# Teaching Autonomous Robotics Using Player and ROS: A CS 148 Starter Guide (DRAFT)

Barbara Teresa Korel  Odest Chadwicke Jenkins

May 21, 2010

# Foreword

This draft contains the first three chapters of the developing textbook "Teaching Autonomous Robotics Using Player and ROS". This draft satisfies the project requirements for Barbara Korel towards the completion of her Masters degree in Computer Science at Brown University, under the supervision of Prof. Chad Jenkins. This book is based on the development and experiences restructuring an undergraduate robotics course, Brown University CS 148 "Building Intelligent Robots", for the iRobot Create platform. Chad Jenkins has taught this updated version of CS 148 since 2006, with Barbara Korel serving as Head Teaching Assistant in 2009.

This draft is intended to serve as a "how-to" guide to setup a basic low-cost mobile robot platform for use in an upper-level undergraduate robotics course. Subsequent chapters will cover project modules suitable for courses using the described robot platform. The projects will cover bug and random navigation algorithms, color object recognition, path planning, robot localization, subsumption architectures, multi-robot coordination, learning from demonstration, and experimentation with significance testing. Up-to-date developments for these materials can be found from the CS 148 website (http://www.cs.brown.edu/courses/cs148) and the Brown Robotics code repository (http://code.google.com/p/brown-rlab/).

