How to use the NXT_driver r2d2mipal and the corresponding NXT_controller

Vladimir Estivill-Castro
MiPal

September 25, 2015

Abstract

This document gets you started on using the NXT_driver r2d2mipaland the corresponding NXT_controller. It can be used as a tutorial to gain an understanding of very basic C++11 programming to control an LEGO-NXT brick. Nevertheless, we also create a hardware abstraction of a differential robot constructed in a standard way for the PUSH approch from ROS.

We recommend you also become familiar with the complementing Model-Driven Development paradigm of logic-labeled finite-state machines (*llfsms*).

Contents

1 The set-up

This document complements the use of clfsm with ROS. In provides and example outside a simulator, using the LEGO-NXT robot. To properly install in Ubuntu 14.04 the the NXT_driverr2d2mipal module you need the blue-thooth library and the usb-library:

```
sudo apt-get install libbluetooth-dev
sudo apt-get install libusb-1.0-0-dev
```

Download the NXT_driver r2d2mipal module, it is a catkin package.

```
git clone https://github.com/mipalgu/NXTdriver.git
```

Place it in your catkin worksapce and compile it with catkin_make. If you have a LEGO-NXT robot and you connected to the USB to your Ubuntu, then you can run the demo program.

```
sudo devel/lib/r2d2mipal/testDemo
```

Every time you press the sensor, motor A spins, and you see the revolutions in the standard output. Note, in Ubuntu, the NXT grabs the device as root by defualt. You can change the usb device with

```
sudo chmod ugo+rwx /dev/bus/usb/02/*
```

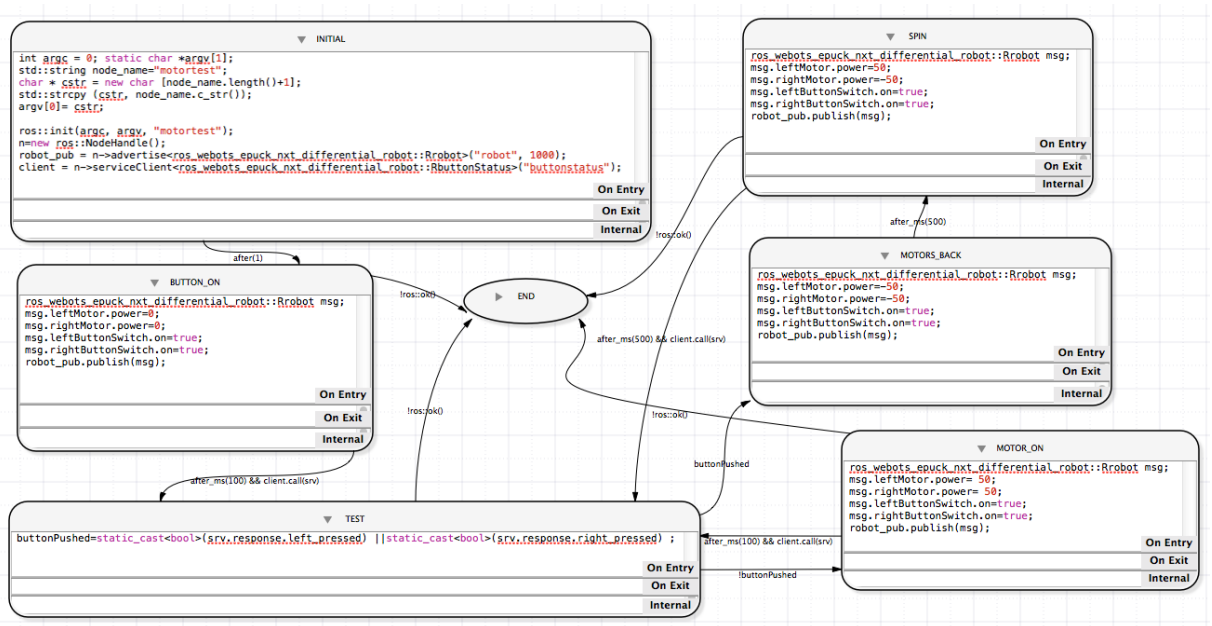

Figure 1: Machine that represents a very simple behavior. It is compiled and executed by *MiPal*'s `clfsm` and uses `ROS`-msg an `ROS`-srv.

