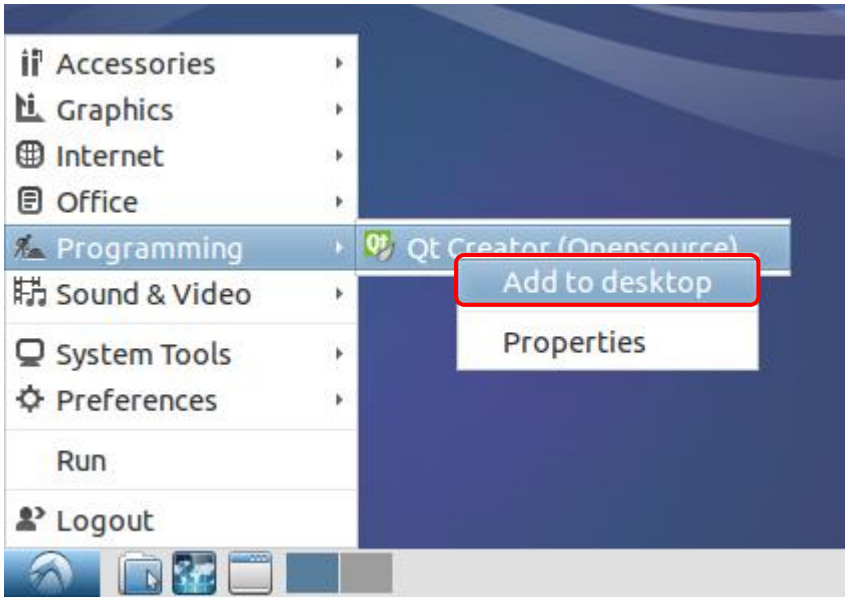
<Figure 5-14>

$ sudo apt-get install ros-indigo-qt-create



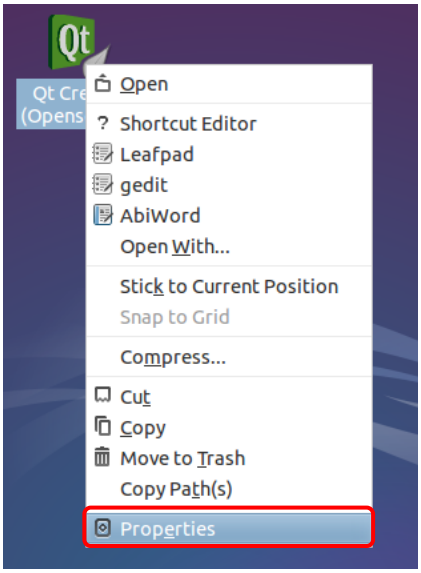
<Figure 5-15>

When the installation is finished, move on to configuration. Add 'Qt Creator' program to Desktop as shown in <Figure 5-16>. If you don't see 'Qt Creator', restart the computer.



<Figure 5-16>

Right click the icon created on the Desktop to enter Properties.

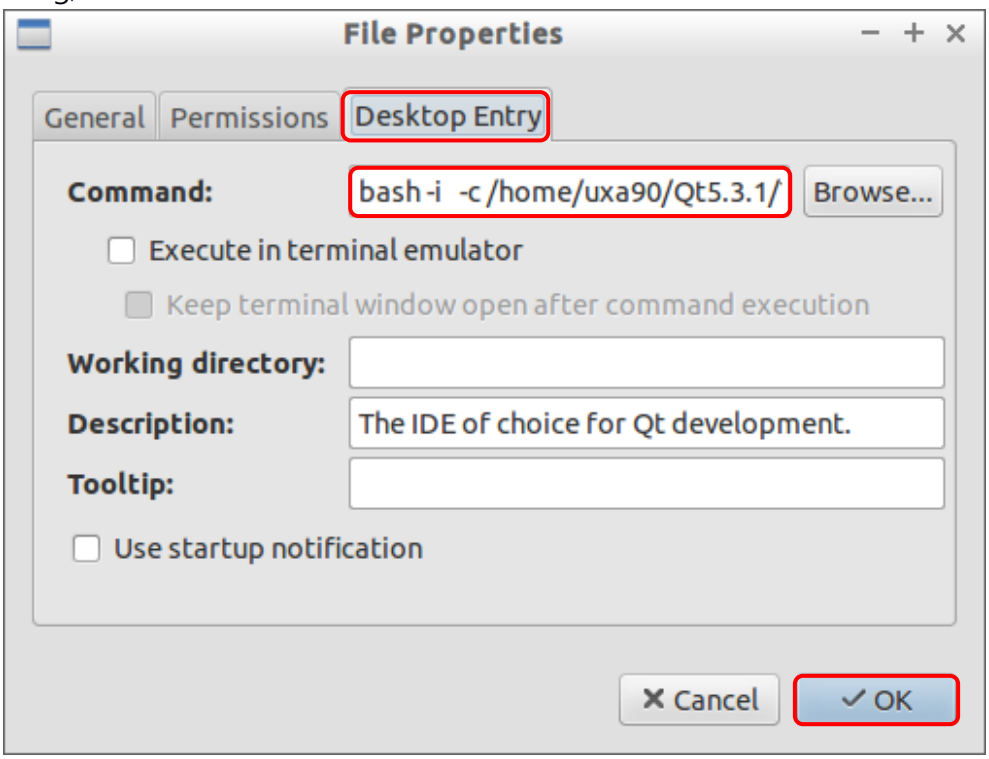


<Figure 5-17>

When you see the screen as in <Figure 5-18>, go into the 'Desktop Entry' tab and modify 'Command'.

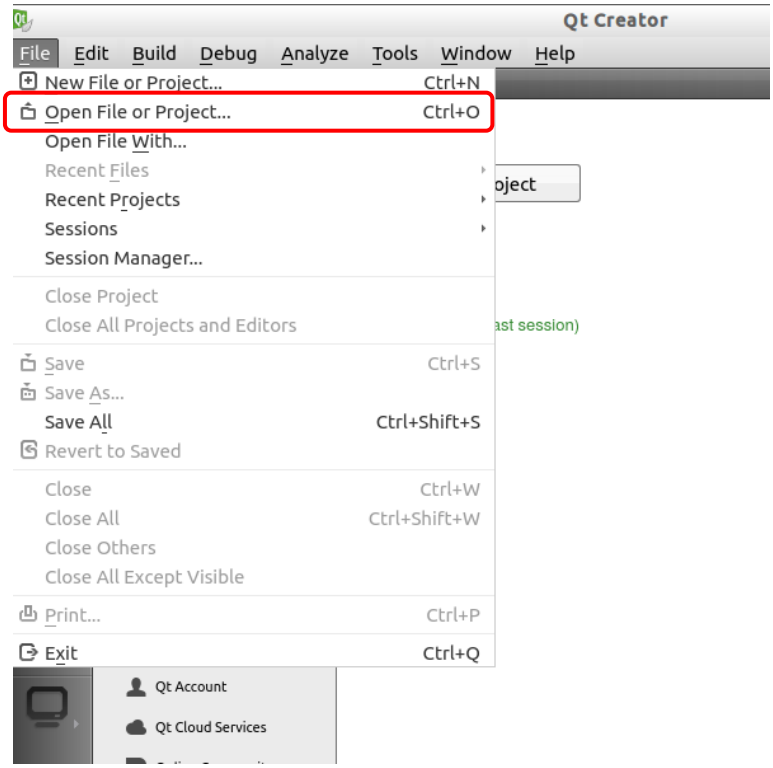
By default, it will say '/home/(username)/Qt5.3.1/Tools/QtCreator/bin/qtcreator'. Here, username will be 'uxa90' in this case.

At the beginning and the end, a command needs to be added: '**bash –i –c** /home/(username)/Qt5.3.1/Tools/QtCreator/bin/qtcreator **%F**'. That is, add 'bash –i –c' at the beginning, and '%F' at the end.



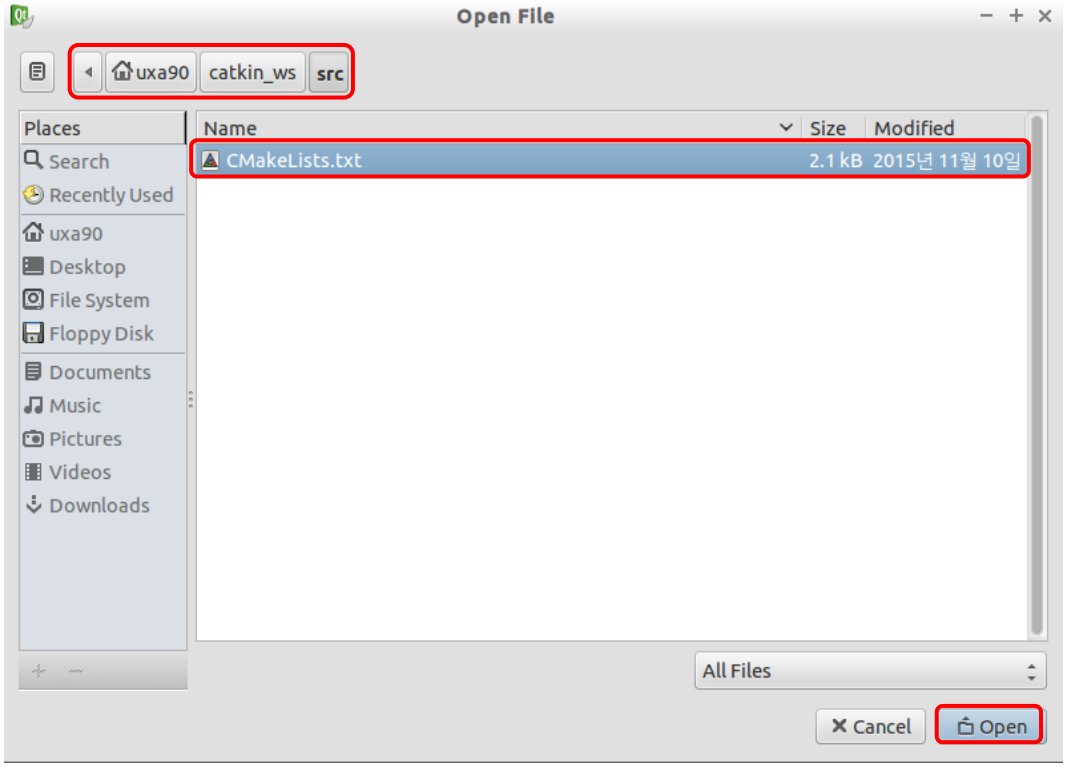
<Figure 5-18>

Once completed, double click the icon to run the program. And then select 'File' → 'Open File or Project' as shown in <Figure 5-19>.



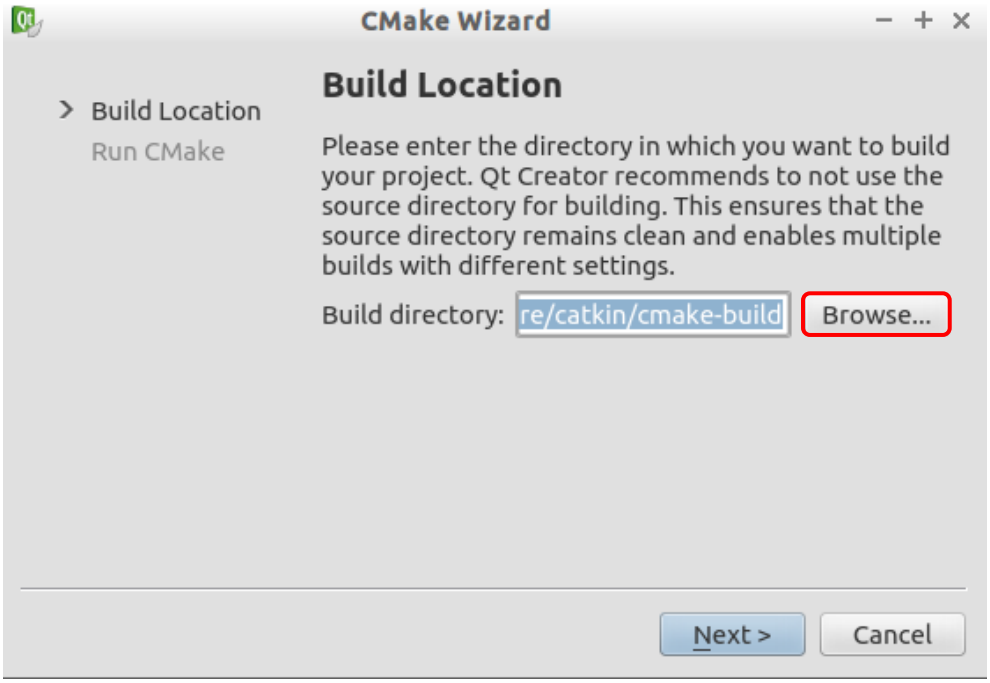
<Figure 5-19>

Next, move to '/home/(username)/catkin_ws/src' and open 'CMakeLists.txt'.

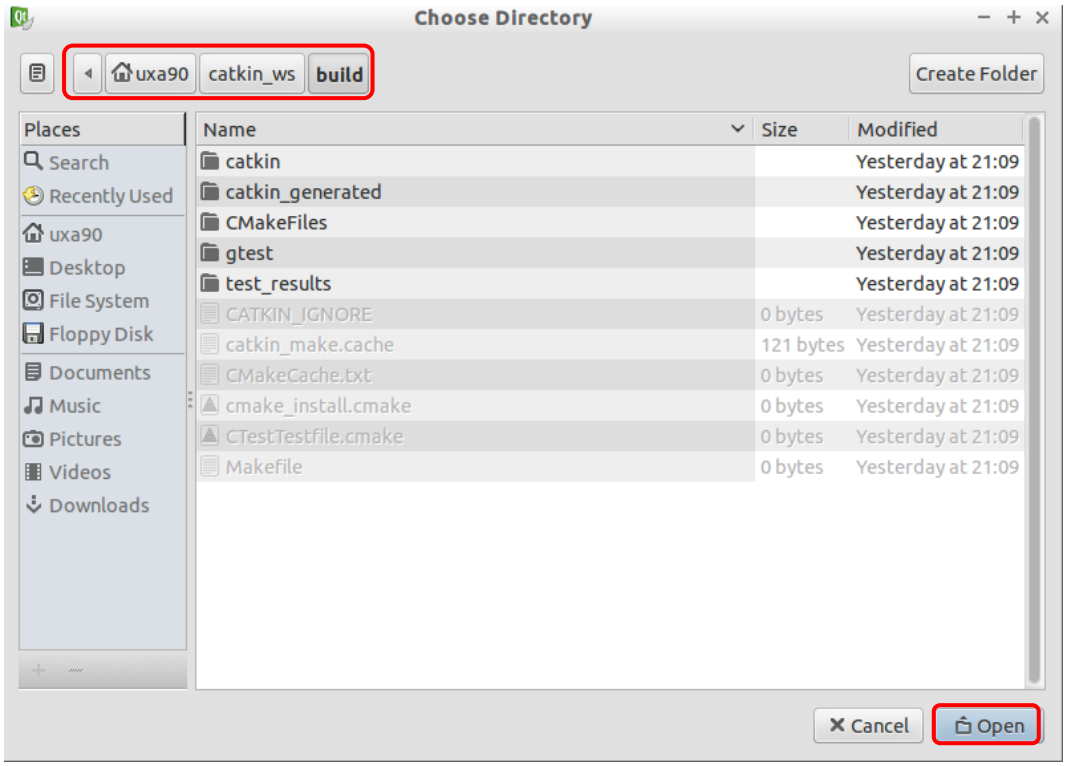


<Figure 5-20>

If you open the file, 'CMake Wizard' will open like <Figure 5-21>. Here, edit 'Build directory'. Click 'Browse...' and change the location to '/home/(username)/catkin_ws/build'. Click 'Open' and then click 'Next'.

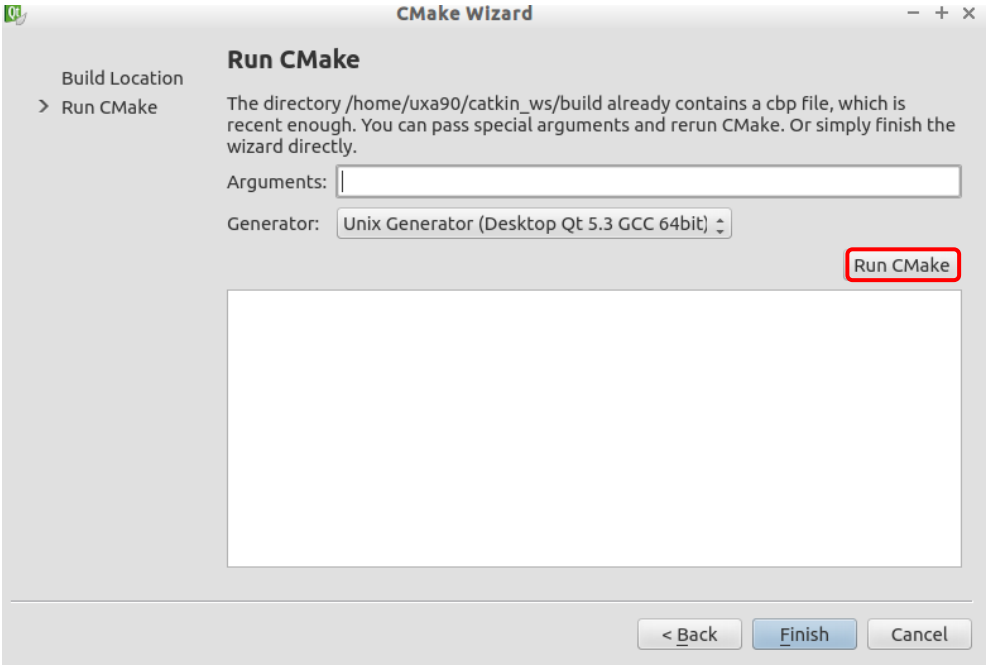


<Figure 5-21>

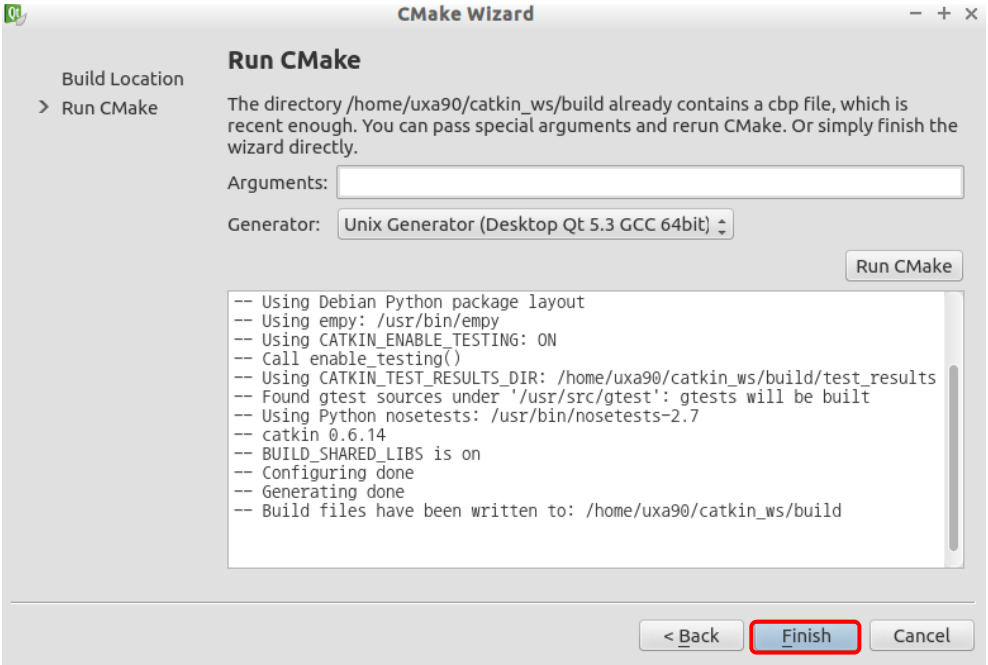


<Figure 5-22>

When you click 'Next', you will see the window as shown in <Figure 5-23>. Click 'Run CMake' and if you don't see an error like <Figure 5-24>, go on and click 'Finish'.