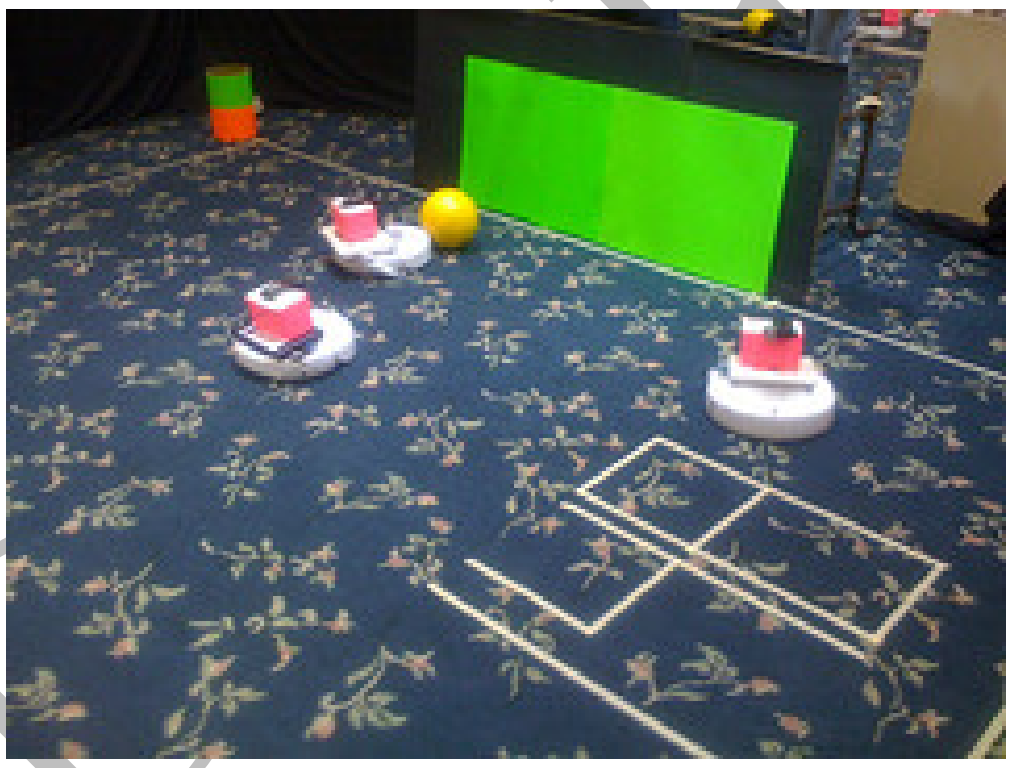
# TEACHING AUTONOMOUS MOBILE ROBOTICS USING PLAYER AND ROS


Odest Chadwicke Jenkins
Barbara Korel

# Contents

# Chapter 1

# Course Objectives



This book is based on our experiences developing and restructuring an undergraduate robotics course, Brown University CS 148 "Building Intelligent Robots", for the iRobot Create platform. Taking the perspective of CS 148, we describe our objectives and approach for an upper-level course in autonomous robotics. Our main educational goal is to focus on the computational aspects of robotics while minimizing the necessary mechatronic startup costs, which is best suited for a complementary course. To this end, we use a solderless robot platform comprised of "off-the-shelf" components that can be readily assembled by undergraduate (and secondary) students. This platform is meant to be both a quick entry point for students to start programming robots and, using research-grade robot middleware (e.g., Player, ROS), a path that can directly lead to understanding and extending the state-of-the-art in robotics.

Figure 1.1: The Aldebaran Nao.

For instructors, this book is meant to be a "how-to" guide for getting an autonomous robotics course modules up-and-running with as little overhead as possible. The first three chapters describe how to assemble, program and remotely control a basic low-cost mobile robot platform. Subsequent chapters cover project modules suitable for this (and similar) mobile robotics, as conducted in CS 148 in recent years. These projects build upon each other from bug and random navigation algorithms and color object recognition up through path planning, robot localization, subsumption architectures, multi-robot coordination, learning from demonstration, and experimentation with significance testing. These projects are pedagogically cast within a "robot control loop" (consisting of sensing, perception, decision making, motion control, and physics). The control loop serves as a conceptual scaffold similar to the notion of a pipeline in computer graphics.

In terms of collaboration, this book is intended to be a community resource that facilitates sharing and collaboration in and beyond the classroom. Given the larger integrative nature of robotics projects, CS 148 projects utilize version control systems (Subversion in particular) to enable collaboration among student groups, reproducibility by course graders, and portability when moving code across robots. For educators, this book itself and supporting materials will be maintained (and continually updated) in a Subversion repository:

`http://brown-rlab.googlecode.com/svn/trunk/teaching_autonomous_robotics`

We encourage educators to freely use the book and materials in their courses and broader educational efforts. These materials are meant to be copy-pasted, modified, tested, and refined as a community to improve the quality and accessibility of robotics education. We welcome and encourage contributions to the continued development of this book, especially those that can be patched into the repository. Up-to-date usage of these materials in CS 148 can be accessed through the course website:

`http://www.cs.brown.edu/courses/cs148`

## 1.1 CS 148 Course Description

`(taken from Brown 2009-10 course catalog)`

CS148 is an introduction to fundamental topics in autonomous robot control. This course focuses on the development of "brains" for robots. That is, given a machine with sensing, actuation, and computation, how do we develop programs that allow the machine to function autonomously? We answer this question through a series of class discussions and group projects.

CS148 class meetings cover technical and algorithmic aspects of robotic decision making and perception, as well as societal and philosophical implications posed by autonomous robots. Discussions amongst the class pose and address questions related to how robots can contribute to society, what technical functionality is needed, and how will such technologies affect the human-robot dynamic.

CS148 projects center on a "robot soccer" task, where students program robot vacuum-like devices to play soccer in a structured environment. Various approaches to robot control (spanning reaction to deliberation) are covered using the Brown iRobot Create platform robots (which are iRobot Roomba/Create hardware with onboard ASUS EeePC subnotebooks, as seen in Figure 1.2 on the far right). We support either the Player/Stage/Gazebo (PSG) robot framework or the Robot Operating System (ROS) as a robot middleware solution for the course projects. Similarly, the project implementations are not restricted to the iRobot Create platforms. Rather any robot that has the physical components for playing soccer, such as Aldebaran's Nao humanoid robot (Figure 1.1), can be used to program the assignments. Thus, the practical aspects of this course are not limited to a specific robot platform or middleware.