2 An example of a behavior by a simple *llfsm* using the `LEGO-NXT` robot and `ROS`

This is an illustration of building a logic-labeled finite-state machine and demonstrate running the interpreter `clfsm` ot a robot (the `LEGO-NXT`) with only `ROS` interfaces (`ROS`-msg and `ROS`-srv, that is `ROS`-messages and `ROS`-services). It requires familiarity with the beginning tutorials of `ROS` and the use of MIEDITLLFSM to build a *llfsm*. The other aspect is you need `clfsm` installed to run the machine.

Figure 1. presents a diagram of the machine we will build.

It should not be hard to follow this behavior. The behavior starts in the state INITIAL where the `ROS` handlers for sending commands to the robot or receiving sensor information are initialized. After one second (transition with label `after(1)`, we move to the state BUTTON_ON). In the state BUTTON_ON, a command is sent to turn the touch sensors on (both the left and right). There is a transition `after_ms(100) && client.call(srv)` to the state TEST (this transition means that it must be the case that 100 mili-seconds have occurred and we have received sensor information on the service). The state TEST simply updates the Boolean variable `buttonPushed` to whether either button or both are pushed. The **OnEntry** section of the TEST state is as follows.

```
buttonPushed=
static_cast<bool>(srv.response.left_pressed) ||static_cast<bool>(srv.response.right_pressed)
;
```

If no button has been pushed (no obstacle), we move to the state state MOTOR_ON. In the state MOTOR_ON, the motors are set forward. After a little while we read the sensors again and return to the state TEST. If a button has been pushed, then we actually move to the state MOTORS_BACK that sets the motors back and after 500 mili-seconds (half a second), we go to the state SPIN. In SPIN one motor goes forward and the other backward, so the robot spins until

we move back to TEST which is after half a second and another reading of the sensors. The END state is empty and is a terminal state we arrive if the ROS-systems is going down.

Running this demonstration machine

To run this demonstration machine you need to download the *MiPal* clfsm.tar.bz2 and install and compile it correctly. You need a LEGO-NXT robot configured as a differential two wheels robot, with bumper for the 2 touch sensors at the front. Motors should be on B and C. Input port for the button/touch sensors should be 1 and 2.

Make sure you can run the testDemo program of the NXT_driver. This program is built in the /devel/lib/r2d2mipal directory after issuing the corresponding catkin_make.

Recall that in Ubuntu, connecting the LEGO-NXT may require that you update the permission to the USB port. Look what is the new port and usually something like

```
sudo chmod ugo+rwx /dev/bus/usb/001/038
```

will enable the testDemo to run and connect via USB to the LEGO-NXT. The program starts motor A when button in 2 is pushed and shows readings of the rotors.

Download the catkin package NXT_controller ros_webots_epuck_nxt_differential_robo (this is the package that has the nxt_controller). To download the NXT_controller r2d2mipal module, it is a catkin package.

```
git clone https://github.com/mipalgu/NXTcontroller.git
```

Put this package in your working space and issue a catkin_make. You should see that now also the programs in this package are built. You should find the file devel/lib/ros_webots_epuck_nxt_differential_robot/nxt_controller/nxt_controll has been produced.

Also, download the machine motorTest.machine.tar. Unpack the machine with tar -xvf and place the folder motorTest.machine in the folder machines.

```
cd $DOWNLOADS
tar -xvf motorTest.machine.tar
cd $HOME/catkin_ws/src
catkin_create_pkg motorTest std_msgs roscpp clfsm libclfsm
cd motorTest
mkdir -p machines
mv $DOWNLOADS/motorTest.machine machines
```

We recommend the building of a catkin package for this machine using the method described with the demonstraton *llfsms* for ROS and clfsm. That is using the script machine_step.sh

Alternatively, you can use the bmake method also described with the demonstraton *llfsms* for ROS and clfsm. With the bmake method , place the given Makefile (also in the same download section from *MiPal*'s downloads inside the recent machines directory. Check the Makefile lsit the motorTest as a machine to compile. Then, issue bmake. The machine is compiled.