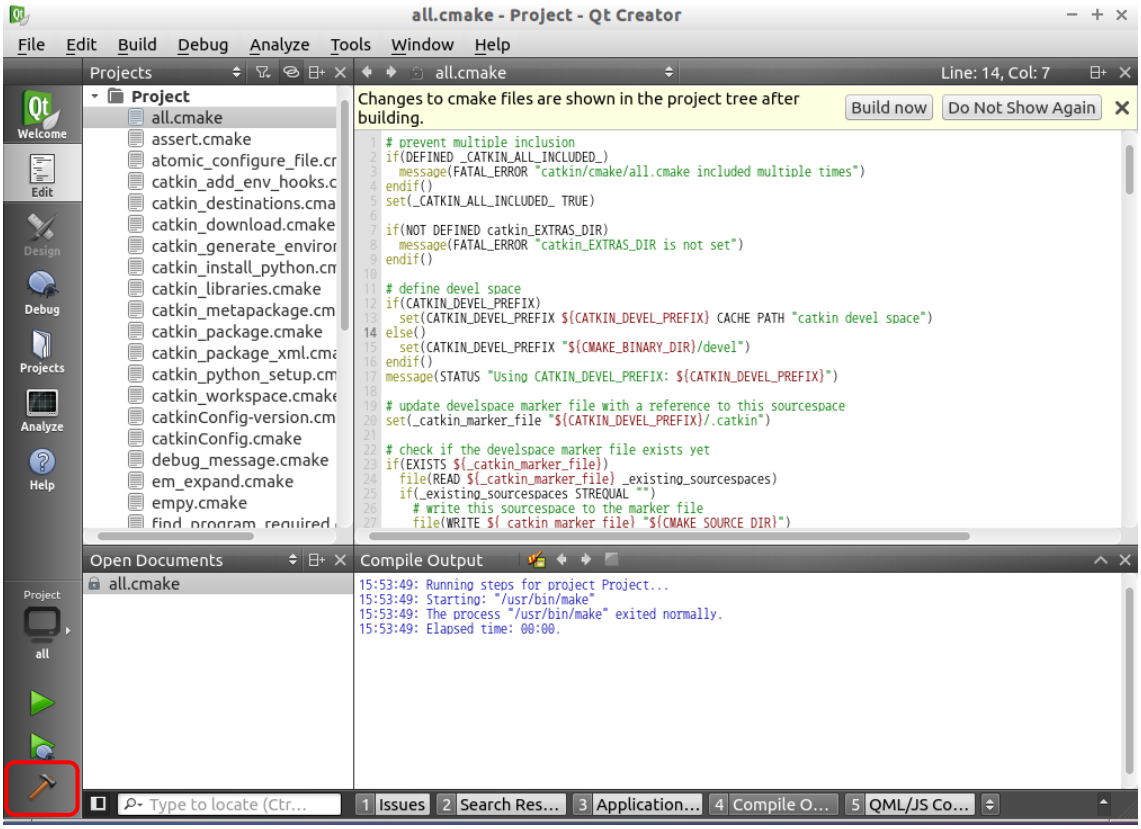


<Figure 5-23>



<Figure 5-24>

After you click 'Finish', the CMake file list appears as shown in <Figure 5-25>. Press 'Ctrl + B' or click the hammer tool on the bottom left as shown in <Figure 5-25> to build. If you don't see an error, you are good to use ROS in Qt.



<Figure 5-25>

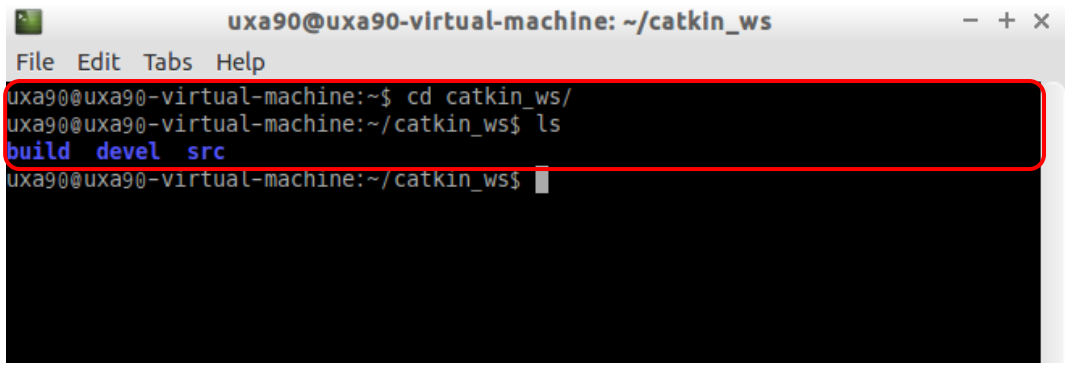
5. QT project creation

This chapter will explain how to create a ROS project using Qt installed in Chapter 5.

First, take a look at the workspace created in Chapter 3 while installing ROS. Run the terminal and change the location to look at the inside the workspace.

$ cd ~/catkin_ws

$ ls

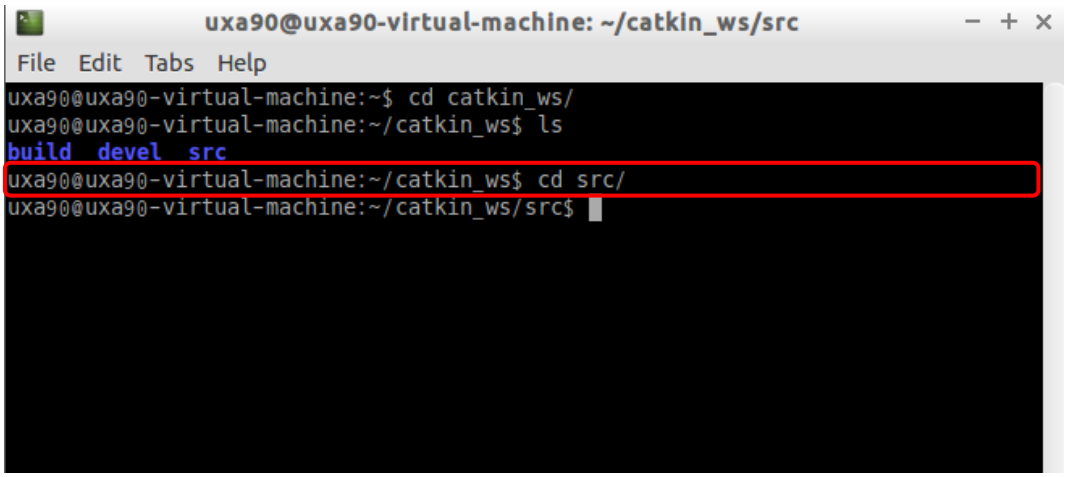


<Figure 6-1>

There are 3 folders inside the workspace: the build environment file in 'build', msg and srv header file, user library, and application file in 'devel', and user package file in 'src'.

First, to create a project, move to 'src'.

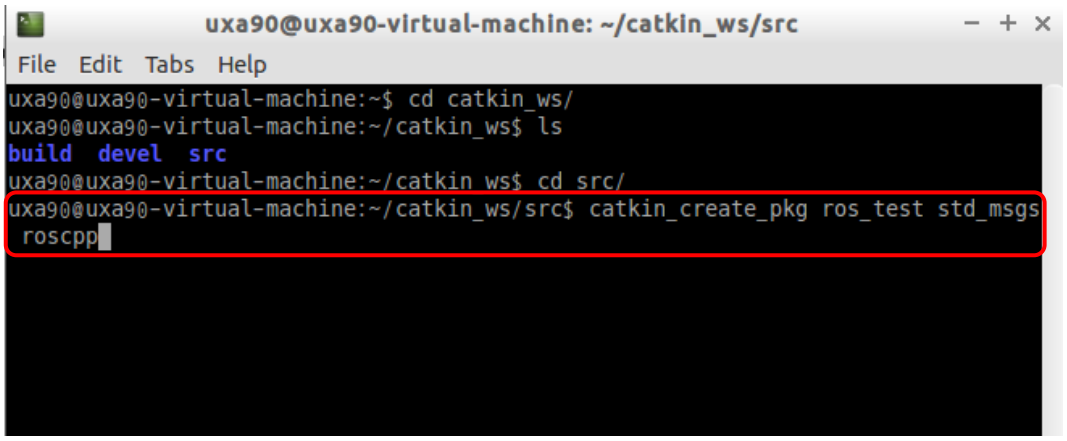
$ cd src



<Figure 6-2>

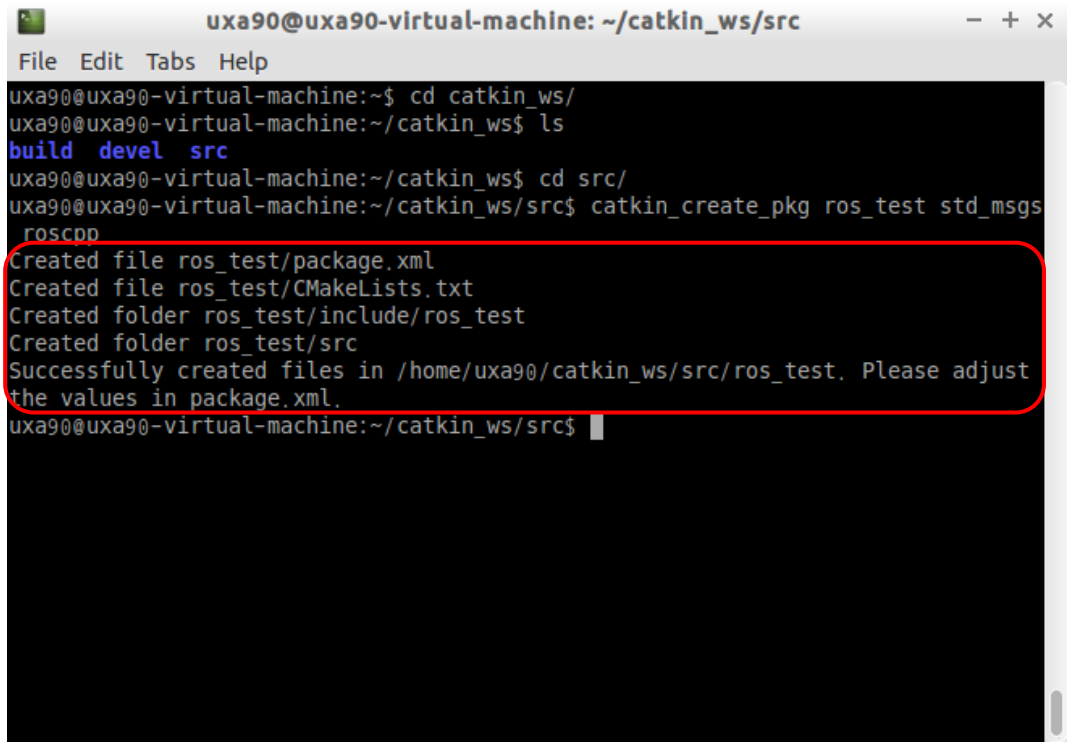
To create a package, use the command 'catkin_create_pkg'. You can create a package by entering the package name and dependent packages. In this example, the package name will be 'ros_test', and 'std_smgs' and 'roscpp' will be added as options. This is to use the ROS standard message package and the c++ client library.

$ catkin_create_pkg ros_test std_msgs roscpp



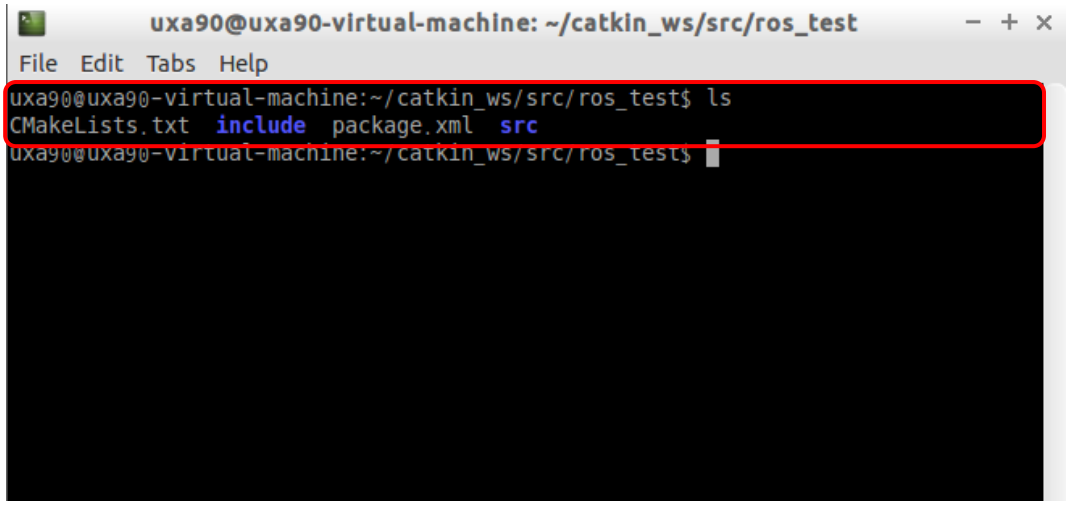
<Figure 6-3>

If you see the output as <Figure 6-4>, it means the package was created successfully.



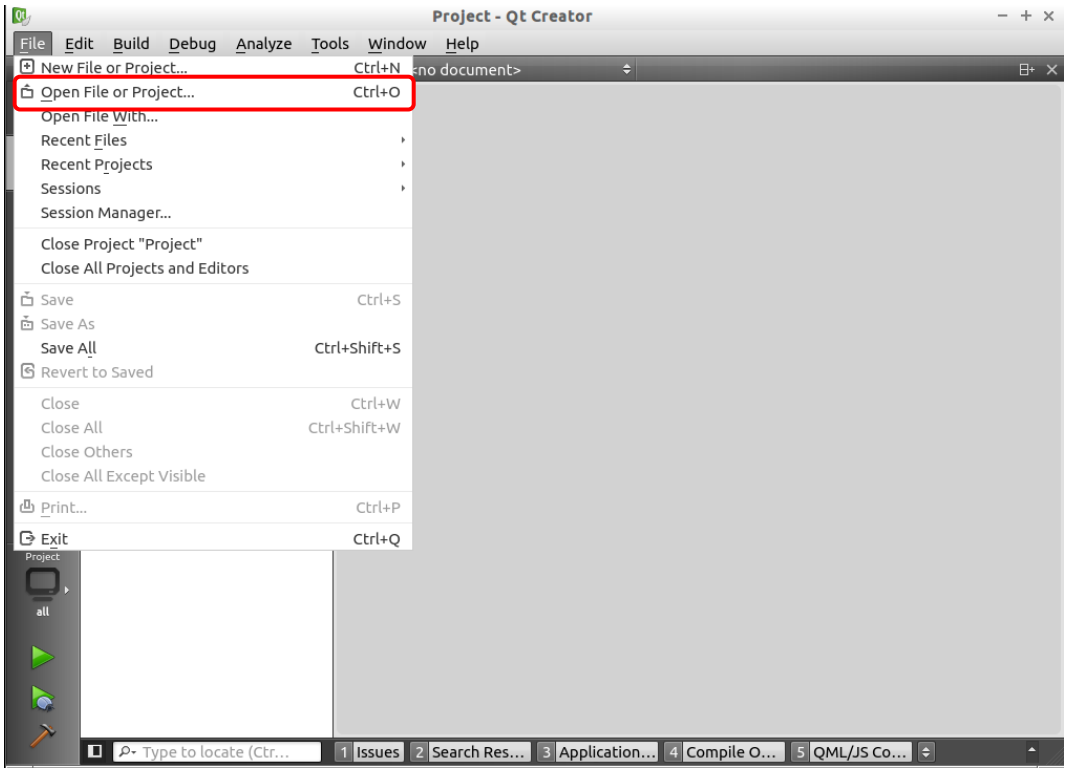
<Figure 6-4>

Refer to <Figure 6-5> to take a brief look at the package that has been created. 'CMakeLists.txt' is the build setting file, 'src' is the source code folder, and 'package.xml' is the package setting file.



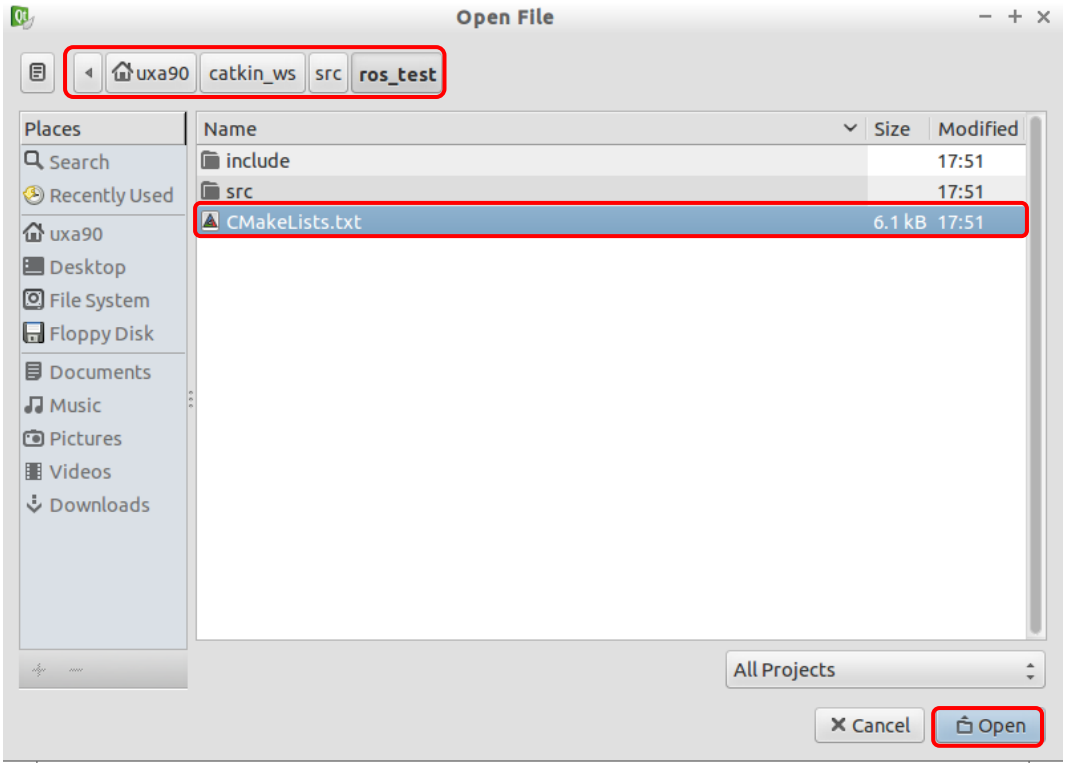
<Figure 6-5>

Once the package has been created successfully, try opening the package in Qt. As shown in <Figure 6-6>, go to 'File' → ' Open File or Project'.



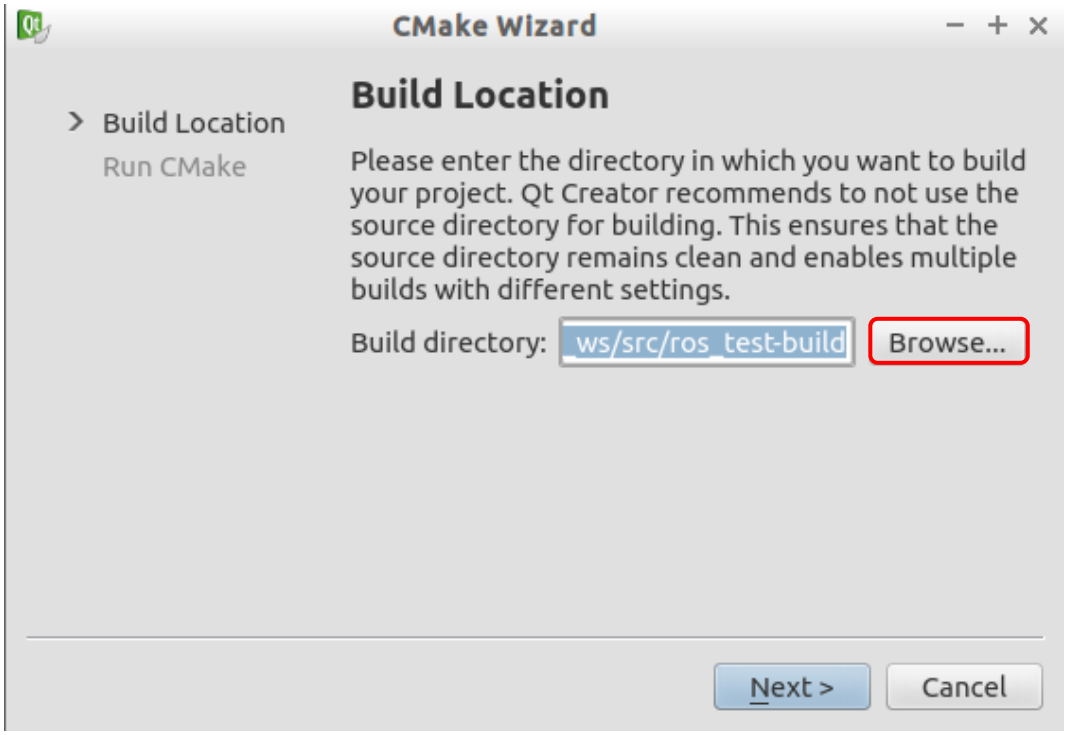
<Figure 6-6>

When the 'Open File' window appears, go to the location where the new package was created and open 'CMakeFile.txt'. See <Figure 6-7>. The location is '~/catkin_ws/src/ros_test'.