Figure 1.2: Evolution of the Brown SmURV robot platform. Version 0.5 (2006), Logitech QuickCam and iRobot Roomba connected to a larger Dell laptop, usually carried offboard, using a RoboDynamics RooStick. Version 1.0 (2007), a Mini-ITX computer and Unibrain Fire-I camera connected to and powered by an iRobot Create. Version 1.5 (2008), a One Laptop Per Child XO, with onboard camera, connected to a Roomba. Version 2.0 (2009), an Asus EEE PC and Logitech QuickCam Ultra on an iRobot Create.

## 1.2 Why Robotics?

Why should anyone consider robotics? One quick answer: the 3 D's, "Dirty, Dull, and Dangerous." Robots have long been proposed as a way to relieve humans from tasks necessary for society that can be unpleasant, mundane, or hazardous to human health. More generally, robotics can be considered a means to enhance physical human productivity that augments or even multiplies a human user's ability to perform tasks in the physical world.

The application of robots already has a strong presence in many different fields, illustrated in the collage of Figure 1.3. In healthcare, robots have assisted doctors, nurses, and caretakers in the delivery of medicines in hospitals, surgical operations, and post-stroke rehabilitation. The use of robotics has the potential to reduce health care costs and help the elderly and chronically ill to remain independent. Exploration robots have been able to collect scientific data on planets and deep sea areas that would be difficult (perhaps impossible) for humans to reach. Domestic service robots such as the iRobot Roomba and Willow Garage's PR2 can perform basic household chores, such as vacuuming and laundry. Robots in manufacturing assist with labor intensive jobs to lower production costs and increase productivity. Unmanned vehicles for military applications, such as the General Atomics MQ-1 Predator and iRobot PackBot, can provide greater reconnaissance and reduce the risk of human casualties by using robots in place of humans. Along with the benefits of each of these applications, robots raise serious questions about unintended or harmful consequences, such as the effects to world labor markets and global armed conflicts.

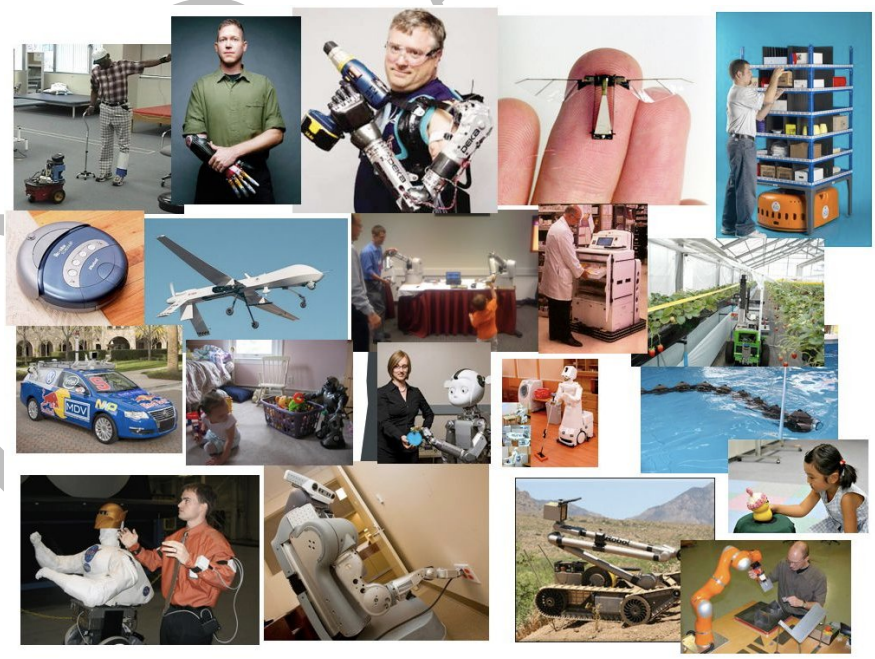


Figure 1.3: Examples of robotic applications.