Make sure your robot has room to run and move. With a LEGO-NXT connected vis USB to your computer, start roscore in one terminal, start the controller in another terminal

```
./devel/lib/ros_webots_epuck_nxt_differential_robot/nxt_controller/nxt_controlle
```

In a third terminal start the finite state machine with clfsm.

```
cd $HOME/catkin_ws/machines
../devel/lib/clfsm/clfsm -v motorTest.machine
```

The robot will act!

Building the machine yourself, and understanding the details

Using the INCLUDE button in MIEDITLLFSM you open the window to define the global include paths for this machine.

```
#include "ros/ros.h"
#include "ros_webots_epuck_nxt_differential_robot/Rrobot.h"
#include "ros_webots_epuck_nxt_differential_robot/RbuttonStatus.h"
#include "CLMacros.h"
```

The ROS infrastructure will be necessary, thus `ros/ros.h`. Our messages to send command control to the robot will be defined in `Rrobot.h`, while the services will be in `RbuttonStatus.h`.

The variables (use the VARIABLES button to add them are:

```
ros::NodeHandle* n
ros_webots_epuck_nxt_differential_robot::RbuttonStatus srv
ros::Publisher robot_pub
bool buttonPushed
ros::ServiceClient client
```

And the **OnEntry** section of the INITIAL state is as follows.

```
int argc = 0;
static char *argv[1];
std::string node_name="motortest";
char * cstr = new char [node_name.length()+1];
std::strcpy (cstr, node_name.c_str());
argv[0]= cstr;
ros::init(argc, argv, "motortest");
n=new ros::NodeHandle();
robot_pub =
n->advertise<ros_webots_epuck_nxt_differential_robot::Rrobot>("robot", 1000);
client =
n->serviceClient<ros_webots_epuck_nxt_differential_robot::RbuttonStatus>("buttonstatus");
```

So, perhaps it is a good idea to have a look at the `catkin` package for the `NXT_controller` `ros_webots_epuck_nxt_differential_robot`
(this is the package that has the `nxt_controller`).

You can inspect the soruce code of the package yourself. We recommend this as it will review some aspects of ROS.

```
cd $HOME/catkin_ws/src
catkin_create_pkg ros_webots_epuck_nxt_differential_robot std_msgs
roscpp
```

You can construct the types for the ROS-msg.

```
cd $HOME/catkin_ws/src/ros_webots_epuck_nxt_differential_robot
mkdir msg
cd msg
```

You can create the file `Rmotor.msg`. With your favorite text editor. The content is

```
Header Rmotor
int32 power
```

That is, a motor command is defined by an signed integer giving it power (positive is forwards, negative is backwards and 0 is stop; and we will make it in relative terms: a percentage of maximum power).

A second message file is `Rbutton.msg` and its content is

```
Header Rbutton
bool on
```

This Boolean value indicates whether the button is active or not. With this two message files, we can actually build the message structure for commands to the robot with the behavior above. Create the file `Rrobot.msg` with the content

```
Header Rrobot
ros_webots_epuck_nxt_differential_robot/Rmotor leftMotor
ros_webots_epuck_nxt_differential_robot/Rmotor rightMotor
ros_webots_epuck_nxt_differential_robot/Rbutton leftButtonSwitch
ros_webots_epuck_nxt_differential_robot/Rbutton rightButtonSwitch
```

That is, our commands to a robot instruct it on what power its left and right motor shall have and if the touch/button sensors are active or not.