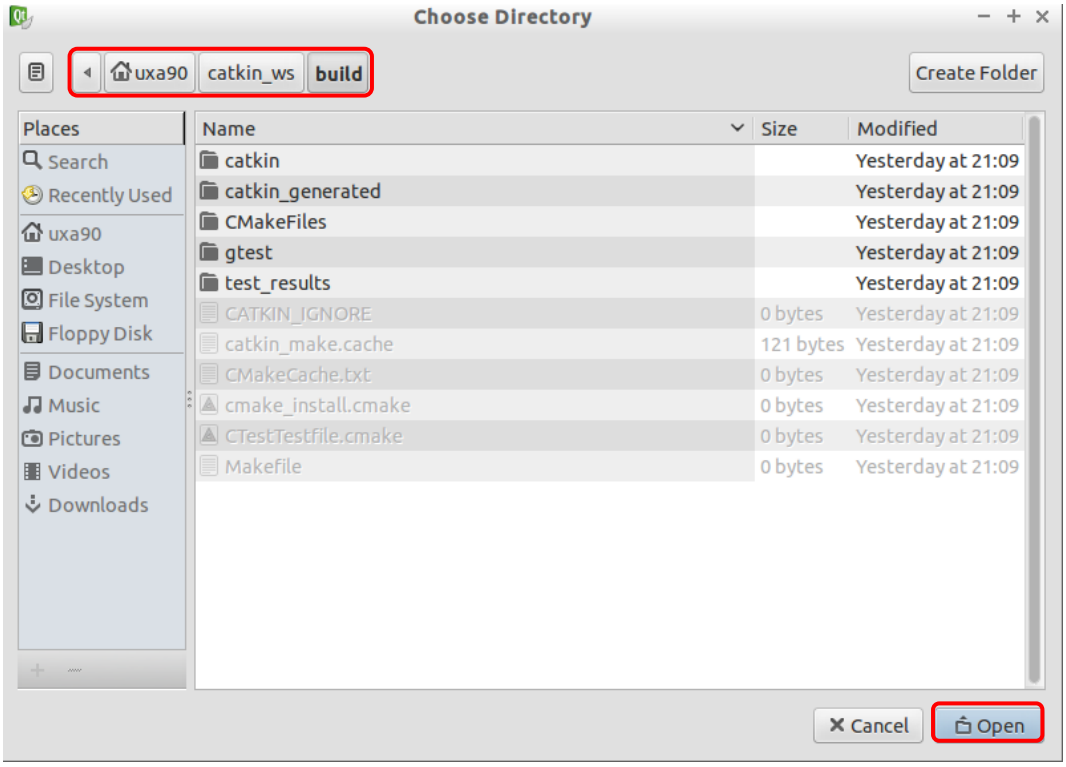
<Figure 6-7>

When you click 'Open' in <Figure 6-7>, the window in <Figure 6-8> appears. Here you have to modify the directory. Click 'Browse...' to edit.



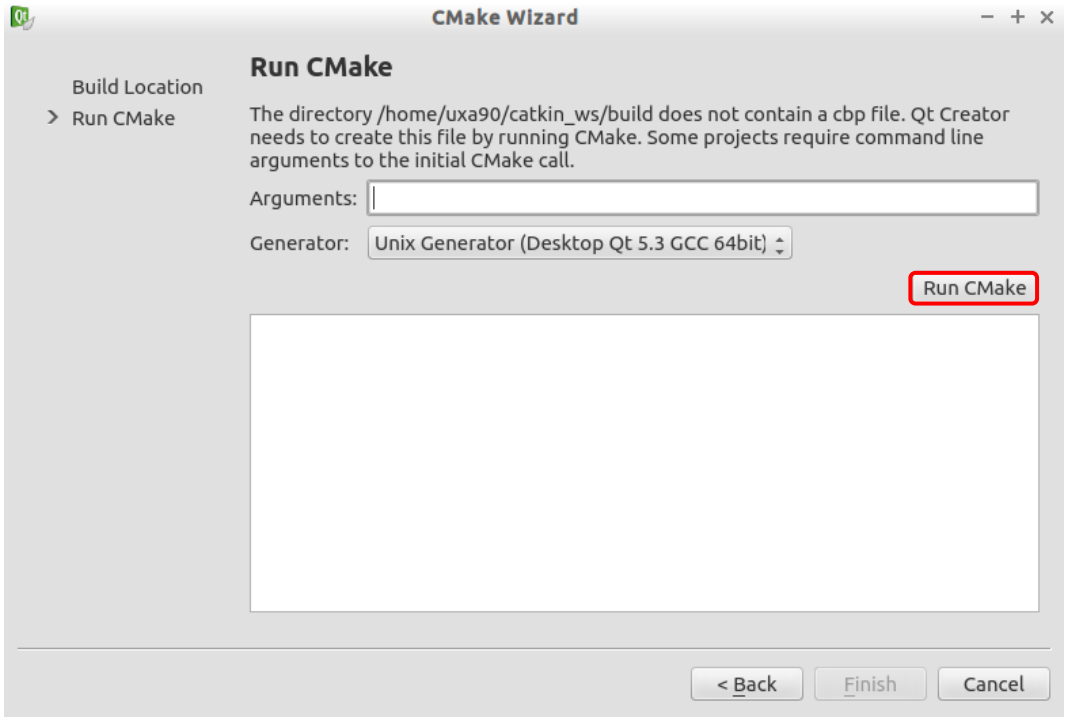
<Figure 6-8>

Edit the directory as shown in <Figure 6-9> to '~/catkin_ws/build/'.



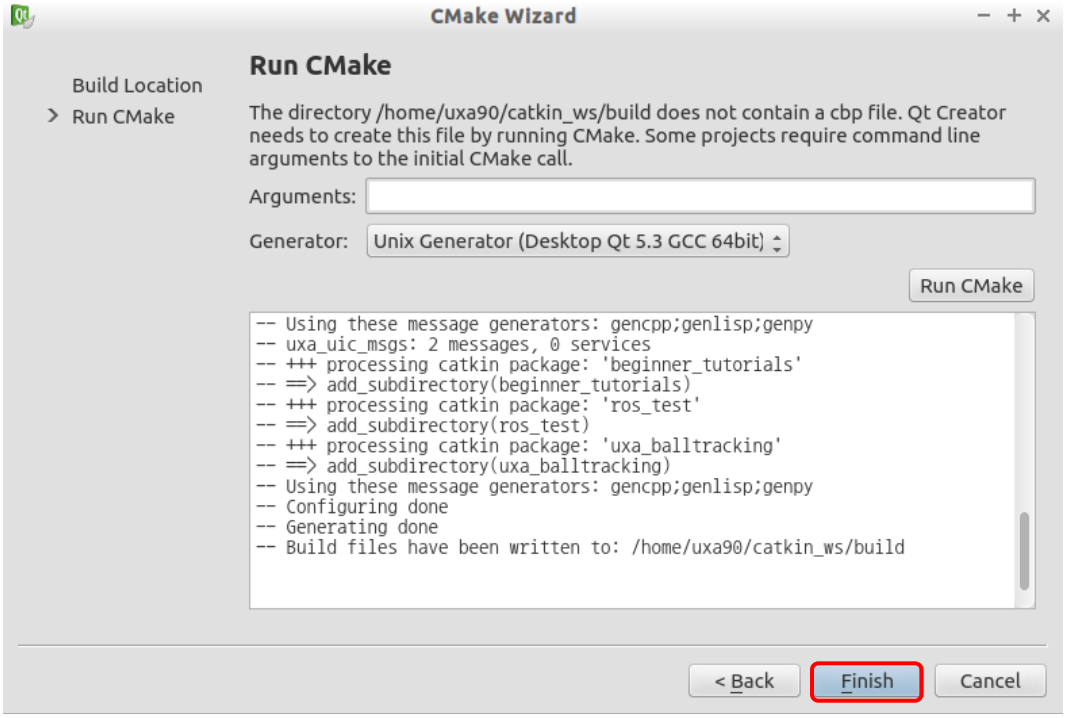
<Figure 6-9>

Once the window in <Figure 6-10> appears, click 'Run CMake' to continue.



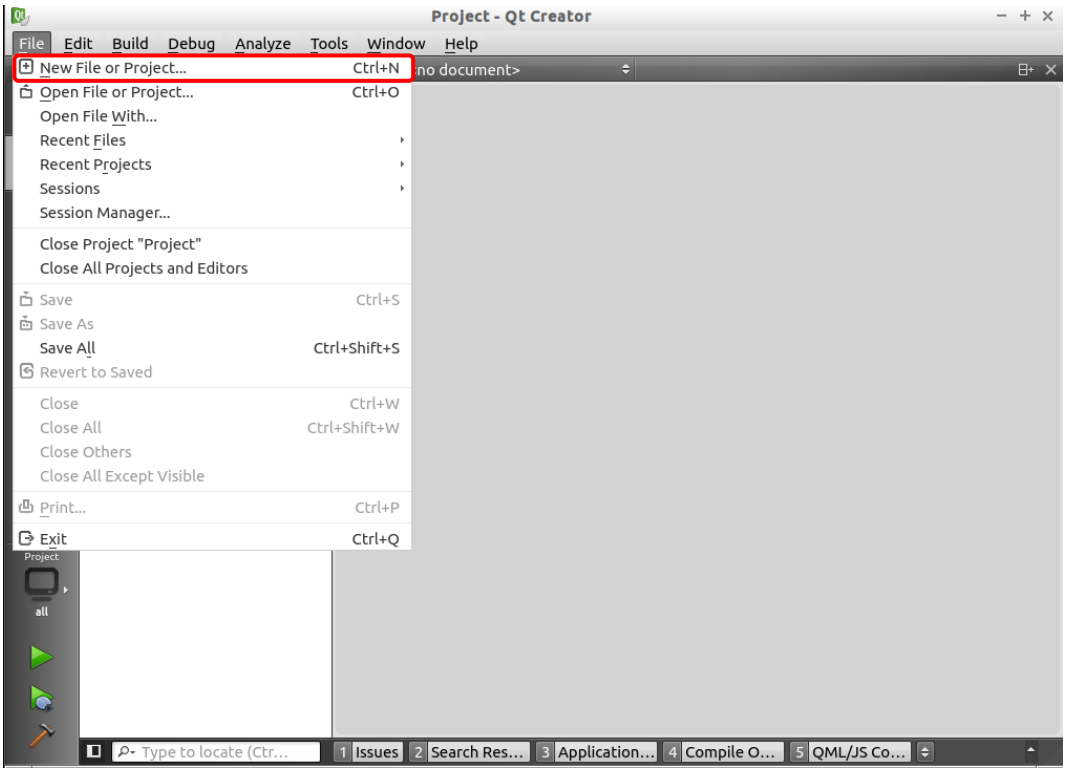
<Figure 6-10>

If <Figure 6-11> appears, you have done it successfully. Click 'Finish' to close 'CMake Wizard'.



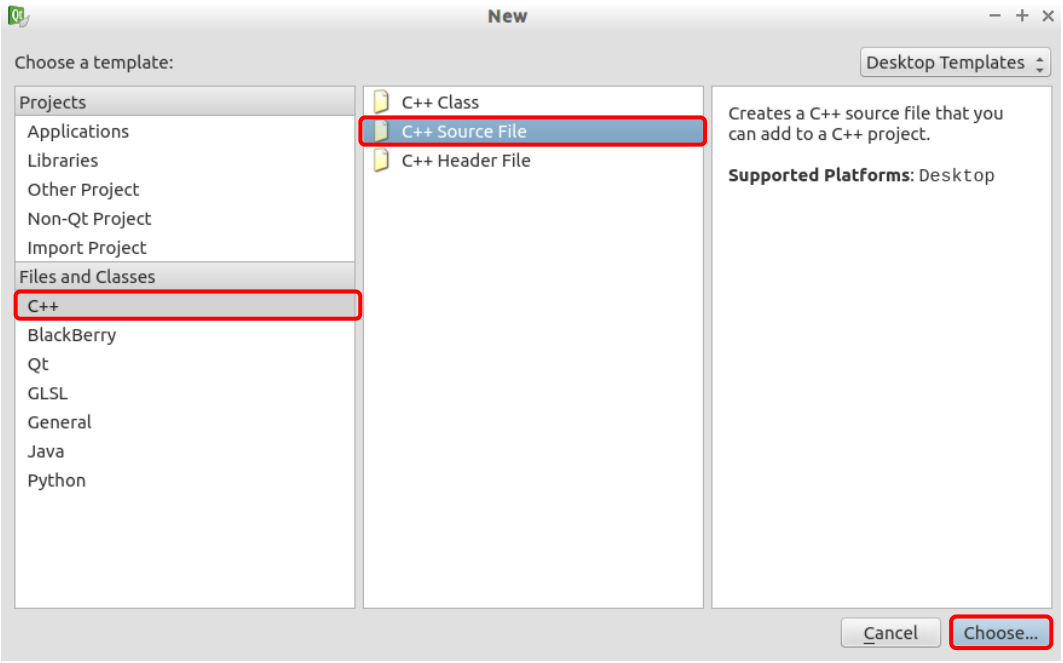
<Figure 6-11>

If you have opened the package, let's create a .cpp file for source code. Go to 'File' → 'New File or Project'.



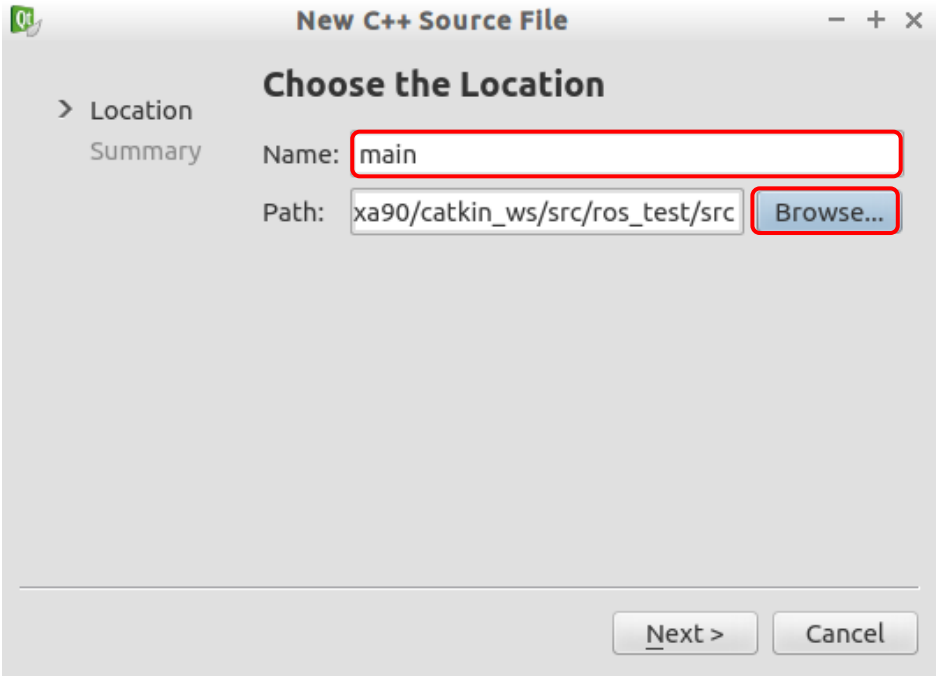
<Figure 6-12>

From the new screen, select 'C++' and then 'C++ Source File'. See <Figure 6-13>.



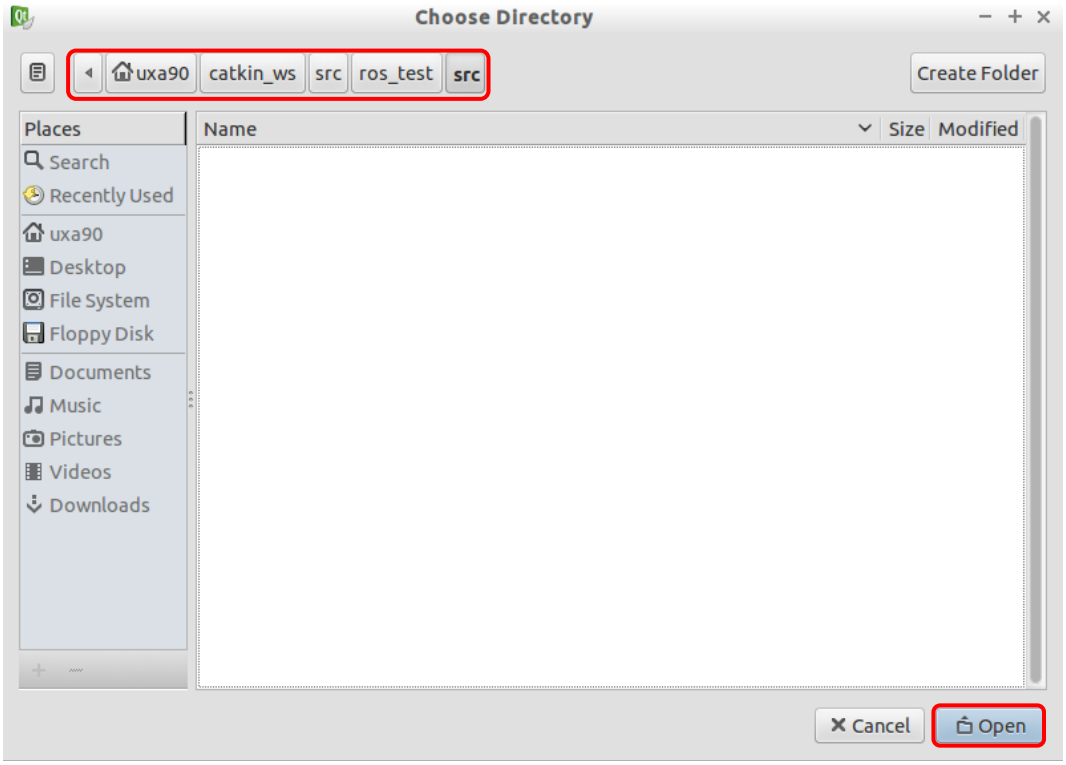
<Figure 6-13>

'Name' is the file name, and 'Path' is the location of the file. In this example, the file name is 'main'.



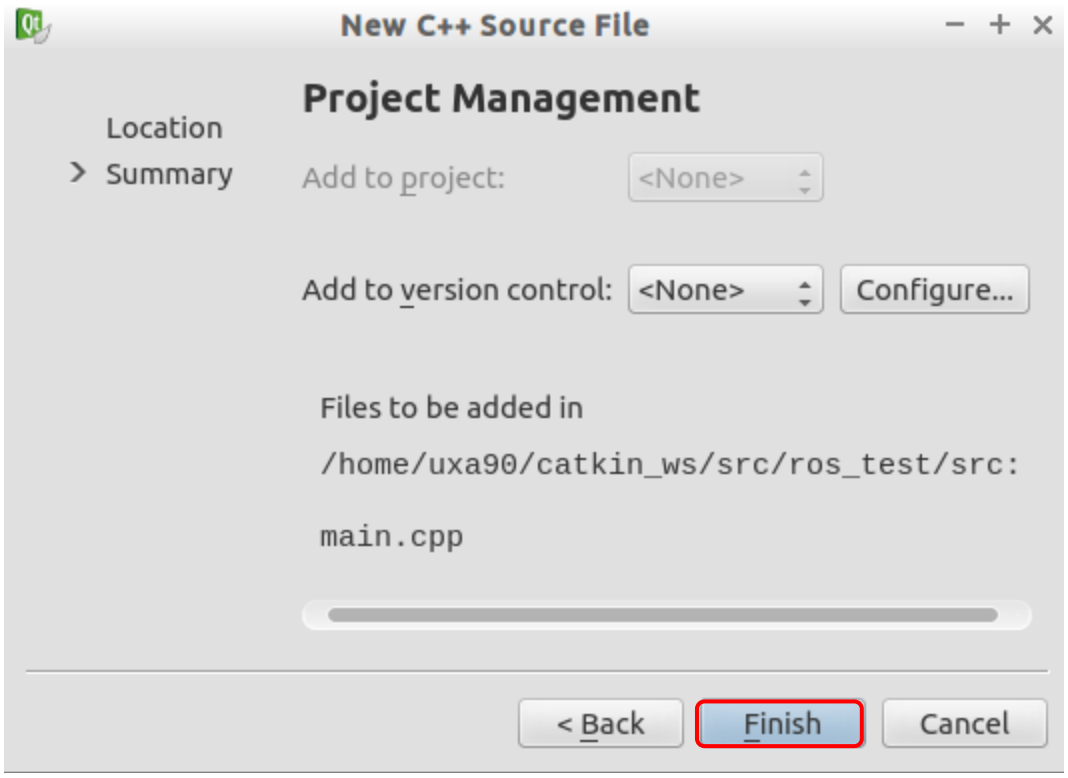
<Figure 6-14>

Set the path as shown in <Figure 6-15>: '~/catkin_ws/src/ros_test/src'. It is the 'src' folder created inside the new package folder.