Given this wide array of applications, it is important to note that robotics is not neces-

sarily about to replace human labor. Although not currently the case, at some point, robots may outperform humans at certain jobs due to being cheaper, faster, or easier to train. However, robots will not and should not be the right solution for all of society's labor. Rather, robotics should complement human productivity. There are many tasks that will remain difficult for robots, such as those that involve intuition, uncertainty, or human sociability. The most productive use of robots will likely specialize robotic tasks to those that are not necessarily suited to or desired by their human users. For example, instead of spending an hour vacuuming their house, a human user can delegate this task to a robot and spend this hour exercising.


Figure 1.4: Examples of heterogeneous computing devices.

Robotics can also be thought of as an extension of computing and the Internet beyond digital worlds into the physical world that we inhabit. The explosion of personal computing devices present in our everyday lives, as seen in Figure 1.4, have become not just an amenity but a necessity. Personal computing has enabled individuals to significantly increase their productivity in digital domains, such as electronic communication, publishing, and virtual telepresence (teleconferencing, video games, etc.). Similarly, the prevalence of commodity robotic devices will move this productivity past the digital boundary and into the physical world.

Finally, whereas personal computing has enabled individuals to manage digital information by supervising teams of heterogeneous computing devices, autonomous robotics will enable individuals to manage physical environments by supervising teams of heterogeneous robotic devices, as seen in Figure 1.5. If a heterogeneous collection of robots is to affect the physical world by interacting with and manipulating its features to achieve a predefined desire, the coordination among the robotic devices is essential. The question of how to coordinate and control a team of heterogeneous robotic devices is still an open challenge and needs to be addressed for this vision to be realized.

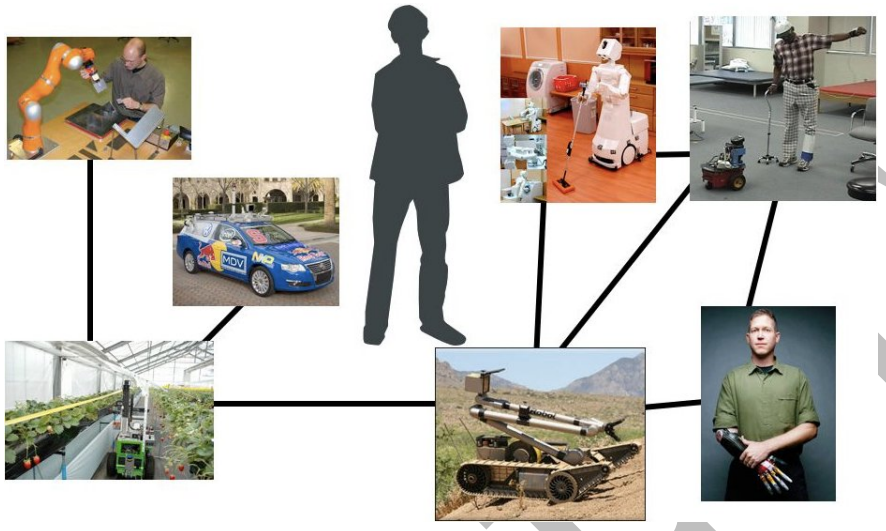
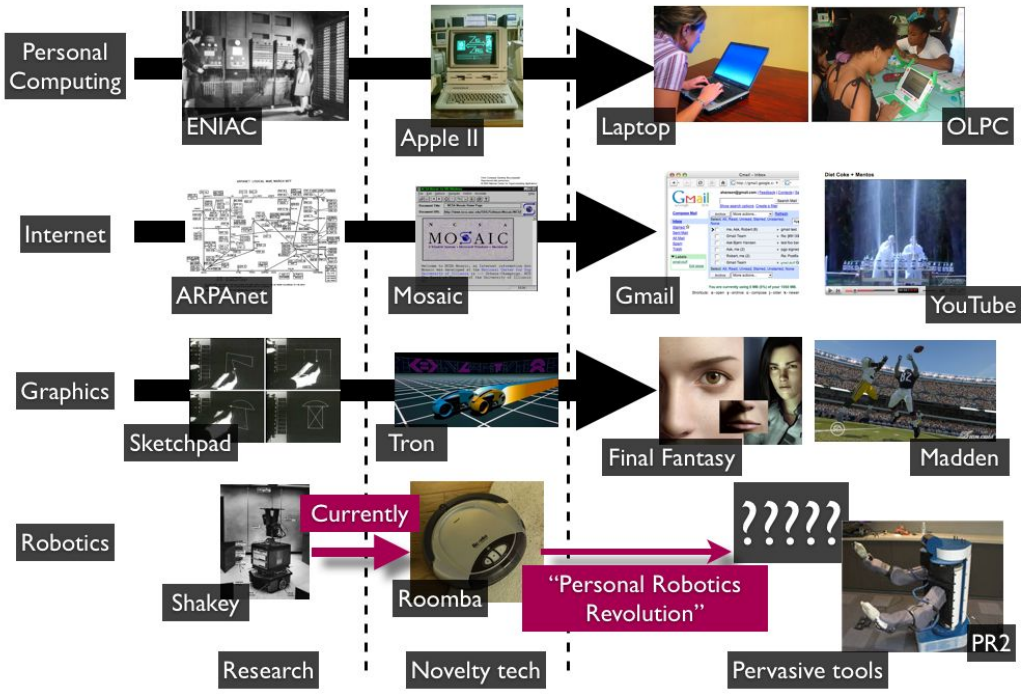