Now, it should be no mystery why the **OnEntry** section of the state Button_On looks as follows.

```
ros_webots_epuck_nxt_differential_robot::Rrobot msg;
msg.leftMotor.power=0;
msg.rightMotor.power=0;
msg.leftButtonSwitch.on=true;
msg.rightButtonSwitch.on=true;
robot_pub.publish(msg);
```

In `msg`, we get an object of the class `ros_webots_epuck_nxt_differential_robot::Rrobot`, we complete its fields. Both motors are halted and both sensors are activated. Then, we publish this message through the `ROS` handler.

Look at how similar are the states MOTOR_ON, MOTORS_BACKWARDS, and SPIN.

When you create the message types that you need, the `ROS` environment must be instructed to create the marshaling and types for connection. The first place this has to be specified is in the file `package.xml` of the package. You needs lines like

```
<build_depend>message_generation</build_depend>
<run_depend>message_runtime</run_depend>
```

to indicate we are going to use `ROS`-msg and `ROS`-srv.

The second part where this is specified in the `CMakeLists.txt` of the package. There, we need to set up sections like.

```
find_package(catkin REQUIRED COMPONENTS
roscpp
std_msgs
message_generation
r2d2mipal
)
```

This is also to show that we will use *MiPal*'s `NXT_driver` `r2d2mipal` as the connection to the robot. We need also as follows.

```
catkin_package(
# INCLUDE_DIRS include
LIBRARIES ros_webots_epuck_nxt_differential_robot
CATKIN_DEPENDS roscpp std_msgs message_runtime
r2d2mipal # DEPENDS system_lib
DEPENDS ${LIBUSB_LIBRARY}
)
```

Then we need to specify our message files.

```
add_message_files(
FILES
# Message1.msg
# Message2.msg
Rmotor.msg
Rbutton.msg
Rrobot.msg
)
```

And also our services files.

```
add_service_files(
FILES
# Service1.srv
# Service2.srv
RbuttonStatus.srv
)
```

We need to generate the headers.

```
generate_messages(
DEPENDENCIES
std_msgs
)
```

Place also the corresponding flags in the build section

```
set(CMAKE_CXX_FLAGS "-std=c++11")
```

And where to find the includes of `r2d2mipal`

```
include_directories(
${catkin_INCLUDE_DIRS}
)
include_directories(${r2d2mipal_INCLUDE_DIRS})
```

Now we can have the directive on how to build the main node, that is the `nxt_controller`.

```
add_executable(nxt_controller src/nxt_controller.cpp src/nxt_interface.cpp)
```

Which has some dependencies.

```
add_dependencies(nxt_controller ros_webots_epuck_nxt_differential_robot_generate_messages_cpp
r2d2mipal)
```

And its linking is performed as follows.

```
target_link_libraries(nxt_controller
r2d2mipal
${LIBUSB_LIBRARY}
${LIBBLUETOOTH_LINKER_FLAGS} ${LIBBLUETOOTH_LIBRARY}
$catkin_LIBRARIES
)
```

The ROS-srv

To find out the status of sensors we will use ROS-srv.

```
cd $HOME/catkin_ws/src/ros_webots_epuck_nxt_differential_robot
mkdir srv
cd srv
```

In this case, the file that describes the the ROS-service (request and response pair) is simple. The file is named RbuttonStatus.srv and its contents is as follow.

```
--
bool left_pressed
bool right_pressed
```

That is, the request part is empty, while the response part has two Boolean fields indicating whether the corresponding sensors is pressed or not.

The structure of the nxt_controller

The node nxt_controller is defined by a simple starting program nxt_controller.cpp

```
/**
 * \file    nxt_driver.cpp
 *   nxt_driver.cpp
 *   Created by
 *   \author Vlad Estivill-Castro
 *   \date 13/10/2014.
 */



#include "ros_webots_epuck_nxt_differential_robot/nxt_interface.h"


int main(int argc, char **argv)
    {
       NXT_interface  *subscriber = new NXT_interface();


    subscriber->run(argc, argv);


    return 0;
}
```

This program is in the source (i.e. src) directory of the package. It is simple because all the work is done by the class in nxt_interface. In this main program, we only create an object of the class NXT_interface and then invoke the run method on it.