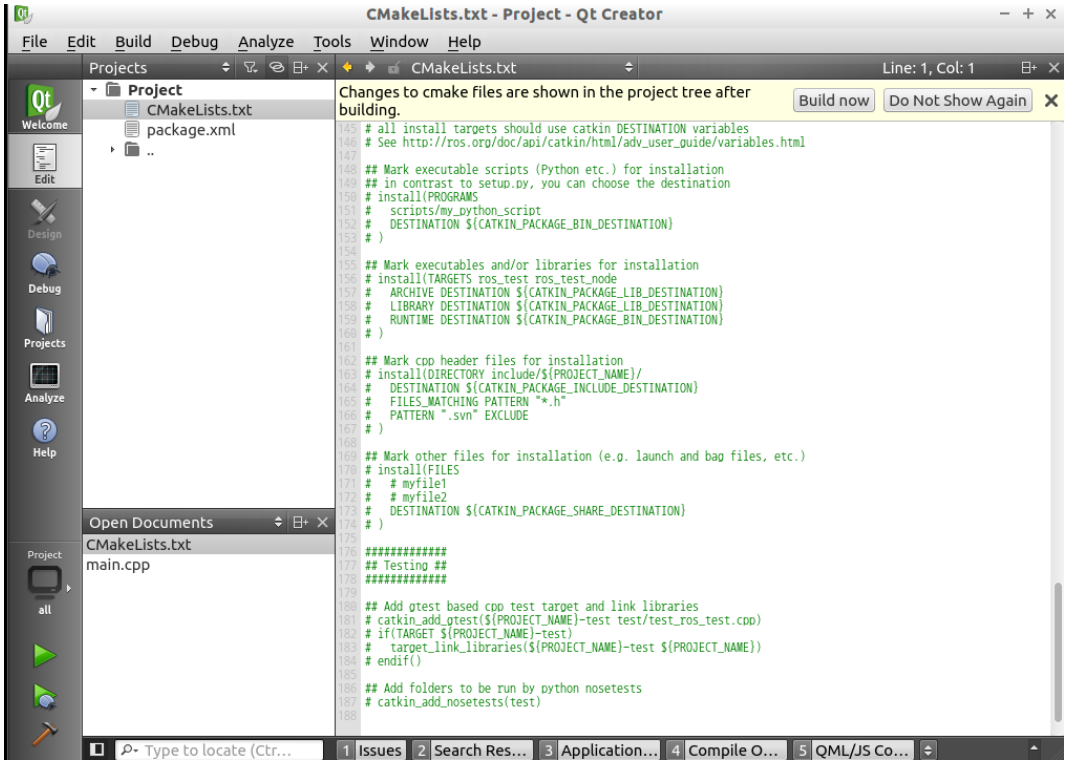
<Figure 6-15>

Check the setting and click 'Finish' to finish creating the .cpp file.



<Figure 6-17>

When the .cpp file has been created, you have to register the .cpp file to 'CMakeLists.txt' file. First, open 'CmakeLists.txt', then you will see <Figure 6-18>.



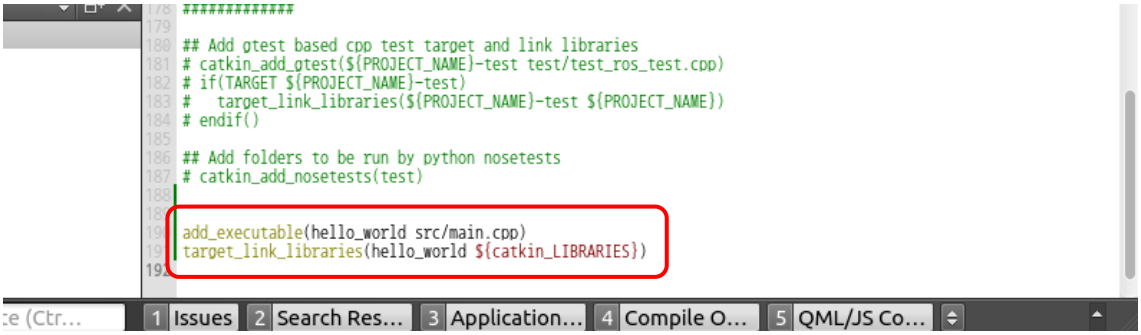
<Figure 6-18>

Add the following code to the very bottom of 'CMakeLists.txt'.

'add_executable' creates a node named 'hello_world' using the file 'main.cpp'. Also, 'target_link_libraries' links the library that needs to be linked before creating a node.

See <Figure 6-19>.

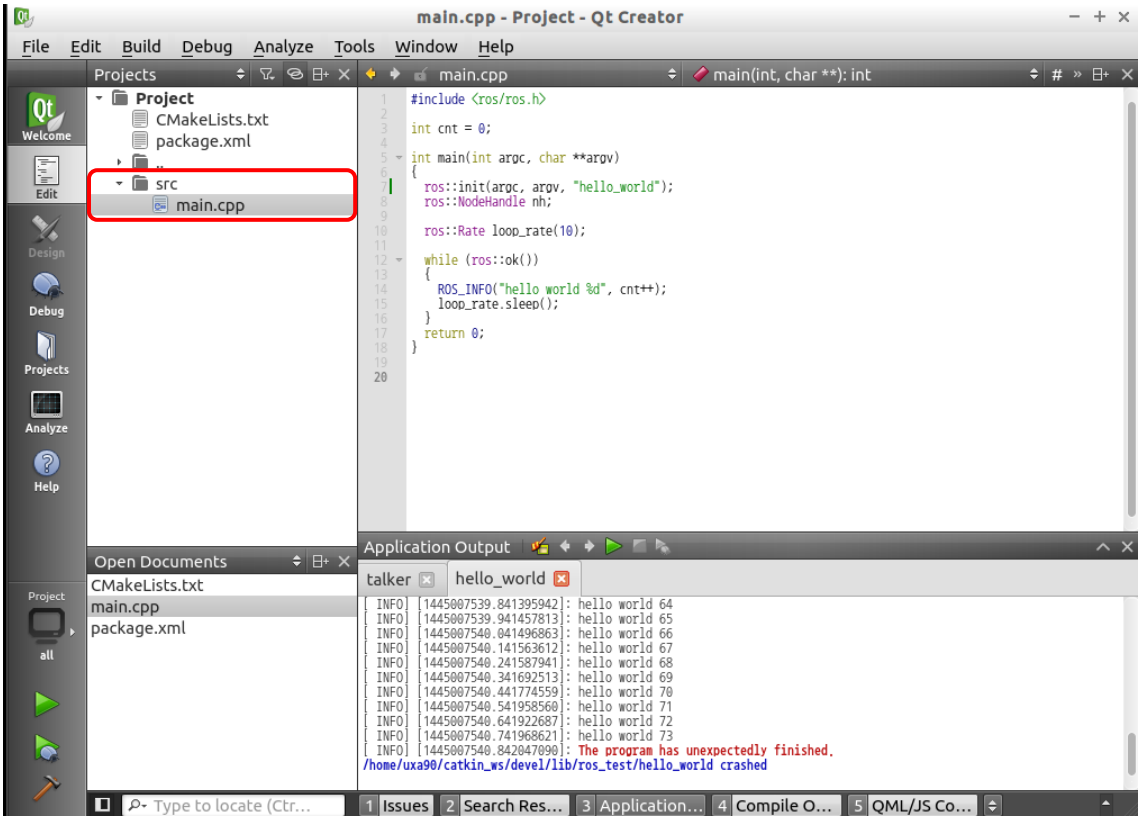
```
add_executable(hello_world src/main.cpp)
target_link_libraries(hello_world ${catkin_LIBRARIES})
```



<Figure 6-19>

Next, let's create a sample code in 'main.cpp'. If you don't see 'main.cpp' on the project list like shown in <Figure 6-20>, edit 'CMakeLists.txt' and restart Qt. Reopen the package.

See the following box for the code.

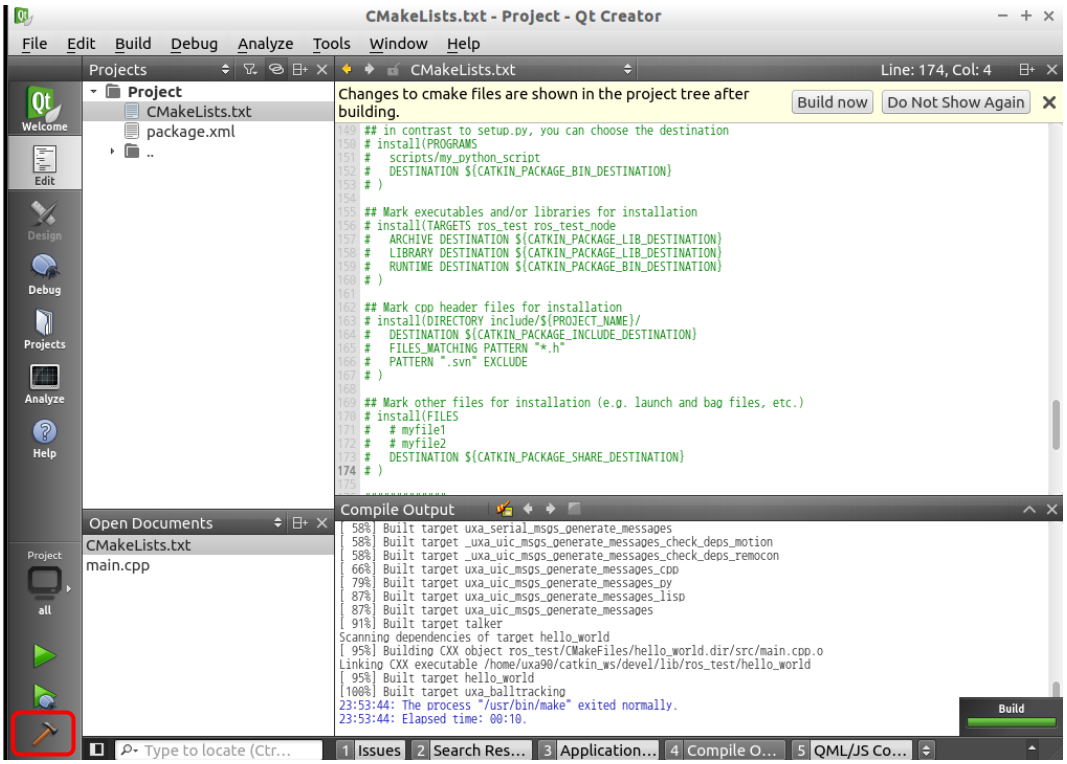


<Figure 6-20>

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

int cnt = 0;

int main(int argc, char **argv)
{
  ros::init(argc, argv, "hello_world_node");
  ros::NodeHandle nh;

  ros::Rate loop_rate(10);