Figure 1.5: Examples of heterogeneous robotic devices.

### 1.2.1 The Personal Robotics Revolution

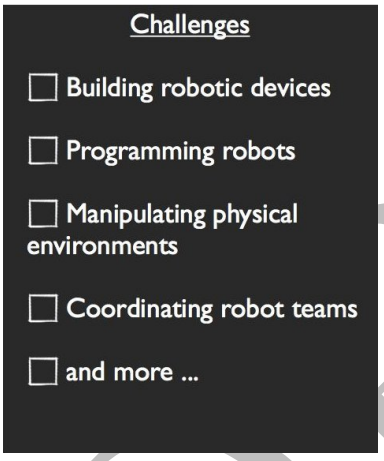



While problematic, the challenges of the physical world offer new and great opportunities for the pioneering spirit. Think of robotics as a means to extend computing and the Internet

beyond controlling digital environments into autonomously working in the physical world. This is the perfect time to be a roboticist because innovators can shape the technology! Similar to the "personal computing revolution" a few decades ago, robotics is on the dawn of its own "personal robotics revolution", with many great robotic technologies that are now starting to come together.

Robotics is currently in the same stage of the exponential technology growth as personal computing in the 1970s, computer graphics in the 1980s and the Internet in the early 1990s. These three trends all hit the technology exponential, explained by Moore's law which describes "a long-term trend in history of computing hardware, in which the number of transistors that can be placed inexpensively on an integrated circuit has doubled approximately every two years."[1]. Robotics is currently a technology exponential in the making, thus this is the time to get involved in robotics!

### 1.2.2 CS 148 Disclaimer: the real world is unforgiving




The first prerequisite for building intelligent robots is to engineer robotic devices. Provided the abundance of robot systems along with the commercial availability of relatively cheap robots such as the Roomba, for the purposes of this class we will assume this prerequisite has been successfully met. The next challenge is to program robots to function autonomously in an unpredictable world. This is the challenge CS148 will address.

This course involves a significant amount of programming and decision making under uncertainty. Robotics is about understanding and functioning in the real world. However, the real world is highly dynamic, uncontrolled, and nondeterministic. These factors result in uncertainty that is unlike the structure and determinism of traditional computer science. Programming in the face of such uncertainty is a persistent challenge faced at all levels of robotics. Keep in mind that robots are not sentient, but their programmers are!