So, lets look at the header file of the nxt_interface: nxt_interface.h

```
/**
 * \file    nxt_interface.h
 *   nxt_interface.h
 *   Created by
 *   \author Vlad Estivill-Castro
 *   \date 14/10/2014.
```

```
*/


#include "r2d2_base.h"
#pragma clang diagnostic ignored "-Wold-style-cast"
#include "usb.h"

#include "ros/ros.h"
#include "ros_webots_epuck_nxt_differential_robot/Rmotor.h"
#include "ros_webots_epuck_nxt_differential_robot/Rbutton.h"
#include "ros_webots_epuck_nxt_differential_robot/Rrobot.h"
#include "ros_webots_epuck_nxt_differential_robot/RbuttonStatus.h"

class NXT_interface
{
          public:
            ///< constructor
        NXT_interface();

        std::string banner() { return std::string("(c) Vlad Estivill_Castro, demo subscriber R@D@-NXT
ROS driver");}

    void run(int argc, char **argv);

        /// call-back method robot
void   robotCallback(const ros_webots_epuck_nxt_differential_robot::Rrobot::ConstPtr& msg);

        /// call-back method button status/value
    bool value_buttonCallback(
             ros_webots_epuck_nxt_differential_robot::RbuttonStatus::Request & req,
             ros_webots_epuck_nxt_differential_robot::RbuttonStatus::Response& res);

    private:
            r2d2::Brick* brick;
        r2d2::NXT* nxt;
        r2d2::Sensor* sensor_left; r2d2::Sensor* sensor_right;
        bool status_sensor_left; bool status_sensor_right;

        r2d2::Motor* motor_right; r2d2::Motor* motor_left;
};
```

The program needs to know about the `NXT_driverr2d2mipal` and about the `usb` library for connecting to the `LEGO-NXT`. Its has a constructor, a banner for feedback, and a call-back for the control messages and a callback for the status services. Naturally, a `run()` method to spin/sleep, as the work will be performed by the call-backs. The private variables are to hold the data-structures to actually work as an interface. They will be initialized in the constructor. They will

hold the usb-connection and boolean to record if our sensors are on or off, and the actual objects for sensors and motors to issue `r2d2mipal` commands to them. The file `nxt_interface.h` is in the `include` directory of our package and has a path as the name of our package.