  while (ros::ok())
  {
    ROS_INFO("hello world %d", cnt++);
    loop_rate.sleep();
  }
  return 0;
}
```

If you finished writing the code, go on to build. Press 'Ctrl+B' or click the hammer tool on the bottom left shown in <Figure 6-21> to start build.

It's a success if there is no error.



<Figure 6-21>

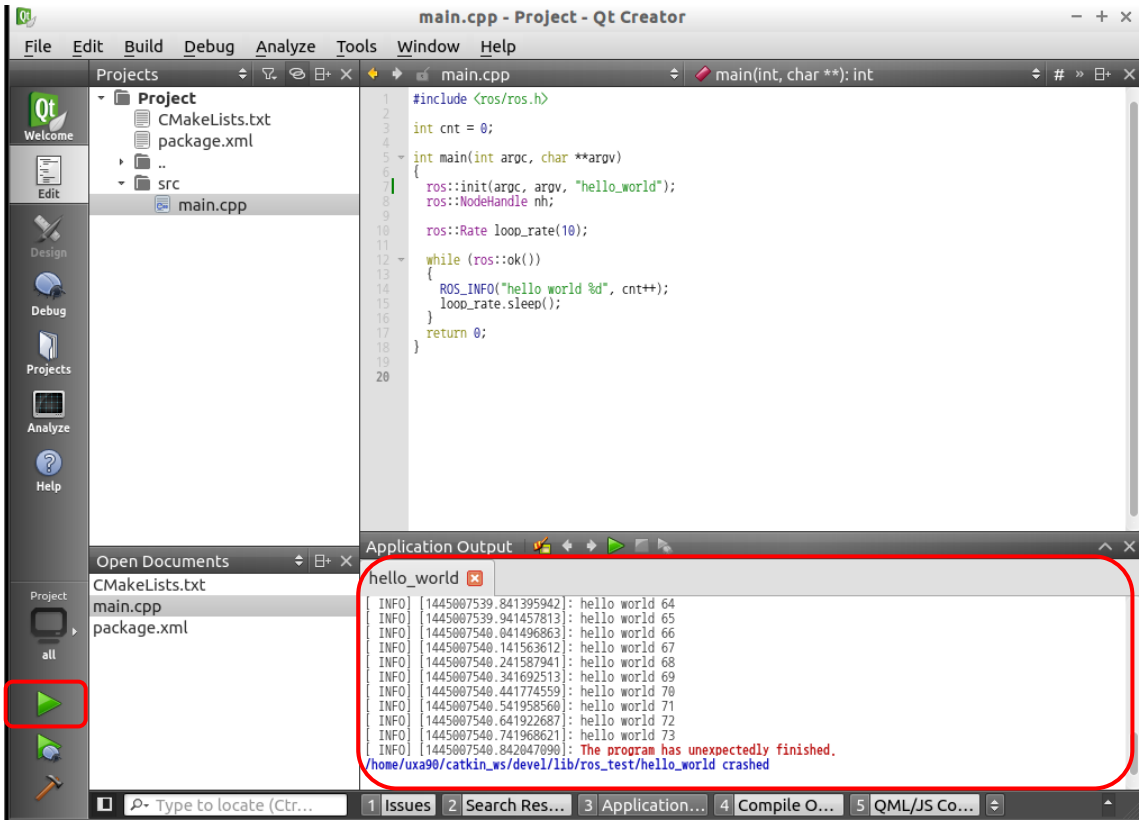
After building, try running. You can either do so through Qt or through the terminal. First is through Qt.

Before anything, run 'roscore' to operate the ROS system. Open the terminal to enter the command.

$ roscore

<Figure 6-22>

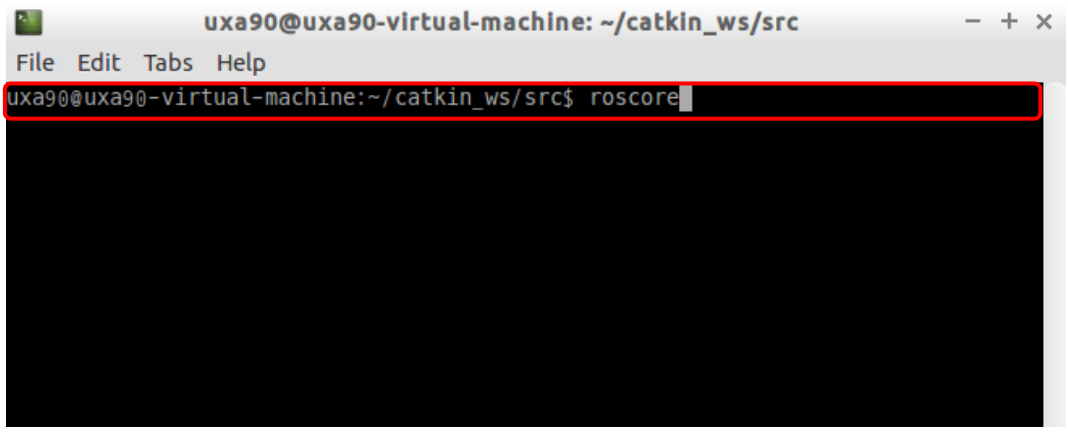
Next, go back to Qt and click the green Play button on the bottom left or press 'Ctrl' + 'R'. Then you will see the result shown in <Figure 6-23>.



<Figure 6-23>

Another method is using the terminal. Using Qt may cause a delay while producing the text as output. So using the terminal is recommended. Just like for Qt, open the terminal and run 'roscore'.

$ roscore

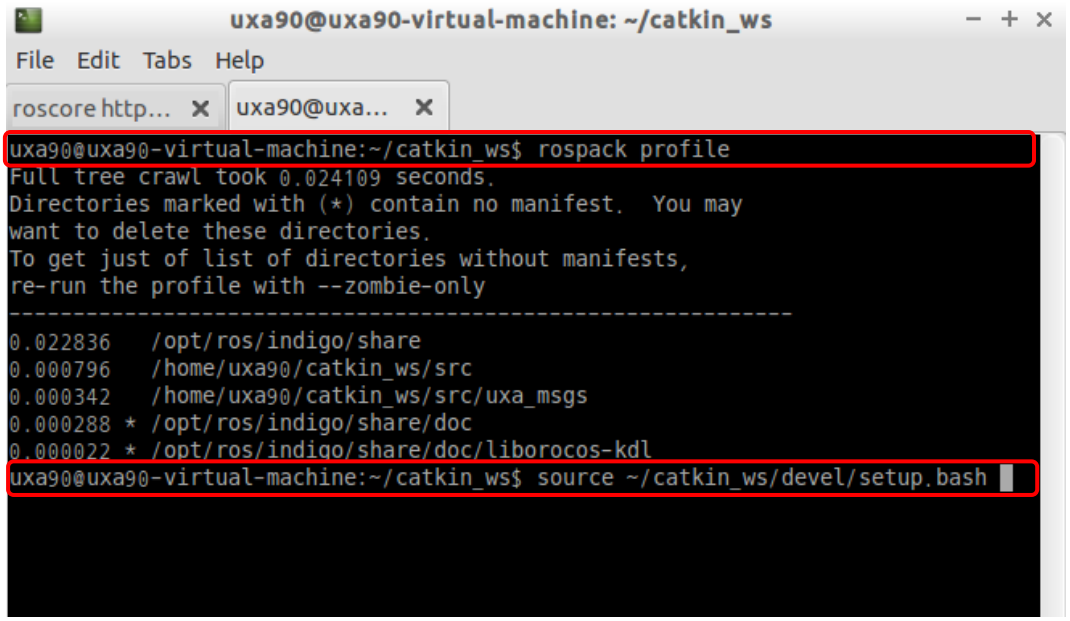


<Figure 6-24>

Next, open a new window and register the bash file to reflect the new package on the ROS package list.

$ rospack profile

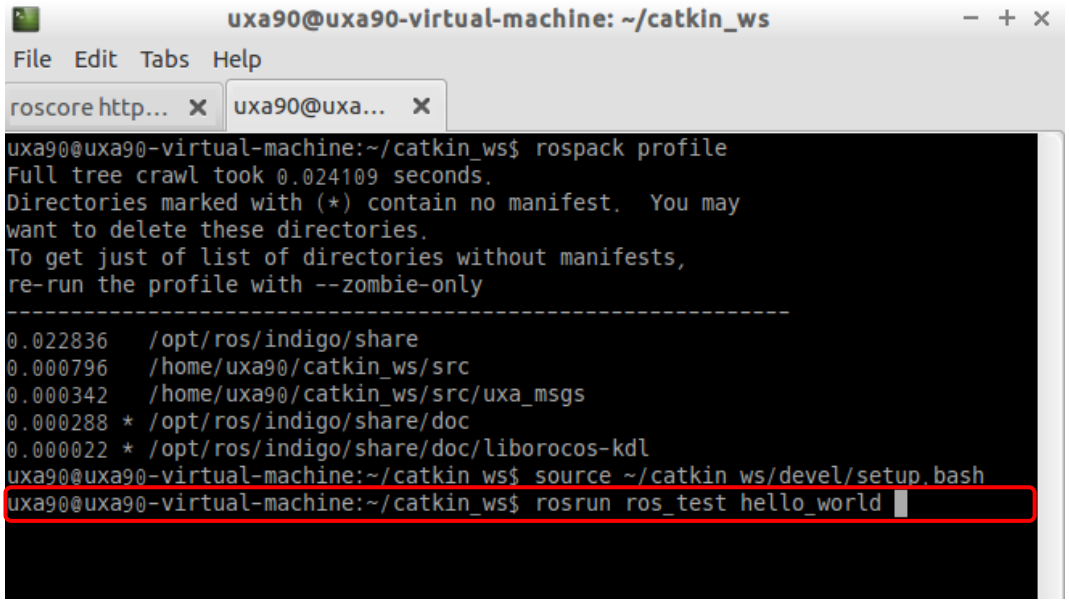
$ source ~/catkin_ws/devel/setup.bash



<Figure 6-25>

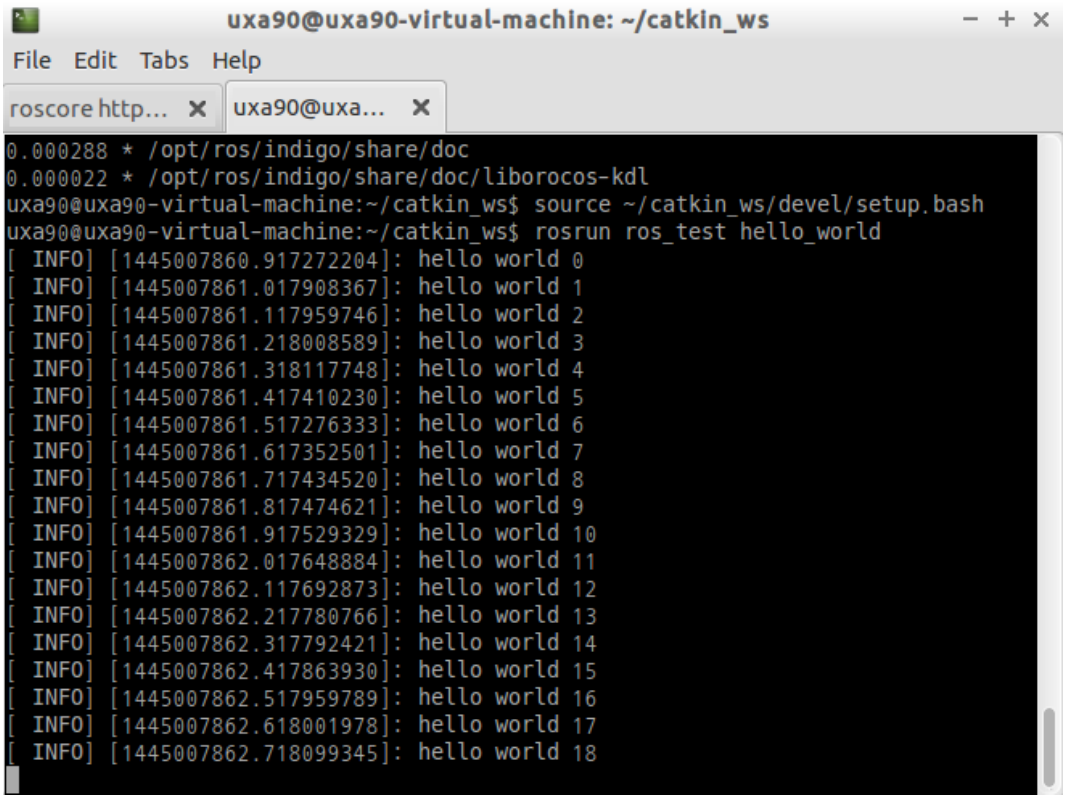
Next use the command 'rosrun' to run the package. Enter 'rosrun (package name) (node name)'.

$ rosrun ros_test hello_world



<Figure 6-26>

When it's running, you will see the result as <Figure 6-27>.



<Figure 6-27>

6. UXA90 communication protocol

This chapter explains the communication protocol used in UXA90. You can control UXA90 using this protocol.

This manual will only mention the protocols for UXA90. For other protocols, refer to the Robobuilder protocol manual.

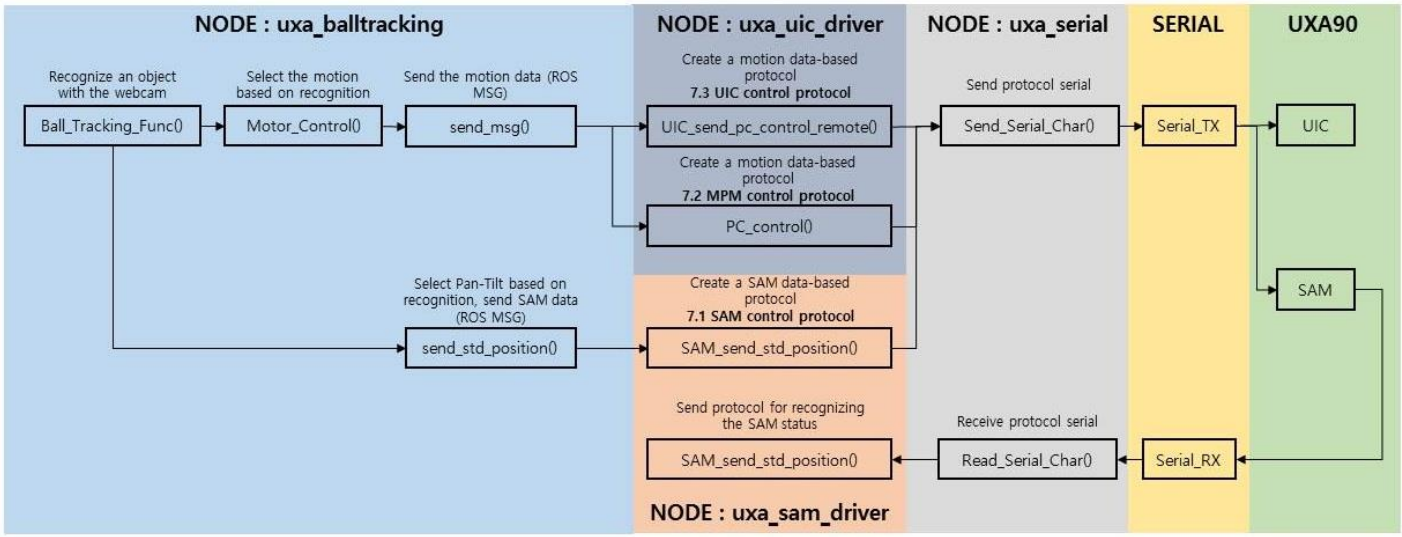
<Figure 7-1> shows how the protocol is created and transferred in each node (function name is in the square). Mastering Chapter 8 beforehand will help understand this chapter better.

<Figure 7-1>, in short, shows the ROS node or device where the action takes place depending on each background color.

'uxa_balltracking' node recognizes an object and decides how to move UXA90. The decision is sent to 'uxa_uic_driver' and 'uxa_sam_driver' as an ROS message.

The data in the message is produced to fit each protocol and sent to 'uxa_serial' as an ROS message. This chapter will explain the protocol rules required to produce protocols.

ROS messages from 'uxa_serial' node is sent to UXA90 device with serial communication. And the data received from UXA90 through serial communication will be sent to 'uxa_sam_driver' node as an ROS message.



<Figure 7-1>

7.1 SAM actuator control protocol

The first protocol we will look at is the protocol used to operate the SAM actuator in the head of UXA90. See <Table 7.1-1>.

Byte1	Byte2		Byte3	Byte4
Header	Speed	ID	Position	CheckSum
0xFF	0 ~ 4	0 ~ 30	1 ~ 254	CheckSum

<Table 7.1-1>

The protocol in <Table 7.1-1> is one of protocols of Robobuilder that controls the location of SAM actuator.

Byte1 is the header and it is set as '0xFF'.

Byte2 represents the speed and ID of SAM actuator. Among 8bit data bit, the top 3 bit shows the speed, and the bottom 5 bit shows the SAM actuator ID.

Byte3 enters the location which the SAM actuator will move to. The range of SAM actuator movement can be divided into 255 steps and controlled.

Byte4 is CheckSum to recognize the communication error. Data is created according to the rules, and if CheckSum data received by SAM actuator does not match the calculated CheckSUm data, the commanded protocol is not applied. CheckSum data is created as follows.

CheckSum = (Byte2 XOR Byte3) AND 0x7F

Once the protocol is created as above and sent to SAM actuator, SAM actuator analyzes the protocol and sends a response protocol if no error has been found. The response protocol is simple – it shows the electric current and position. See <Table 7.1-2>.

Byte1	Byte2
Current	Position
0 ~ 55	0 ~ 255

<Table 7.1-2>

The protocol above controls the position of SAM. But 255 is not enough for the actuator's position resolving power, try using the next protocol. With this protocol, you can use 16,384 steps of resolving power.

Byte1	Byte2		Byte3	Byte4	Byte5	Byte6	Byte7
Header	Mode	-	Mode	ID	Position	Position	CheckSum
0xFF	7	Don't care	200	0 ~ 254	상위 7bit	하위 7bit	CheckSum

<Table 7.1-3>

<Table 7.1-3> is one of Robobuilder's protocols and it can precisely control the position of SAM actuator.

Byte1 is the header and is set as '0xFF'.

Byte2 selects the precise control mode for SAM, so top 3bit should be set as 'true'. The bottom 5bit is not involved, therefore simply enter '0xE0'.

Byte3 is the mode setting packet for the precise control mode for SAM actuator. Set to 200(0xC8).

For Byte4, enter the ID of SAM actuator to control.

For Byte5, enter the position of SAM. Enter for the top 7bit, and do not use the very top bit. See <Table 7.1-4>.

For Byte6, enter the position data of SAM actuator for the bottom 7 bit. Do not use the very top bit. See <Table 7.1-4>.

Byte5		Byte6	
X	target(H7)	**X**	target(L7)
0~16383			

<Table 7.1-4>

Byte7 is CheckSum to recognize communication error. Data is created according to the rules, and if CheckSum data received by SAM actuator does not match the calculated CheckSUm data, the commanded protocol is not applied. CheckSum data is created as follows.

CheckSum = (Byte2 XOR Byte3 XOR Byte4 XOR Byte5 XOR Byte6) AND 0x7F

7.2 MPM control protocol

Next is the protocol that runs a motion on UXA90. See <Table 7.2-1>.

Byte1	Byte2	Byte3	Byte4	Byte5	Byte6
Header	Mode1	Mode2	MPM ID	Motion No.	CheckSum
0xFF	0xE0	0xE1	0 ~ 254	1 ~ 34	CheckSum

<Table 7.2-1>

<Table 7.2-1> is one of Robobuilder's protocols and it controls the MPM. You can use this protocol to run a motion.

Remember that before calling on a motion, you have to call the basic motion (INIT, motion #7) before running another walking motion.

Byte1 is the header and is set as '0xFF'.

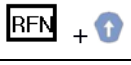
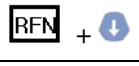
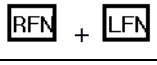
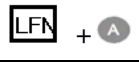
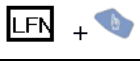
Byte2 is the mode setting to control the MPM. Enter '0xE0'.

Byte3 is the mode setting to run the motion. Enter '0xE1'.

For Byte4, enter the MPM ID.

For Byte5, enter the motion number to run: between 0 to 254. See <Figure 7.2-1> and <Table 7.2-2> for the list of motions.

<Figure 7.2-1>

Button	Mode	Motion No.	Motion List	Remark
		7	INIT	Init motion
		1	BASIC	Basic Posture for walking
			PWR_ON	Power ON & INIT
		2	KICK	Right kick
		3	TURN_LEFT	Turn left
		4	WF_SHORT	Walking forward 1 step
		5	TURN_RIGHT	Turn right
		6	SIDE_LEFT	Step left
		8	SIDE_RIGHT	Step right
		9	WF_4STEP	Walking forward 4 step
		10	BASIC	Basic posture for walking
		11	WF_6STEP	Walking forward 6 step
		12	INTRODUCTION	Introduction
		13	GANGNAM	Gangnam Style dance
		14		
		15		
		16		
		17		
		18		
		19		
		20		
		21	SITDOWN	Sit down
		33	PWR_OFF	System Shutdown (Built-in PC)
		34	MODE_CHANGE	Mode change

<Table 7.2-2>

Byte6 is CheckSum for recognizing communication error. Data is generated based on the rules, and if CheckSum data received by MPM does not match the calculated data, the command protocol does not apply. CheckSum data is generated as follows.

CheckSum = (Byte2 XOR Byte3 XOR Byte4 XOR Byte5) AND 0x7F

7.3 UIC (robot controller) control protocol

Lastly, let's take a look at the protocol to control the UIC board on the ZBOX of UXA90. The basic structure is as <Table 7.3-1>.

Byte1 ~ Byte 8	Byte9	Byte10	Byte11 ~ Byte 14	Byte15 ~	
Header	Command type	platform	Command size	Command content	CheckSum
0xFF, 0xFF, 0xAA, 0x55, 0xAA, 0x55, 0x37, 0xBA	-	-	-	-	CheckSum

<Table 7.3-1>

The protocol in <Table 7.3-1> is one of Robobilder's protocols and it directly controls the robot controller (UIC) from the PC. This protocol has a complex structure and has numerous menus, so on the protocols that this manual uses will be covered. For more information, refer to Robobuilder's protocol manual.

Based on the protocol in <Table 7-3.1>, the protocol in <Table 7.3-2> can be obtained.

Byte1 ~ Byte 8	Byte9	Byte10	Byte11 ~ Byte 14	Byte15	Byte 16
0xFF, 0xFF, 0xAA, 0x55, 0xAA, 0x55, 0x37, 0xBA	0x10	0x00	0x00000001	0x01	CheckSum

<Table 7.3-2>

In short, Byte1 through Byte8 are 8 headers, which are '0xFF, 0xFF, 0xAA, 0x55, 0xAA, 0x55, 0x37, 0xBA'.

Byte9 is the protocol that sets the command type. Depending on this command type, the platform and command content may change. To enter the PC direct control mode, use 0x10(decimal to hexadecimal).

Byte10 is the platform. In the PC direct control mode, enter 0x00.

Byte11 to Byte14 shows the size of command content data. That is, it shows the byte of command from Byte15. Since the command content is 1Byte, enter 0x00000001 and the send from MSB in order.

Byte15 is the actual command. Here, enter 0x01.

The last Byte is CheckSum. CheckSum is exclusive OR (XOR) for all command from Byte 15.

Chapter 8 analyzes the ROS package of UXA90. When generating a protocol to send UXA90, use the protocol analyzed in Chapter 7.

7. UXA90 package analysis

This chapter will talk about the basic package provided with UXA90.

First, take a brief look at the UXA90 package. There are 6 folders in the source folder. <Table 8-1> describes each package.

Package name	Function
uxa_msgs	Define messages to share between nodes
uxa_sam_driver	SAM actuator control package for Pan_Tilt
uxa_serial	Package for serial communication
uxa_uic_driver	UIC motion driver package
uxa_ball_tracking	Ball tracking algorithm package
uxa_dashboard	UXA90 controlling UI package

<Table 8-1>

'uxa_msgs' package defines the messages to share between the nodes.

Each node transfers data with other node through a message. A message can contain a simple data. A series of message can be used as a structure. Between the nodes, a message takes a form of one-way communication. In message communication, there is a publisher and a subscriber, which plays a role as a sender and a receiver respectively.

First, see <Table 8-2> to take a look at the messages used in the UXA90 package.

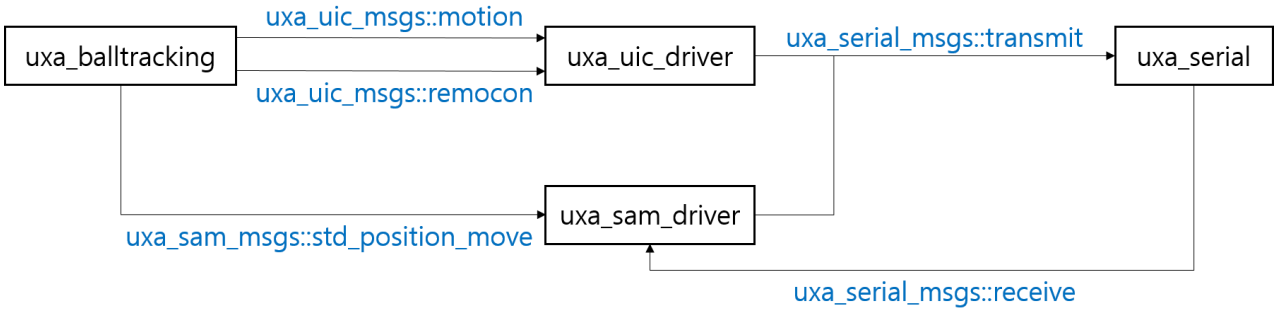
Message name	Function
uxa_uic_msgs::remocon	Run remote controller motion
uxa_uic_msgs::motion	Stop motion
uxa_serial_msgs::receive	Receive data for serial communication
uxa_serial_msgs::transmit	Transmit data for serial communication
uxa_sam_msgs::position_move	SAM operation data
uxa_sam_msgs::std_position_move	SAM operation data

<Table 8-2>

The interrelation of nodes and messages in <Table 8-1> and <Table 8-2> are shown in <Figure 8-1>. In <Figure 8-1>, nodes are square boxes, and the texts in blue are messages between the nodes.

'uxa_msgs' package defines the messages, and 'uxa_dashboard' package controls UXA90 using the UI. These two are not included in <Figure 8-1>.

<Figure 8-1> is a block diagram showing the ROS message sharing between packages (nodes) in the process of operating UXA90. <Table 8-3> is a summary of functions used in the packages (nodes) in <Figure 8-1>.



<Figure 8-1>

Package	Function	Role	Description
uxa_serial	Rev_func()	Receive message	Receive 'uxa_serial_msgs::transmit'
	Send_Serial_Char()	Transmit serial	Transmit serial data
	Send_Serial_String()	Transmit serial	Transmit serial data
	Read_Serial_Char()	Receive serial	Receive serial data
uxa_sam_ driver	SAM_STD_POS_MOVE_FUNC()	Receive message	Receive uxa_sam_msgs::std_position_move
	SAM_POS_MOVE_FUNC()	Receive message	Receive uxa_sam_msgs::position_move
	SAM_send_std_position()	Generate protocol	Chapter7.1 SAM control protocol generation
	SAM_send_position()	Generate protocol	Chapter7.1 SAM control protocol generation
	Message_sender()	Transmit serial	Transmit 'uxa_serial_msgs::transmit'
	SERIAL_SUB_FUNC()	Receive message	Receive 'uxa_serial_msgs::receive'
uxa_uic_ driver	UIC_REMOCON_FUNC()	Receive message	Receive 'uxa_uic_msgs::remocon'
	UIC_MOTION_FUNC()	Receive message	Receive 'uxa_uic_msgs::motion'
	UIC_send_remote()	Generate protocol	Chapter7.3 UIC control protocol generation
	UIC_send_pc_control_remote()	Generate protocol	Chapter7.2 MPM control protocol generation
	PC_control()	Generate protocol	Chapter7.3 UIC control protocol generation
	Message_sender()	Transmit message	Transmit 'uxa_serial_msgs::transmit'
uxa_ball tracking	Ball_Tracking_Func()	Receive image	Receive and recognize image from camera
	Motor_Control()	Judge motion	Judge SAM operation using the recognition information
	send_std_position()	Transmit message	Transmit uxa_sam_msgs::std_position_move
	send_msg()	Transmit message	Transmit 'uxa_uic_msgs::motion'

<Table 8-3>

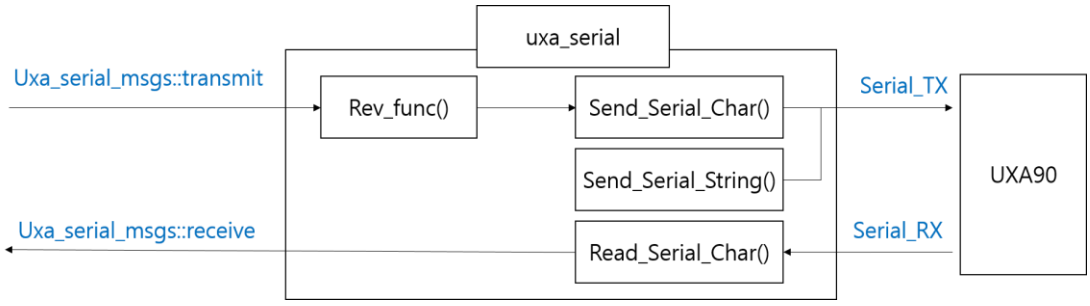
Now, the 4 packages in <Figure 8-1> will be explained in relation to the functions in <Table 8-3>. 'uxa_msgs' package has been explained earlier. 'uxa_dashboard' package is a simple program that sends messages to control UXA90. Further explanation will be left out.

8.1 'uxa_serial' node

First is a brief explanation on 'uxa_serial' node that can directly communicate with the hardware.

'uxa_serial' node sends (Serial_TX) the message (Uxa_serial_msgs::transmit) received from 'uxa_uic_driver' node and 'uxa_sam_driver' node to the hardware through serial communication. And it also sends the data (Serial_RX) from the hardware to 'uxa_sam_driver' node as a message (Uxa_serial_msgs::receive).

<Figure 8.1-1> shows the diagram showing the functions related to 'uxa_serial' node.



<Figure 8.1-1>

Next, take a look at the source code for this node. First, see <Figure 8.1-2>. This is the part that declares the initial setting and the header file.

```
#include "uxa_serial.h"

int main(int argc, char **argv)
{
    sleep(5);

    ros::init(argc, argv, "uxa_serial");
    ros::NodeHandle n;


    ros::Publisher uxa_serial_pub = n.advertise<uxa_serial_msgs::receive>("uxa_serial_publisher", _MSG_BUFF_SIZE);
    ros::Subscriber uxa_serial_sub = n.subscribe<uxa_serial_msgs::transmit>("uxa_serial_subscriber", _MSG_BUFF_SIZE, rev_func);

    ros::Rate loop_rate(1000);
```

<Figure 8.1-2>

#include "uxa_serial.h"

It includes the header file created by the user. It can be found in the folder named 'include' in the same location as the node package.

ros::init(argc, argv, "uxa_serial");

It initializes the node name.

ros::NodeHandle n;

It declares to link with the ROS system.

ros::Publisher uxa_serial_pub = n.advertise<uxa_serial_msgs::receive>
("uxa_serial_publisher", _MSG_BUFF_SIZE);

It declares the publisher to allow transmission. The type of message is 'receive' in 'uxa_serial_msgs' package, and the topic name is 'uxa_serial_publisher'.

Buffer size (_MSG_BUFF_SIZE) is 20. This is defined in the header file.

ros::Subscriber uxa_serial_sub = n.subscribe<uxa_serial_msgs::transmit>
("uxa_serial_subscriber", _MSG_BUFF_SIZE, rev_func);

It declares the subscriber to allow reception. The type of message is 'transmit' in 'uxa_serial_msgs' package, and the topic name is 'uxa_serial_subscriber'.

Buffer size (_MSG_BUFF_SIZE) is 20. This is defined in the header file.

When a message is received, 'rec_func()' callback function goes into action.

ros::Rate loop_rate(1000);

It sets the loop cycle. 1000 refers to 1000hz.

Let's take a look at the callback function that goes into action when a message is received. It's the callback function activated when the 'uxa_serial_sub' message is received.