## 1.3 Autonomous Robotics

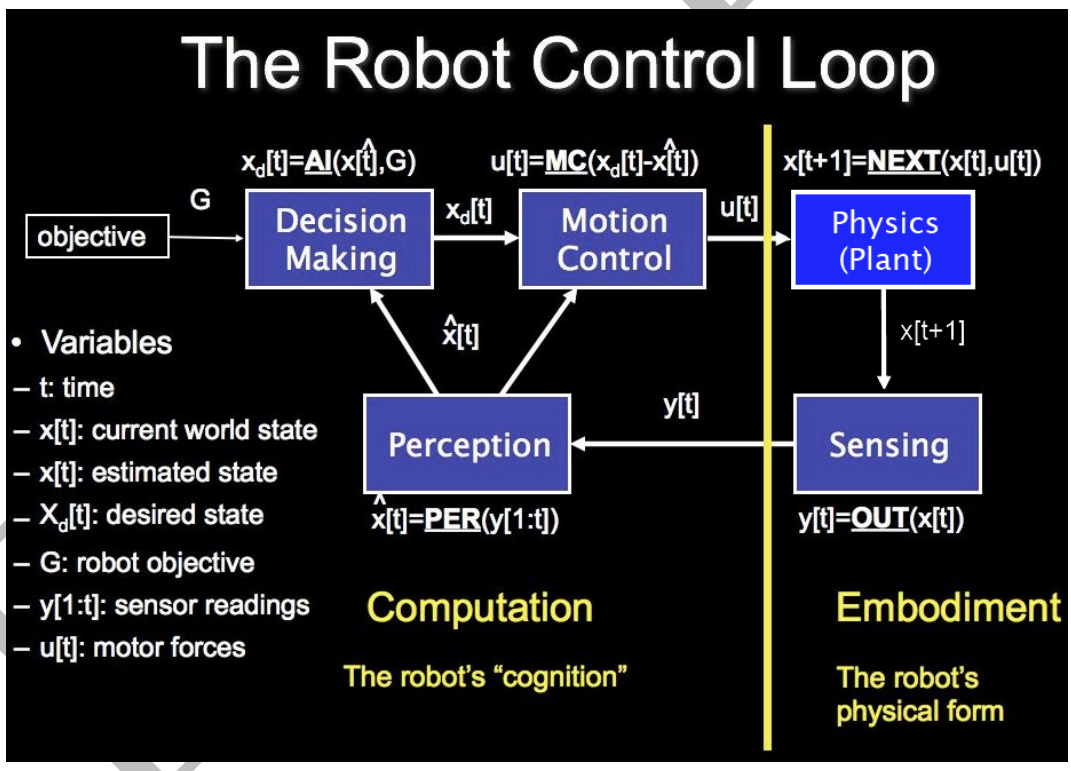
We define an autonomous robot as a decision making agent with a physical embodiment that acts solely based on its perception of the world through sensors and its own internal information. Teleoperated robots, or robots that are controlled by human operators, are not autonomous and are not studied in this course. Because humans are not directly making decisions for the robot, an autonomous control program must be executed that performs the

[1]Moore's law: en.wikipedia.org/wiki/Moore%27s_law

"cognitive" functions of the robot, specifically: perception, decision making, and executing actions (or motion control). Combined these components act as the brains for the robot and enable the robot to be autonomous.

CS148 is the study of autonomous robotics. That is, given robot hardware with sensing and actuation, we are challenged with programming the robot to autonomously perform a task. The algorithms and architectures for autonomous robot control are studied, which provide the ability to perceive, make decisions and act. It is important to note that this course is not a robotics engineering course. We will not build robot hardware, nor study in detail the physical dynamics and kinematics for expressing motion or robot control theory. CS148 utilizes a relatively cheap robot platform that allows us to study both in theory and in practice the core concepts of autonomous control.