So, now we are in a position to look at the definition of the interface. This is the file `nxt_interface.cpp` in the source directory of our package. `nxt_interface.cpp`

```
/**
 * \file    nxt_interface.cpp
 *    nxt_interface.cpp
 *    Created by
 *    \author Vlad Estivill-Castro
 *    \date 14/10/2014.
 */


#include "ros_webots_epuck_nxt_differential_robot/nxt_interface.h"


            ///< constructor
NXT_interface::NXT_interface()
    { banner();


    r2d2::USBBrickManager usbm;


    brick = usbm.list()->at(0);


        nxt = brick->configure(r2d2::SensorType::TOUCH_SENSOR,
                                        r2d2::SensorType::TOUCH_SENSOR,
                                        r2d2::SensorType::NULL_SENSOR,
                                        r2d2::SensorType::NULL_SENSOR,
                                        r2d2::MotorType::STANDARD_MOTOR,
                                        r2d2::MotorType::STANDARD_MOTOR,
                                        r2d2::MotorType::STANDARD_MOTOR);
        if (nxt != nullptr) {
        /* check the connections of the NXT you are using */
            sensor_right = nxt->sensorPort(r2d2::SensorPort::IN_1);
            sensor_left = nxt->sensorPort(r2d2::SensorPort::IN_2);
            motor_right = nxt->motorPort(r2d2::MotorPort::OUT_B);
            motor_left = nxt->motorPort(r2d2::MotorPort::OUT_C);
        // initially sensors are off
        status_sensor_left = false;
        status_sensor_right = false;
               ROS_INFO("Connection established" );
    }
    else
               ROS_INFO("ERROR: Conenction failed" );
    }
```

```
void NXT_interface ::  run(int argc, char **argv)
{
        if (nxt != nullptr) {
                ros::init(argc, argv, "nxt_driver");
                ros::NodeHandle n;

                ros::Subscriber subRobot = n.subscribe("robot", 1000, & NXT_interface::robotCallback,this);

                ros::ServiceServer serviceStatusButton = n.advertiseService("buttonstatus",  & NXT_interface::value
        ROS_INFO("Service ready");
                ros::spin();

        std::cerr<< "This EXITING sometimes does not happen when roscore goes down" << std::endl;
        }
}

        /// call-back method button status/value
bool NXT_interface :: value_buttonCallback(
             ros_webots_epuck_nxt_differential_robot::RbuttonStatus::Request & req,
             ros_webots_epuck_nxt_differential_robot::RbuttonStatus::Response& res)
{
    if (status_sensor_right)
    { res.right_pressed=sensor_right->getValue();
    }
    if (status_sensor_left)
    { res.left_pressed=(true==sensor_left->getValue());
    }

    if (! (status_sensor_left || status_sensor_right))
        {
               ROS_INFO("service invoked with both SENSOR OFF:" );
               ROS_INFO("FALSE exit:" );
           return false;
        }
    else
    {
        return true;
    }
}

        // call-back method robot_control
void NXT_interface ::  robotCallback(const ros_webots_epuck_nxt_differential_robot::Rrobot::ConstPtr&
msg) {
    int leftPower=msg->leftMotor.power;
    int rightPower=msg->rightMotor.power;
```

```
    ROS_INFO("Setting Motors Left: [%d] Right: [%d]", leftPower,rightPower);
    if ( (leftPower) && (100>= leftPower) )
        motor_left->setForward(leftPower);
    else    { // negative values should be back
         leftPower= - leftPower;
         if ( (leftPower) && (100>= leftPower) )
             motor_left->setReverse(leftPower);
         else // stop with power ==0
         motor_left->stop(false);
        }

    if ( (rightPower) && (100>= rightPower) )
        motor_right->setForward(rightPower);
    else
        {   rightPower= -rightPower;
         if ( (rightPower) && (100>= rightPower) )
             motor_right->setForward(rightPower);
         else
             motor_right->stop(false);
        }

    status_sensor_left =static_cast<bool>(msg->leftButtonSwitch.on);
    status_sensor_right =static_cast<bool>(msg->rightButtonSwitch.on);
    ROS_INFO("Switch sensors Left: [%s] Right: [%s]", status_sensor_left ? "ON" : "OFF",  status_sensor_right?
"ON" :"OFF");
}
```

The constructor attempts to establish the USB connection. If it works, it configures the brick using the `NXT_driver-r2d2mipal`'s interface, and creates the objects for the sensors and the motors (the private attributes). It also sets that the sensors are not active.

The `run()` methods should not be surprising to those used to writing `ROS`-subscribers or `ROS`-service providers. It gets a `ROS`-handle and subscribes the corresponding call-backs to the named topics. Then it spins.

The first callback `value_buttonCallback()` is the one that handles a request about the value of the touch sensors. Note that the signature is a bit more complicated as our call-backs are methods of our class. The code is rather simple, we check if our sensors are active, indicated by our private variables for this. We then use the `NXT_driver-r2d2mipal` methods to find the value of the corresponding sensor and place them in the fields of the response. We answer `false` if both sensors were inactive, we want to enforce that programs that use us, invoke values of the sensors after activating them.

The callback for the control message is `robotCallback()` and this one is more standard. It collects the values of motors and acts if they re in the range [-100,100]. It sets the motors forwards if the value is positive, and in reverse if it is negative, while stopping if the value is zero. At the

end of this method we also extract commands about our sensors, turning off or on according to the value we receive for this in the message.