```
void rev_func(const uxa_serial_msgs::transmit::ConstPtr &msg)
{
    ROS_INFO("recieve msg : 0x%x",msg->tx_data);
    msg_buf[0] = msg->tx_data;
    Send_Serial_Char(Serial, msg_buf);
}
```

<Figure 8.1-3>

It transfers the data through serial communication using 'Send_Serial_Char()' function.

'Send_Serial_Char()' function is composed as shown in <Figure 8.1-4>. It can send through serial communication the buffer 1 data received as a factor value through the 'write()' function. The details will be explained again in the part on serial communication.

```
void Send_Serial_Char(int Serial, unsigned char *Trans_chr)
{
    write(Serial, Trans_chr, 1);
}
```

<Figure 8.1-4>

Next is the part on setting the serial connection for serial communication and declaring buffer. See <Figure 8.1-5>.

```
if((Serial = Init_Serial(_SERIAL_PORT)) != -1)
{
    unsigned char Trans_chr[_SERIAL_BUFF_SIZE];
    unsigned char Recev_chr[_SERIAL_BUFF_SIZE];
    unsigned char cnt = 0;
```

<Figure 8.1-5>

Serial = Init_Serial(_SERIAL_PORT)) != -1

It is the syntax for serial connection. With 'Init_Serial()' function, establish the serial connection and receive the result in return. '_SERIAL_PORT' is '/dev/ttyUSB0', and this is defined in the header file.

Let's take a look at the 'Init_Serial()' function used in <Figure 8.1-5>.

```
if((Serial = open(Serial_Port, O_RDWR | O_NONBLOCK | O_NOCTTY)) == -1)
{
    cout << "SERIAL : " << Serial_Port << " Device open error" << endl;
    cout << "SERIAL : " << Serial_Port << " Device permission change progress...." << endl;
    for(int temp = 0; temp < 5; temp++)
    {
        if(chmod(Serial_Port, __S_IREAD | __S_IWRITE) == 0){

            cout << "SERIAL : " << Serial_Port << " Device permission change complete" << endl;
            Serial = open(Serial_Port, O_RDWR | O_NONBLOCK | O_NOCTTY);

            if(Serial == -1)
            {
                cout << "SERIAL : " << Serial_Port << " Device Not Found" << endl;
                return -1;
            }
            else
                cout << "SERIAL : " << Serial_Port <<" Device open" << endl;
        }
        else
        {
            cout << "SERIAL : " << Serial_Port << " Device permission change error" << endl;
            //return -1;
        }
    }

}

else
    cout << "SERIAL : " << Serial_Port << " Device open" << endl;
```

<Figure 8.1-6>

Serial = open(Serial_Port, O_RDWR | O_NONBLOCK | O_NOCTTY)) == -1

It is the function that opens the actual serial port. This is not created by the user, so the details will not be explained. 'Serial_Port' is the name of serial port mentioned above. The second factor is the setting for the serial connection.

The 'open()' function result will tell either why you have failed to open the device or that you have successfully opened the device.

If you have successfully opened the port, go ahead with the setting for communication. See <Figure 8.1-7>.

```
memset(&Serial_Setting, 0, sizeof(Serial_Setting));
Serial_Setting.c_iflag = 0;
Serial_Setting.c_oflag = 0;
Serial_Setting.c_cflag = _BAUDRATE | CS8 | CREAD | CLOCAL;
Serial_Setting.c_lflag = 0;
Serial_Setting.c_cc[VMIN] = 1;
Serial_Setting.c_cc[VTIME] = 0;

cfsetispeed(&Serial_Setting, _BAUDRATE);
cfsetospeed(&Serial_Setting, _BAUDRATE);

tcflush(Serial, TCIFLUSH);
tcsetattr(Serial, TCSANOW, &Serial_Setting);

return Serial;
```

<Figure 8.1-7>

memset(&Serial_Setting, 0, sizeof(Serial_Setting));

Initialize the communication buffer.

Serial_Setting.c_cflag = _BAUDRATE | CS8 | CREAD | CLOCAL;

_BAUDRATE : Set the communication speed. The header file has set it as 'B921600'.

CS8 : (8 data bit, no parity, 1 stop bit)

CREAD : Enable receiving texts.

CLOCAL : Do not control modem.

Serial_Setting.c_cc[VMIN] = 1;

Serial_Setting.c_cc[VTIME] = 0;

When 'vmin' is greater than 0 and 'vtime' is 0, it will wait indefinitely until vmin number of messages are received.

tcflush(Serial, TCIFLUSH);

tcsetattr(Serial, TCSANOW, &Serial_Setting);

Empty the communication buffer.

After the setting, restore the 'Serial' variable to send the serial communication information as function call value. This will be used as an ID for serial communication when sharing data.

Now you have made the connection and established the settings for serial communication. Now, send the protocol for initialization of UXA90.

```
Trans_chr[cnt++] = 0xFF;
Trans_chr[cnt++] = 0xFF;
Trans_chr[cnt++] = 0xAA;
Trans_chr[cnt++] = 0x55;
Trans_chr[cnt++] = 0xAA;
Trans_chr[cnt++] = 0x55;
Trans_chr[cnt++] = 0x37;
Trans_chr[cnt++] = 0xBA;
Trans_chr[cnt++] = 0x10;
Trans_chr[cnt++] = 0x00;
Trans_chr[cnt++] = 0x00;
Trans_chr[cnt++] = 0x00;
Trans_chr[cnt++] = 0x00;
Trans_chr[cnt++] = 0x01;
Trans_chr[cnt++] = 0x01;
Trans_chr[cnt++] = 0x01;

Send_Serial_String(Serial, Trans_chr, cnt);
sleep(1);

cnt = 0;

Trans_chr[cnt++] = 0xFF;
Trans_chr[cnt++] = (unsigned char)(7 << 5);
Trans_chr[cnt++] = 225;
Trans_chr[cnt++] = 0;
Trans_chr[cnt++] = 0X07;
Trans_chr[cnt++] = (Trans_chr[1]^Trans_chr[2]^Trans_chr[3]^Trans_chr[4]) & 0x7F;

Send_Serial_String(Serial, Trans_chr, cnt);
sleep(1);

memset(Trans_chr, '\0', sizeof(Trans_chr));
memset(Recev_chr, '\0', sizeof(Recev_chr));
```

<Figure 8.1-8>

After the serial connection has been established, create the protocol in 'Trans_chr' buffer, and use 'Send_Serial_String()' function to send the data saved in the buffer.

The protocol created on the top of <Figure 8.1-8>is **Chapter 7.3 UIC control protocol**. The command type of Byte9 is 16(0x10), so this protocol allows entering the direct control mode where the UIC will be directly controlled from the PC.

After sending the protocol above, generate a 6Byte protocol. This will be **Chapter 7.2 MPM control protocol** (controlling UXA90's motions). The motion number is 7, so open INIT motion to initialize before UXA90 walks or takes another action.

'Send_Serial_String()' function is described in <Figure 8.1-9>.

```
void Send_Serial_String(int Serial, unsigned char *Trans_chr, int Size)
{
    write(Serial, Trans_chr, Size);
}
```

<Figure 8.1-9>

You can send data using serial communication if you enter the serial port connection information, buffer, and buffer size in 'write' function.

Once the initial setting is done, you will enter 'while()'. See <Figure 8.1-10>.

```
uxa_serial_msgs::receive msg;

while(ros::ok())
{
    loop_rate.sleep();
    if(Read_Serial_Char(Serial, Recev_chr) == 1)
    {
        msg.rx_data = Recev_chr[0];
        uxa_serial_pub.publish(msg);
    }
    ros::spinOnce();
}
```

<Figure 8.1-10>

uxa_serial_msgs::receive msg;

It declares the message variables in the form of 'receive' message file before entering 'while()'.

loop_rate.sleep();

It will go into sleep following the loop cycle assigned above.

Read_Serial_Char(Serial, Recev_chr) == 1

It is the function that receives texts through serial communication. Details will be explained later.

msg.rx_data = Recev_chr[0];

Received text is assigned as a message.

uxa_serial_pub.publish(msg);

It publishes a message.

ros::spinOnce();

It is the function to call the callback function. It runs the callback function when a message is received.

The program is repeated as above. Lastly, let's take a look at the function that receives data through serial communication. This function saves received data considering the variables with serial communication information, reception buffer, and the number of data received. See <Figure 8.1-11>.

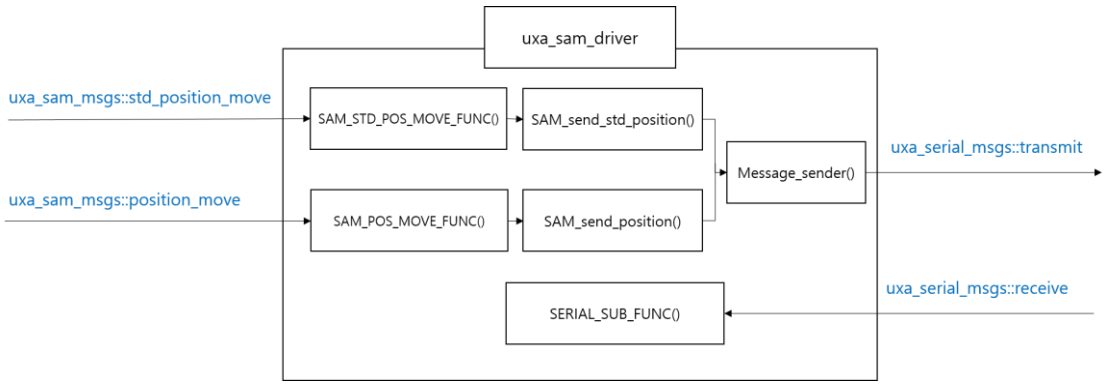
```
int Read_Serial_Char(int Serial, unsigned char *Recei_chr)
{
    if(read(Serial, Recei_chr, 1) > 0)
        return 1;

    return -1;
}
```

<Figure 8.1-11>

8.2 'uxa_sam_driver' node

Next is on 'uxa_sam_driver' node that directly controls the SAM, which is located in the head part of UXA90. 'uxa_sam_driver' node receives SAM actuator-related messages (uxa_sam_msgs::std_position_move, uxa_sam_msgs::position_move) from 'uxa_balltracking' node and 'uxa_dashboard' node, creates a protocol that can control the SAM actuator with 'uxa_serial' node, and then sends the message (uxa_serial_msgs::transmit). Also, it receives the data from 'uxa_serial' node in as a message (uxa_serial_msgs::receive).

<Figure 8.2-1> is a diagram showing the functions in 'uxa_sam_driver' node.



<Figure 8.2-1>

Now, let's analyze the source code to see what are the roles of functions and the codes.

This node has a simple program. First, let's look at 'main()' function in 'uxa_sam_driver.cpp' file. From this node, we will skip explanation on ROS initialization and basic ROS operation.

```
#include "uxa_sam_driver.h"

int main(int argc, char **argv)
{

    ros::init(argc, argv, "uxa_sam_driver");
    Init_Message();

    ros::Rate loop_rate(1000);

    cout << "SAM_DRIVER : " <<  "SAM DRIVER stand by.." << endl << endl;

    while(ros::ok())
    {
        loop_rate.sleep();
        ros::spinOnce();
    }


    cout << endl;
    cout << "SAM_DRIVER : " << "uxa_sam_driver node terminate." << endl;
    return 0;
}
```

<Figure 8.2-2>

The overall node operation initializes ROS and the messages. And after that, it checks for callback function in 1000hz cycle. 'Init_Message()' function has the part on message initialization. It is in 'sam_packet.cpp' file.

See 'sam_packet.cpp' file.

```
void Init_Message()
{
    ros::NodeHandle n;

    uxa_serial_pub = n.advertise<uxa_serial_msgs::transmit>
            ("uxa_serial_subscriber", _MSG_BUFF_SIZE);

    uxa_serial_sub = n.subscribe<uxa_serial_msgs::receive>
            ("uxa_serial_publisher", _MSG_BUFF_SIZE, SERIAL_SUB_FUNC);

    sam_driver_position_move_sub = n.subscribe<uxa_sam_msgs::position_move>
            ("sam_driver_position_move", _MSG_BUFF_SIZE, SAM_POS_MOVE_FUNC);

    sam_driver_std_position_move_sub = n.subscribe<uxa_sam_msgs::std_position_move>
            ("sam_driver_std_position_move", _MSG_BUFF_SIZE, SAM_STD_POS_MOVE_FUNC);

}
```

<Figure 8.2-3>

This is the 'Init_Message()' function used in 'main()' function in 'sam_packet.cpp' file. 'Init_Message()' has the ORS message initialization process. Let's take a look at this part first.

uxa_serial_pub = n.advertise<uxa_serial_msgs::transmit>
("uxa_serial_subscriber", _MSG_BUFF_SIZE);

It declares the publisher to allow transmission. The type of message is 'transit' in 'uxa_serial_msgs' package, and the topic name is 'uxa_serial_subscriber'.

Buffer size (_MSG_BUFF_SIZE) is 20. It is defined in the header file.

uxa_serial_sub = n.subscribe<uxa_serial_msgs::receive>
("uxa_serial_publisher", _MSG_BUFF_SIZE, SERIAL_SUB_FUNC);

It declares the subscriber to allow reception. The type of message is 'receive' in 'uxa_serial_msgs' package, and the topic name is 'uxa_serial_publisher'.

Buffer size (_MSG_BUFF_SIZE) is 20. It is defined in the header file.

When a message is received, 'SERIAL_SUB_FUNC()' callback function goes into action.

sam_driver_position_move_sub = n.subscribe<uxa_sam_msgs::position_move>
("sam_driver_position_move", _MSG_BUFF_SIZE, SAM_POS_MOVE_FUNC);

It declares the subscriber to allow reception. The type of message is 'position move' in 'uxa_sam_msgs' package, and the topic name is 'sam_driver_position_move'.

Buffer size (_MSG_BUFF_SIZE) is 20. It is defined in the header file.

When a message is received, 'SAM_POS_MOVE_FUNC()' callback function goes into action.

sam_driver_std_position_move_sub = n.subscribe
<uxa_sam_msgs::std_position_move> ("sam_driver_std_position_move",
_MSG_BUFF_SIZE, SAM_STD_POS_MOVE_FUNC);

It declares the subscriber to allow reception. The type of message is 'std_position_move' in 'uxa_sam_msgs' package, and the topic name is 'sam_driver_std_position_move'.

Buffer size (_MSG_BUFF_SIZE) is 20. It is defined in the header file.

When a message is received, 'SAM_STD_POS_MOVE_FUNC ()' callback function goes into action.

This node declares a total of 3 callback functions. Let's take a look at each function in order.

First is 'SERIAL_SUB_FUNC()' callback function.

```
void SERIAL_SUB_FUNC(const uxa_serial_msgs::receive::ConstPtr &msg)
{
    ROS_INFO("recieve msg : 0x%x",msg->rx_data);
}
```

<Figure 8.2-4>

'SERIAL_SUB_FUNC()' callback function translates data received from ROS message for debugging. This callback function receives messages (uxa_serial_msgs::receive) from 'uxa_serial' node as shown in <Figure 8-1>.

Next is 'SAM_POS_MOVE_FUNC()' callback function.

```
void SAM_POS_MOVE_FUNC(const uxa_sam_msgs::position_move::ConstPtr &msg)
{
    SAM_send_position(msg->id, msg->torqlevel, msg->pos);
}
```

<Figure 8.2-5>

'SAM_POS_MOVE_FUNC()' callback function includes the function that creates the protocol to send to SAM. The factors include ID, torque level, and position. This callback function is not included in <Figure 8-1>, but it receives messages (uxa_sam_msgs::position_move) from 'uxa_dashboard' node.

Next is 'SAM_STD_POS_MOVE_FUNC()' callback function.

```
void SAM_STD_POS_MOVE_FUNC(const uxa_sam_msgs::std_position_move::ConstPtr &msg)
{
    SAM_send_std_position(msg->id, msg->pos14);
}
```

<Figure 8.2-6>

'SAM_STD_POS_MOVE_FUNC()' callback function includes the function that creates a protocol to send to SAM. It is different from 'SAM_POS_MOVE_FUNC()' as this does not include the torque level as the factor. As shown in <Figure 8-1>, this callback function receives messages (uxa_sam_msgs::std_position_move) from 'uxa_balltracking' node.

Now we have reviewed all three callback functions. It's time to take a look at 'SAM_send_position()' and 'SAM_send_std_position()' functions included in the callback functions.

These two functions use the data received to create a protocol to suit the SAM and then sends the buffer and the number of buffers with 'Message_sender()' function.

In <Figure 8.2-7>, the protocol created with 'SAM_send_position()' function is **Chapter 7.1 SAM actuator control protocol**. This protocol can control the position of SAM actuator. It creates a protocol of 4Byte.

Also, the protocol created with 'SAM_send_std_position()' function is **Chapter 7.1 SAM actuator control protocol**. It can precisely control the position of SAM actuator. It creates a protocol of 7Byte.

```
void SAM_send_position(unsigned char id, unsigned char torq, unsigned char pos)
{
    unsigned char cnt = 0;

    Send_buf[cnt++] = 0xFF;
    Send_buf[cnt++] = id | (torq << 5);
    Send_buf[cnt++] = pos;
    Send_buf[cnt++] = (Send_buf[1]^Send_buf[2]) & 0x7F;

    Message_sender(Send_buf, cnt);
}

void SAM_send_std_position(unsigned char id, unsigned int pos14)
{
    unsigned char cnt = 0;

    Send_buf[cnt++] = 0xFF;
    Send_buf[cnt++] = (unsigned char)(7 << 5);
    Send_buf[cnt++] = 200;
    Send_buf[cnt++] = id;
    Send_buf[cnt++] = pos14 >> 7;
    Send_buf[cnt++] = (unsigned char)(pos14 & 0x7F);
    Send_buf[cnt++] = (Send_buf[1]^Send_buf[2]^Send_buf[3]^Send_buf[4]^Send_buf[5]) & 0x7F;

    Message_sender(Send_buf, cnt);
}
```

<Figure 8.2-7>

Lastly, let's take a look at 'Message_sender()' function. This function makes a protocol out of the function in <Figure 8.2-7> and sends it as factors, which will be sent as a ROS message. The data in the buffer is saved in the ROS message, and this message is transmitted. This action is repeated for the number of buffers to transmit the data in the buffers.

```
void Message_sender(unsigned char *Send_data, int Size)
{
    for(char cnt = 0; cnt < Size; cnt++)
    {
        serial_pub_msg.tx_data = Send_data[cnt];
        uxa_serial_pub.publish(serial_pub_msg);
    }
}
```

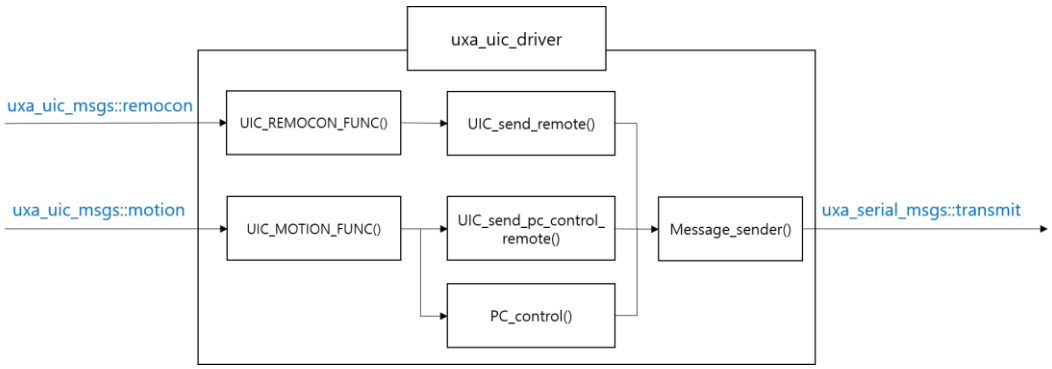
<Figure 8.2-8>

This will be the end of the explanation on 'uxa_sam_driver' node.

8.3 'uxa_uic_driver' node

Next is on 'uxa_uic_driver' node. 'uxa_uic_driver' node receives UIC-related messages (uxa_uic_msgs::motion, uxa_uic_msgs::remocon) from 'uxa_balltracking' node and 'uxa_dashboard' node and creates a protocol to control the UIC with 'uxa_serial' node and delivers the message (uxa_serial_msgs::transmit).

<Figure 8.3-1> is a diagram that shows the functions in 'uxa_uic_driver' node.



<Figure 8.3-1>

Let's analyze the source code to find out what the functions and the codes do.

Programming of this node is very simple. First, let's take a look at 'main()' function in the 'uxa_uic_driver.cpp' file. We will skip The part on ROS initialization and basic ROS operation.

```
#include "uxa_uic_driver.h"

int main(int argc, char **argv)
{

    ros::init(argc, argv, "uxa_uic_driver");
    Init_Message();

    ros::Rate loop_rate(1000);

    cout << "UIC_DRIVER : " <<  "UIC DRIVER stand by." << endl << endl;

    while(ros::ok())
    {
        loop_rate.sleep();
        ros::spinOnce();
    }

    cout << endl;
    cout << "UIC_DRIVER : " << "uxa_uic_driver node terminate." << endl;
    return 0;
}
```

<Figure 8.3-2>

The node initializes the messages after initializing ROS. And it checks the callback function every 1000hz cycle. 'Init_Message()' function has the message initialization part in the 'uic_packet.cpp' file.

Take a look at 'uic_packet.cpp' file.

```
void Init_Message()
{
    ros::NodeHandle n;

    uxa_serial_pub = n.advertise<uxa_serial_msgs::transmit>
            ("uxa_serial_subscriber", _MSG_BUFF_SIZE);

    uic_driver_remocon_sub = n.subscribe<uxa_uic_msgs::remocon>
            ("uic_driver_remocon", _MSG_BUFF_SIZE, UIC_REMOCON_FUNC);

    uic_driver_motion_sub = n.subscribe<uxa_uic_msgs::motion>
            ("uic_driver_motion", _MSG_BUFF_SIZE, UIC_MOTION_FUNC);

}
```

<Figure 8.3-3>

It is the 'Init_Message()' function used for 'main()' function in 'uic_packet.cpp' file. 'Init_Message()' has the part on initialization of ROS message. Let's review this part first.

uxa_serial_pub = n.advertise<uxa_serial_msgs::transmit>("uxa_serial_subscriber", _MSG_BUFF_SIZE);

It declares the publisher to allow transmission. The type of message is 'transmit' in 'uxa_serial_msgs' package, and the topic name is 'uxa_serial_publisher'.

Buffer size (_MSG_BUFF_SIZE) is 20. This is defined in the header file.

uic_driver_remocon_sub = n.subscribe<uxa_uic_msgs::remocon> ("uic_driver_remocon", _MSG_BUFF_SIZE, UIC_REMOCON_FUNC);

It declares the subscriber to allow reception. The type of message is 'remocon' in 'uxa_uic_msgs' package, and the topic name is 'uxa_driver_remocon'.

Buffer size (_MSG_BUFF_SIZE) is 20. This is defined in the header file.

When a message is received, 'UIC_REMOCON_FUNC()' callback function goes into action.

uic_driver_motion_sub = n.subscribe<uxa_uic_msgs::motion>("uic_driver_motion", _MSG_BUFF_SIZE, UIC_MOTION_FUNC);

It declares the subscriber to allow reception. The type of message is 'motion' in 'uxa_uic_msgs' package, and the topic name is 'uxa_driver_motion'.

Buffer size (_MSG_BUFF_SIZE) is 20. This is defined in the header file.

When a message is received, 'UIC_MOTION_FUNC()' callback function goes into action.

This node declares two callback functions. Let's study them in order.

First is 'UIC_REMOCON_FUNC()' callback function.

```
void UIC_REMOCON_FUNC(const uxa_uic_msgs::remocon::ConstPtr &msg)
{
    ROS_INFO("recieve msg : %d",msg->btn_code);
    UIC_send_remote(msg->btn_code);
}
```

<Figure 8.3-4>

'UIC_REMOCON_FUNC()' callback function translates data received from ROS message for debugging and sends to UIC using 'UIC_send_remote()' function. This function will be explained later. This callback function receives messages from 'uxa_balltracking' node as shown in <Figure 8-1>.

Next is 'UIC_MOTION_FUNC()' callback function.

```
void UIC_MOTION_FUNC(const uxa_uic_msgs::motion::ConstPtr &msg)
{

    if(msg->motion_name.compare("basic_motion")==0)
        UIC_send_pc_control_remote(BTN_A);

    else if(msg->motion_name.compare("kick_right")==0)
        UIC_send_pc_control_remote(BTN_B);

    else if(msg->motion_name.compare("turn_left")==0)
        UIC_send_pc_control_remote(BTN_LR);

    else if(msg->motion_name.compare("walk_forward_short")==0)
        UIC_send_pc_control_remote(BTN_U);

    else if(msg->motion_name.compare("turn_right")==0)
        UIC_send_pc_control_remote(BTN_RR);

    else if(msg->motion_name.compare("walk_left")==0)
        UIC_send_pc_control_remote(BTN_L);

    else if(msg->motion_name.compare("walk_right")==0)
        UIC_send_pc_control_remote(BTN_R);

    else if(msg->motion_name.compare("walk_foward_4step")==0)
        UIC_send_pc_control_remote(BTN_LA);

    else if(msg->motion_name.compare("walk_back")==0)
        UIC_send_pc_control_remote(BTN_D);

    else if(msg->motion_name.compare("walk_foward_6step")==0)
        UIC_send_pc_control_remote(BTN_RA);

    else if(msg->motion_name.compare("demo_introduction")==0)
        UIC_send_pc_control_remote(BTN_1);

   else if(msg->motion_name.compare("dance_gangnamstyle")==0)
       UIC_send_pc_control_remote(BTN_2);
```

```
    else if(msg->motion_name.compare("demo")==0)
        UIC_send_pc_control_remote(BTN_3);

    else if(msg->motion_name.compare("stop")==0)
        UIC_send_pc_control_remote(BTN_C);

    else if(msg->motion_name.compare("pc_control")==0)
        PC_control();

}
```

<Figure 8.3-5>

'UIC_MOTION_FUNC()' callback function has a function that creates a protocol to send to UIC. The factors change depending on the motion name received in the ROS message. This callback function receives the message from 'uxa_balltracking' node as shown in <Figure 8-1>.

Now, we have looked at two callback functions. Time to study 'UIC_send_remote()', 'UIC_send_pc_control_remote()', and 'PC_control()' functions.

These three functions use the data received as factors to create a protocol for UIC and the sends the buffer and the number of buffers with 'Message_sender()' function. Explanation on 'Message_sender()' function will be skipped as it is same as 'uxa_sam_driver' node.

See <figure 8.3-6> for the codes for these three functions.

First is 'UIC_send_remote()' function. The protocol this function creates is **Chapter 7.3 UIC control protocol**. Byte9(buffer #8) is 0x14, and it is the protocol motion running function taken as an example in Chapter 7.3. Refer to the example in Chapter 7.3.

'UIC_send_pc_control_remote()' function creates **Chapter 7.2 MPM control protocol** to send the motion number. MPM ID is set as 0 to create a protocol. The header and the ID of the protocol are initialized when the buffer is generated. The protocol is created only with the motion number and CheckSum.

Lastly, 'PC_control()' function also uses **Chapter 7.3 UIC control protocol**. Byte9's command type is 16(0x10), so it is a protocol that commands to enter the direct control mode where the UIC can be directly controlled on PC.

See Chapter 7 for more information on protocol.

```
void UIC_send_remote(unsigned char remote)
{
    Send_buf[8] = 0x14;
    for(char cnt = 9; cnt < 13; cnt++)
        Send_buf[cnt] = 0x00;
    Send_buf[13] = 0x01;
    Send_buf[14] = remocte;
    Send_buf[15] = remote;

    Message_sender(Send_buf, 16);
}

void UIC_send_pc_control_remote(unsigned char remote)
{

    Send_pc_buf[4] = remote;
    Send_pc_buf[5] = (Send_pc_buf[1]^Send_pc_buf[2]^Send_pc_buf[3]^remote) & 0x7F;

    Message_sender(Send_pc_buf, 6);
}


void PC_control()
{
    unsigned char cnt = 0;
    unsigned char buf[17];

    buf[cnt++] = 0xFF;
    buf[cnt++] = 0xFF;
    buf[cnt++] = 0xAA;
    buf[cnt++] = 0x55;
    buf[cnt++] = 0xAA;
    buf[cnt++] = 0x55;
    buf[cnt++] = 0x37;
    buf[cnt++] = 0xBA;
    buf[cnt++] = 0x10;
    buf[cnt++] = 0x00;
    buf[cnt++] = 0x00;
    buf[cnt++] = 0x00;
    buf[cnt++] = 0x00;
    buf[cnt++] = 0x01;
    buf[cnt++] = 0x01;
    buf[cnt++] = 0x01;

    Message_sender(buf, cnt);
}
```

<Figure 8.3-6>

This is the end of the explanation on 'uxa_uic_driver' node.

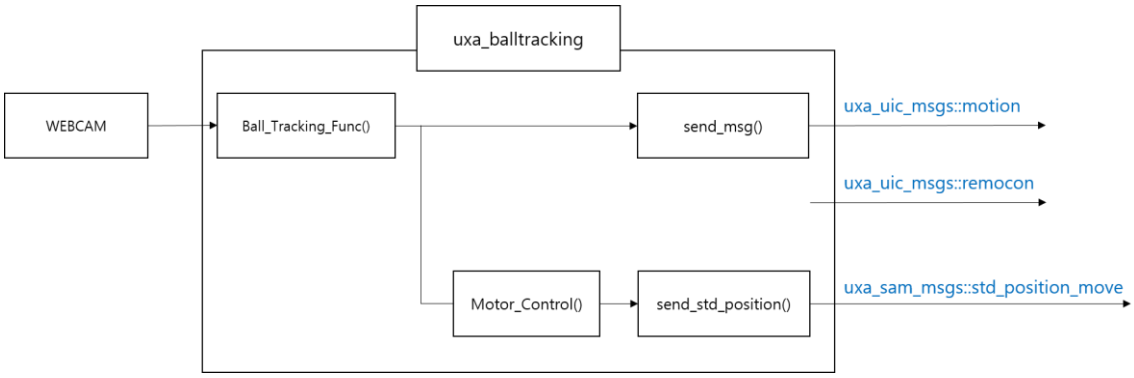
8.4 'uxa_balltracking' node

Last is 'uxa_balltracking' node. This node uses the webcam (OpenCV library) to recognizes the ball and its position, and uses 'uxa_uic_driver' node and 'uxa_sam_driver' node to generate and send the message that decides the motion of UXA90.

With 'uxa_uic_driver', it sends the 'uxa_uic_msgs::motion' message, and with 'uxa_sam_driver', it sends the 'uxa_sam_msgs::std_position_move' message.

'uxa_uic_msgs::remocon' message is declared but is not used.

<Figure 8.4-1> is the diagram that includes the functions in the 'uxa_balltracking' node.



<Figure 8.4-1>

First, let's take a look at the initialization process of the node. Follow <Figure 8.4-2>. We will skip the part on ROS initialization and basic ROS operation.

Here is the initialization process for ROS message. Let's take a look.

```
ros::init(argc, argv, "test_opencv");

ros::NodeHandle nh;
ros::Rate loop_rate(30);

motion_pub = nh.advertise<uxa_uic_msgs::motion>
        ("uic_driver_motion", 100);
remocon_pub = nh.advertise<uxa_uic_msgs::remocon>
        ("uic_driver_remocon", 100);

std_pos_move_pub = nh.advertise<uxa_sam_msgs::std_position_move>
        ("sam_driver_std_position_move", 100);
```

<Figure 8.4-2>

motion_pub = nh.advertise<uxa_uic_msgs::motion>("uic_driver_motion", 100);

It declares the publisher to allow transmission. The type of message is 'motion' in 'uxa_uic_msgs' package, and the topic name is 'uxa_driver_motion'.

remocon_pub = nh.advertise<uxa_uic_msgs::remocon>
("uic_driver_remocon", 100);

It declares the publisher to allow transmission. The type of message is 'remocon' in 'uxa_uic_msgs' package, and the topic name is 'uxa_driver_remocon'.

std_pos_move_pub = nh.advertise<uxa_sam_msgs::std_position_move>("sam_driver_std_position_move", 100);

It declares the publisher to allow transmission. The type of message is 'std_position_move' in 'uxa_sam_msgs' package, and the topic name is 'sam_driver_std_position_move'.

Initialize ROS message and then initialize UXA90.

As shown in <Figure 8.4-3>, use 'send_msg()' function to send messages to 'uxa_uic_driver' node. And then use 'send_std_position()' function to send the message on UXA90 head motor to 'uxa_sam_driver' node.

Lastly, use 'Init_Ball_Tracking()' function to initialize image treatment and ball tracking algorithm.

```
sleep(2);
send_msg("pc_control");
sleep(1);
send_msg("stop");
sleep(7);
send_msg("basic_motion");
send_std_position(Head_Yaw.ID, (unsigned int)Head_Yaw.Pos);
send_std_position(Head_Pitch.ID, (unsigned int)Head_Pitch.Pos);
sleep(2);

if(Init_Ball_Tracking(__WINDOW_MODE) == __WEBCAM_ERROR) return __WEBCAM_ERROR;
```

<Figure 8.4-3>

Here's the explanation on three functions in <Figure 8.4-3>. First, let's take a look at 'send_msg()' function and 'send_std_position()' function. These two functions send the factor value received as data as a ROS message.

'send_msg()' function sends the string message in factor value to 'uxa_uic_driver' node.

Also, 'send_std_position()' function transmits the SAM actuator information in factor value to 'uxa_sam_driver' node. The SAM actuator on Pitch axis gets an ID of 24, and the SAM actuator on Yaw axis gets and ID of 23.

```
void send_msg(std::string STR)
{
    uic_motion_msg.motion_name = STR;
    motion_pub.publish(uic_motion_msg);
}

void send_std_position(unsigned char ID ,unsigned int POS)
{
    sam_std_pos_move_msg.id = ID;
    sam_std_pos_move_msg.pos14 = POS;
    std_pos_move_pub.publish(sam_std_pos_move_msg);
}
```

<Figure 8.4-4>

Next is on 'Init_Ball_Tracking()' function. This function uses OpenCV library to initialize the image treatment part. So make sure to understand the process step by step.

First, <Figure 8.4-5> shows the process of receiving the frame from the camera connected to the USB.

capture = cvCaptureFromCAM(0);

Capture a frame (image) from the camera and save in the 'capture' variable. The factor value here is the camera number of the camera connected to the USB. If there is no data saved in 'capture' variable, an error message appears and the function closes.

Next, use 'cvSetCaptureProperty()' function to set the size of the window that will display the screen. In the screen display mode, three windows are generated.

```
// Open capture device. 0 is /dev/video0, 1 is /dev/video1, etc.
capture = cvCaptureFromCAM( 0 );
if( !capture )
{
        fprintf( stderr, "ERROR: capture is NULL \n" );
        getchar();
        return __WEBCAM_ERROR;
}

cvSetCaptureProperty(capture, CV_CAP_PROP_FRAME_WIDTH, __X_RESOLUTION);
cvSetCaptureProperty(capture, CV_CAP_PROP_FRAME_HEIGHT, __Y_RESOLUTION);
// Create a window in which the captured images will be presented

if(Window_Open == __WINDOW_OPEN)
{
    cvNamedWindow( "Camera", CV_WINDOW_AUTOSIZE );
    cvNamedWindow( "HSV", CV_WINDOW_AUTOSIZE );
    cvNamedWindow( "After Color Filtering", CV_WINDOW_AUTOSIZE );
    printf("Window open mode\n");
}
else
    printf("Window close mode\n");
```

<Figure 8.4-5>

And then set the range for each HSV. HSV is the image that converted the RGB image from the camera into H(Hue), S(Saturation), and V(Value). Use this range to show only the data needed in the HSV image.

After establishing the standard for the HSV image, set the ball information and the UXA90 head motor. Details will be excluded as this setting can be found in the code.

```
Max.H = 32; Max.S = 95;  Max.V = 95;
Min.H = 6;  Min.S = 60;  Min.V = 60;

Max.H /= 2; Min.H /= 2;
Max.S *= 2.55;  Min.S *= 2.55;
Max.V *= 2.55;  Min.V *= 2.55;

Ball.X_Axis = __X_HALF_RESOLUTION;
Ball.Y_Axis = __Y_HALF_RESOLUTION;
Ball.Ball_size = 0;
Ball.Renewal = false;
Ball.X_Search_Completed = false;
Ball.Y_Search_Completed = false;

Head_Pitch.ID = 24;
Head_Pitch.Pos = 2350;
Head_Pitch.Pos_Err = 0;
Head_Pitch.Angle = 0;

Head_Yaw.ID = 23;
Head_Yaw.Pos = 2048;
Head_Yaw.Pos_Err = 0;
Head_Yaw.Angle = 0;
```

<Figure 8.4-6>

Lastly use 'cvScalar()' function to convert each H, S, and V variable (double type) into a HVS variable (CvScalar type) in OpenCV. This is to use the functions provided in OpenCV library later.

Use 'cvCreateImage()' function to create an empty image data space. Think of it as assigning a space to save data. This function includes the size, image information, and the number of channels as its factor values. Size is the size of the image, and the image information shows the bit. 'IPL_DEPTH_8U' refers to unsigned 8bit data size. The last factor value represents the number of channels. Image data such as RGB and HSV are called 3-channel image, and image data with H, S, V, R, G, or B are called 1-channel image.

Before treating the image, it is necessary to assign space for data. This has to perfectly match the information of the image to use, therefore write the codes carefully.

```
hsv_min = cvScalar(Min.H, Min.S, Min.V, 0);
hsv_max = cvScalar(Max.H, Max.S, Max.V, 0);

hsv_frame = cvCreateImage(size, IPL_DEPTH_8U, 3);
thresholded = cvCreateImage(size, IPL_DEPTH_8U, 1);
```

<Figure 8.4-7>

We've looked at 'Init_Ball_Tracking()' function thus far. Now let's take a look at the internal algorithm. As shown in <Figure 8.4-8>, inside the loop is the algorithm.

Inside the loop phrase are two parts: 'Ball_Tracking_Func()' function and 'Motor_Control()' function. 'Ball_Tracking_Func()' function uses OpenCV library to recognize the ball. 'Motor_Control()' function uses the information recognizing the ball to control the head of UXA90 (SAM actuator).

```
while(ros::ok())
{
    loop_rate.sleep();
    Ball.Renewal = false;

    if(Ball_Tracking_Func(__WINDOW_MODE) == __BALL_TRACKING_END)   break;

    if(Motor_Control())
    {
```

<Figure 8.4-8>

First, let's take a look at 'Ball_Tracking_Func()' function. Refer to <Figure 8.4-9> for pre-treatment of the image before running the algorithm for recognizing the ball.

Use the 'cvQuertFrame()' function to get a frame from the connected camera. If there is no frame, an error message appears and the function closes.

'cvCvtColor()' function converts the frame (RGB image) into an image to use (HSV image). Therefore, enter 'CV_BGR2HSV' as the last factor.

Next use 'cvInRangeS()' function to extract the data in the standard range of HSV assigned in the initialization process from the converted HSV camera image. Therefore, it moves data between 'hsv_min' and 'hsv_max' variables from the initialization function of 'hsv_frame' image to 'thresholded' image.

'cvCreateMemStorage()' function is a memory storage required to convert the hough in the very bottom of <Figure 8.4-9>. It's the pointer of the space where the converted hough will be saved.

Use 'cvSmooth()' function to apply the Gaussian filter to the 'threshold' image. This makes the image look softer and reduces noise.

Using 'cvHoughCircles()' function, find the circle in the filtered image. Enter 'threshold' (8bit image) and find circle shapes. The 5th and 6th factor values decide the critical value for the canny edge detector detecting the edge from the image. The 7th and 8th factor value assigns the minimum and maximum radius of the circle to be detected.

```
frame = cvQueryFrame( capture );        // Get one frame
if( !frame )
{
        fprintf( stderr, "ERROR: frame is null...\n" );
        getchar();
        return __BALL_TRACKING_ERROR;
}

cvCvtColor(frame, hsv_frame, CV_BGR2HSV);
cvInRangeS(hsv_frame, hsv_min, hsv_max, thresholded);
CvMemStorage* storage = cvCreateMemStorage(0);

cvSmooth(thresholded, thresholded, CV_GAUSSIAN, 17, 17);

CvSeq* circles = cvHoughCircles(thresholded, storage, CV_HOUGH_GRADIENT, 2,
                                thresholded->height/2, 100, 45, 1, 80);
```

<Figure 8.4-9>

After the pre-treatment for ball tracking, use the 'for' loop phrase to save the data as many as the number of circles detected and the display on the output image. You can mark the circles on the output image using the 'cvCircle()' function.

Use 'cvReleaseMemStorage()' to cancel the memory. Without this process, the memory may run out as there will be no new memory space.

Lastly, use 'cvWaitKey()' function and return '__BALL_TRACKING_END' if the ESC key is pressed and '__BALL_TRACKING_OK' otherwise.

```
for (int i = 0; i < circles->total; i++)
{
    float* p = (float*)cvGetSeqElem( circles, i );

    Ball.X_Axis = p[0];
    Ball.Y_Axis = p[1];
    Ball.Ball_size = p[2];
    Ball.Renewal = true;

    cvCircle( frame, cvPoint(cvRound(p[0]),cvRound(p[1])),
                        3, CV_RGB(0,255,0), -1, 8, 0 );
    cvCircle( frame, cvPoint(cvRound(p[0]),cvRound(p[1])),
                        cvRound(p[2]), CV_RGB(255,0,0), 3, 8, 0 );
}
if(Window_Open == __WINDOW_OPEN)
{
    cvShowImage( "Camera", frame );
    cvShowImage( "HSV", hsv_frame);
    cvShowImage( "After Color Filtering", thresholded );
}
cvReleaseMemStorage(&storage);

if( (cvWaitKey(10) & 255) == 27 ) return __BALL_TRACKING_END;

return __BALL_TRACKING_OK;
```

<Figure 8.4-10>

If the ball recognition is done through image treatment, the UXA90 needs to be operated accordingly. 'Motor_Control()' function in <Figure 8.4-8> moves the head of UXA90 (SAM actuator).

See <Figure 8.4-11> and if the coordinates of the ball in the image saved with the image treatment do not match the center coordinates of the camera, move the Yaw, Pitch motor to adjust the view of the camera and have the ball shown in the middle of the camera image. If the ball is not in the center 'Search_Completed' flag is set as 'false' and it turns to 'true' when the ball is in the center.

The function returns true(1) only when all 'Search_Completed' are set as true and the code in the 'if' sentence in <Figure 8.4-8> takes action. If not, (if the ball is not in the center of the camera), it returns false(0), therefore, the if sentence does not take action. The robot's motion code is sent as a message only when true(1) is returned. (if sentence includes the sentence to deliver the motion message for UXA90).

```
if(Ball.X_Axis >= __X_HALF_RESOLUTION + Dead_Zone)
{
    Head_Yaw.Pos_Err = ((Ball.X_Axis - __X_HALF_RESOLUTION + Dead_Zone)
                        * (Ball.Ball_size * __X_AXIS_P_GAIN));
    Head_Yaw.Pos += Head_Yaw.Pos_Err;
    Ball.X_Search_Completed = false;
}
else if(Ball.X_Axis <= __X_HALF_RESOLUTION - Dead_Zone)
{
    Head_Yaw.Pos_Err = ((Ball.X_Axis - __X_HALF_RESOLUTION - Dead_Zone)
                        * (Ball.Ball_size * __X_AXIS_P_GAIN));
    Head_Yaw.Pos += Head_Yaw.Pos_Err;
    Ball.X_Search_Completed = false;
}
else
    Ball.X_Search_Completed = true;

if(Ball.Y_Axis >= __Y_HALF_RESOLUTION + Dead_Zone)
{
    Head_Pitch.Pos_Err = ((Ball.Y_Axis - __Y_HALF_RESOLUTION + Dead_Zone)
                          * (Ball.Ball_size * __Y_AXIS_P_GAIN));
    Head_Pitch.Pos += Head_Pitch.Pos_Err;
    Ball.Y_Search_Completed = false;
}
else if(Ball.Y_Axis <= __Y_HALF_RESOLUTION - Dead_Zone)
{
    Head_Pitch.Pos_Err = ((Ball.Y_Axis - __Y_HALF_RESOLUTION - Dead_Zone)
                          * (Ball.Ball_size * __Y_AXIS_P_GAIN));
    Head_Pitch.Pos += Head_Pitch.Pos_Err;
    Ball.Y_Search_Completed = false;
}
else
    Ball.Y_Search_Completed = true;
```

```
send_std_position(Head_Yaw.ID, (unsigned int)Head_Yaw.Pos);
send_std_position(Head_Pitch.ID, (unsigned int)Head_Pitch.Pos);

if((Ball.X_Search_Completed == true) &&
   (Head_Pitch.Pos == __NECK_PITCH_MAX))
    return 1;

return 0;
```

<Figure 8.4-11>

When 'Motor_Control()' function tracks the ball and returns true(1), then the if sentences become active. This sentence uses the position of the ball (the ball is in the center of the camera, and the angle of the head motor expresses the position of the ball) to move UXA90.

That is, using the position on the head motor Yaw axis, it can move left and right, and on the Pitch axis, it can move front and back. Therefore, it can send messages like 'walk_right', 'walk_left', 'turn_right', 'turn_left', and 'walk_forward_short' to operate motions. Also if the ball is within the distance where it can be kicked, you can apply the motion message 'kick_right'. See <Figure 8.4-12>.

We will skip the explanation on 'send_msg()' function as it was mentioned previously.

```
if(Motor_Control())
{

    if((Head_Pitch.Pos == __NECK_PITCH_KICK_AREA_OFFSET) &&
       (Head_Yaw.Pos > __NECK_YAW_KICK_RIGHT_AREA_OFFSET) &&
       (Ball.Y_Axis > __Y_KICK_AREA_OFFSET))
    {
        send_msg("walk_right");
        std::cout << "WALK_RIGHT" << std::endl;
    }

    else if((Head_Pitch.Pos == __NECK_PITCH_KICK_AREA_OFFSET) &&
            (Head_Yaw.Pos < __NECK_YAW_KICK_LEFT_AREA_OFFSET) &&
            (Ball.Y_Axis > __Y_KICK_AREA_OFFSET))
    {
        send_msg("walk_left");
        std::cout << "WALK_LEFT" << std::endl;
    }


    else if((Head_Pitch.Pos == __NECK_PITCH_KICK_AREA_OFFSET) &&
            (Head_Yaw.Pos > __NECK_YAW_KICK_LEFT_AREA_OFFSET) &&
            (Head_Yaw.Pos < __NECK_YAW_KICK_RIGHT_AREA_OFFSET) &&
            (Ball.Y_Axis > __Y_KICK_AREA_OFFSET))
    {
            send_msg("kick_right");
            std::cout << "KICK_RIGHT" << std::endl;
    }

    else
    {
        if(Head_Yaw.Pos < __NECK_YAW_LEFT_TURN_AREA_OFFSET)
        {
            send_msg("turn_left");
            std::cout << "TURN_LEFT" << std::endl;
        }
```

```
        else if(Head_Yaw.Pos > __NECK_YAW_RIGHT_TURN_AREA_OFFSET)
        {
            send_msg("turn_right");
            std::cout << "TURN_RIGHT" << std::endl;
        }
        else
        {
            if((Head_Pitch.Pos == __NECK_PITCH_WALK_6SETP_AREA_OFFSET) &&
               (Ball.Y_Axis < __NECK_PITCH_WALK_4SETP_AREA_OFFSET))
            {
               send_msg("walk_forward_short");
               std::cout << "WALK_FORWARD_SHORT" << std::endl;
            }

            else if(Head_Pitch.Pos < __NECK_PITCH_WALK_6SETP_AREA_OFFSET)
            {
               send_msg("walk_forward_short");
               std::cout << "WALK_FORWARD_SHORT" << std::endl;
            }

            else
            {
                send_msg("walk_forward_short");
                std::cout << "WALK_FORWARD_SHORT" << std::endl;
            }

        }
    }
}
```

<Figure 8.4-12>

With the algorithm above, the ball is tracked and the robot can be operated accordingly. When ending the program after all the process, release all the camera information and close the image windows and close the program.

```
cvReleaseCapture( &capture );
cvDestroyWindow( "Camera" );
cvDestroyWindow( "HSV" );
cvDestroyWindow( "After Color Filtering" );
return 0;
```

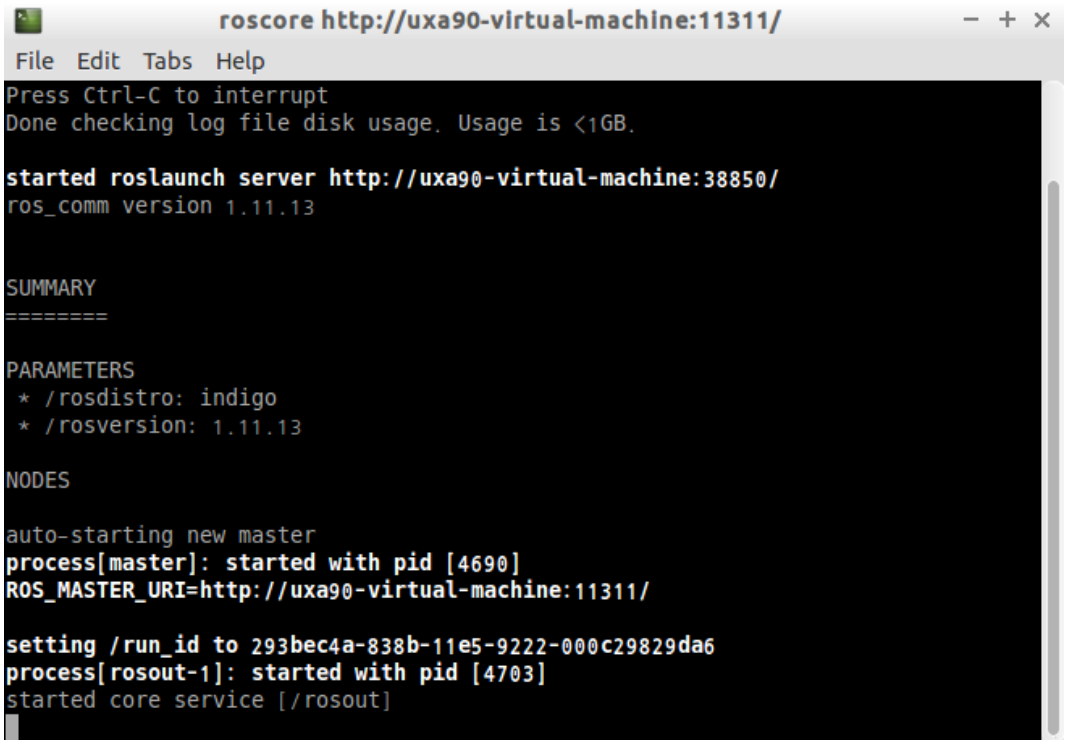
<Figure 8.4-13>

8. UXA90 package run

This chapter is on how to run the basic package for UXA90. Let's find out how to run the overall program.

First, run 'roscore' the key program in the ROS system.

$ roscore



<Figure 9-1>

After running 'roscore', run the UXA package. First, run 'uxa_serial' node. This node was explained in Chapter 7.

After establishing the serial connection, the following code allows you to check if the connection has been made properly. If so, you will see the screen shown in <Figure 9-2>.

$ ls /dev | grep ttyUSB*

<Figure 9-2>

ttyUSB0 is the serial port. This node's source code has the code that allows connection to ttyUSB0. If the number at the end changes such as to ttyUSB1, then the source code has to be modified accordingly. You can change the part in the header file of the package project as shown in <Figure 9-3>.

If the source code is changed, you have to build the package again. Refer to Chapter 6 for the guide.

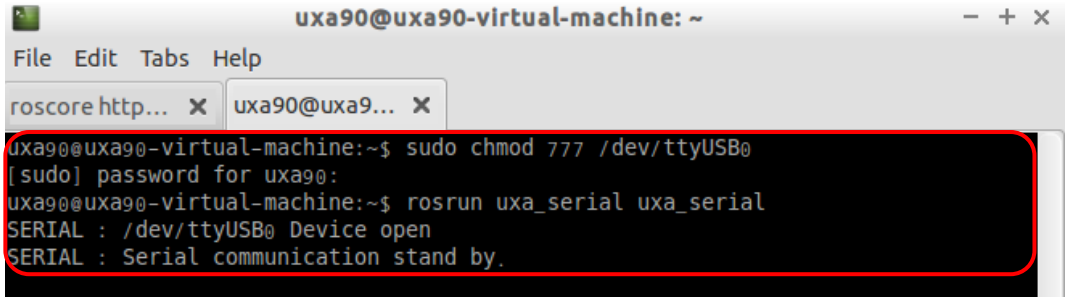
```
#define _SERIAL_PORT "/dev/ttyUSB0"
```

<Figure 9-3>

Before running the node, change the authority of ttyUSB0 port to maximum. In Linux, you cannot use hardware-related things if the authority is low.

$ sudo chmod 777 /dev/ttyUSB0

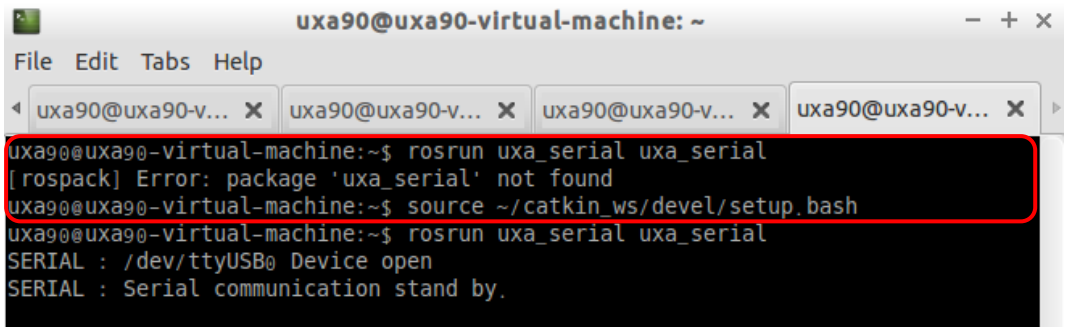
$ rosrun uxa_serial uxa_serial



<Figure 9-4>

If you see an error message like <Figure 9-5>, this means that the package you created has not been registered on the ROS package. Enter the command below and then try again.

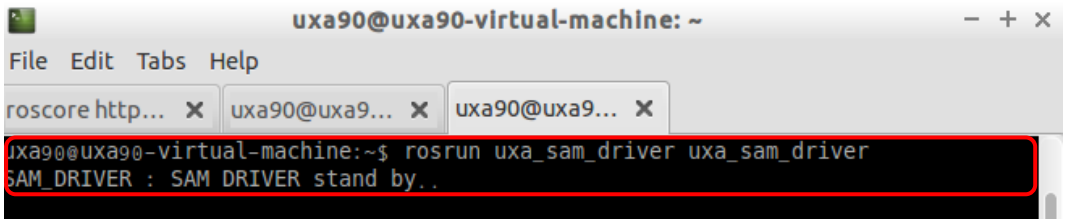
$ source ~/catkic_ws/devel/setup.bash



<Figure 9-5>

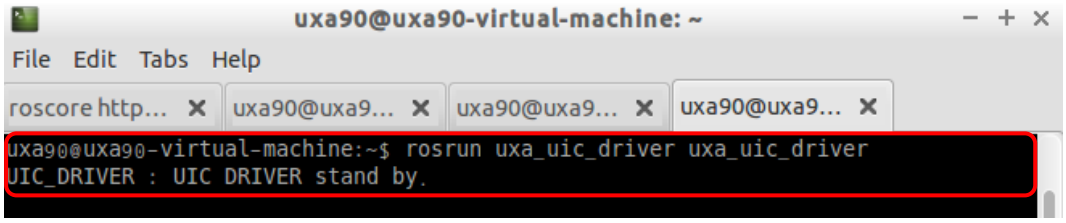
Next, run 'uxa_sam_driver' node and 'uxa_uic_driver' node. Use 'rosrun' command as you have done above.

$ rosrun uxa_sam_driver uxa_sam_driver



<Figure 9-6>

$ rosrun uxa_uic_driver uxa_uic_driver

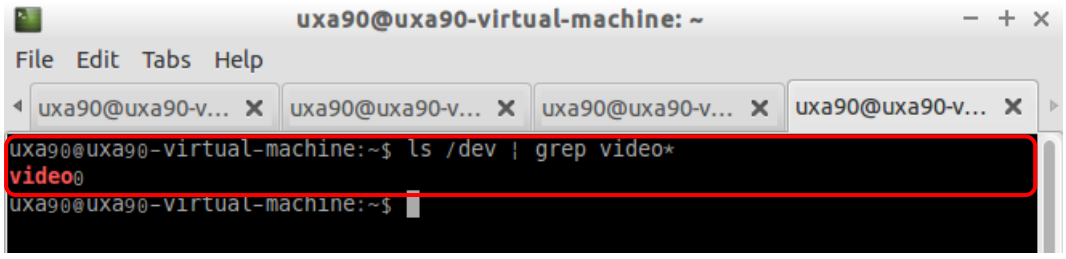


<Figure 9-7>

Lastly, run 'uxa_balltracking' node. For this package, you again have to connect the camera and match the number. Let's first enter the command that checks whether the camera is connected.

$ ls /dev | grep video*

You can see it's recognized as video0 as shown in <Figure 9-8>.



<Figure 9-8>

Just like for the serial port, the number at the end can be different. And you have to change the source code to match this number. The video port number should be changed to the factor value in 'cvCaptureFromCAM()' function.

```
int Init_Ball_Tracking(char Window_Open)
{

    CvSize size = cvSize(__X_RESOLUTION, __Y_RESOLUTION);
    // Open capture device. 0 is /dev/video0, 1 is /dev/video1, etc.
    capture = cvCaptureFromCAM(0);
    if( !capture )
    {
            fprintf( stderr, "ERROR: capture is NULL \n" );
            getchar();
            return __WEBCAM_ERROR;
    }
```

<Figure 9-9>

Also the image cannot be captured due to the camera size. See <Figure 9-10> and change the resolution.

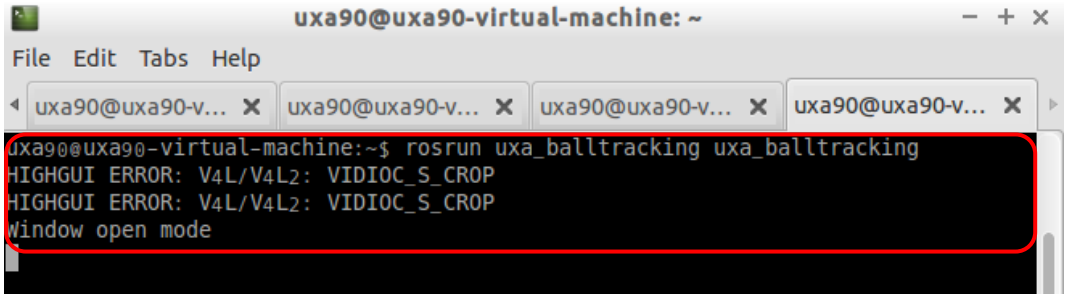
```
#define __X_RESOLUTION 320
#define __Y_RESOLUTION 240
#define __X_HALF_RESOLUTION __X_RESOLUTION/2
#define __Y_HALF_RESOLUTION __Y_RESOLUTION/2
```

<Figure 9-10>

Again, if the source code has been modified, you have to rebuild the package.

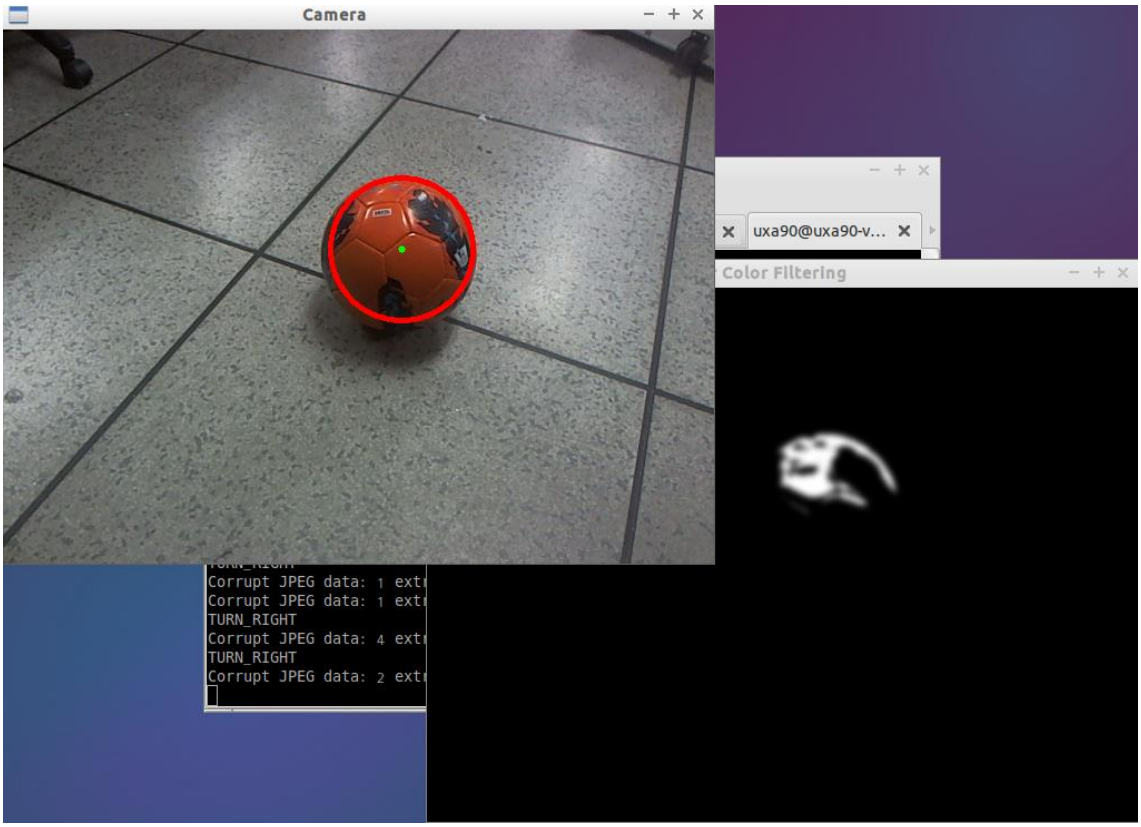
After the setting, run the node. Enter the command below to run the node, and you will see the screen displaying the image from the camera.

$ rosrun uxa_balltracking uxa_balltracking



<Figure 9-11>

Once the screen displays the image, the orange ball is recognized following the programmed algorithm, and UXA90 operates accordingly.



<Figure 9-12>