### 1.3.1 Robot Control Loop




The functions of all autonomous robots can be cast into a robot control loop. The robot control loop must be examined under the five main components that constitute an autonomous robot: plant, sensors, perception, decision making, and movement (or motion control.

The physical embodiment of the robot and its dynamics with the world is called the *plant*. The plant, or robot platform, allows a robot to exist in the physical world and interact with other physical objects in its environment. The plant is the physical embodiment and is

subject to the same laws of physics that apply to humans and any other object. The plant consists of the body of the robot, including motors and actuators; it is the material structure that is the robot's presence in the world.

The robot's *sensing* is its observations about the current state of the world. The information the robot collects through its sensors serves as input for perceiving the environment's state. Sensing does not provide a complete view of the world, but is rather partial observations from the robot's perspective.

The *perception* component of the control loop uses the readings gathered from sensing and perceives the information in terms of itself and the environment to estimate, to the best of its abilities, the current state of the world.

Because we would like to have our robots perform some function, every robot must have an "objective" or goal. The overall goal of a robot may have different levels of abstraction and the overall goal may consist of many sub-goals; however only one objective can serve as the input to the control loop during a single iteration. An autonomous robot uses both the perceived state of the world and an object to guide the *decision making* component of the control loop. Decision making determines the next action the robot should take.

Finally, in order to change the state of the physical embodiment, robots must have *motion control*, based on perception and decision making that dictates the next state of the robot in terms of its physical parts. The motion control calculates the control forces that should be applied to the actuators in order for the robot to enact the desired action.

Together the plant and sensors compose the "embodiment" of the robot: the physical body and a mechanism of sensing to gather information about the environment. CS148 does not cover in substance topics within robot engineering for building the embodiment (actuators, sensors, physical parts) of a robot. For robots that are teleoperated or explicitly programmed to perform a specific task without "thinking", our discussion of the robot control loop can end here; our robot would just consist of the embodiment. In contrast, CS148 focuses on the autonomous, or "cognitive", aspects of this loop, pertaining to programming procedures for perception (state estimation), decision making (deciding how to act), and motion control (executing decided actions).

Lectures and activities in CS148 are centered around the basic notion of an autonomous robot control loop, as illustrated in Figure 1.3.1. The algorithms that we study are cast as components in this loop that will be used to form complete project implementations.

## 1.4 Course Format

Topics covered during lectures will be grounded implementations during interactive sessions and 7 assignments. Interactive sessions are devoted to a hands-on introduction to Player/Stage/Gazebo, ROS, Matlab, and working with our iRobot Create platforms leading up to the projects.

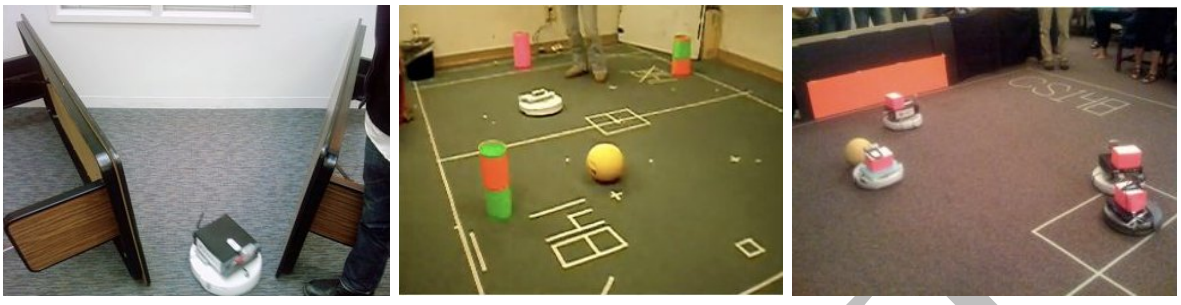


Figure 1.6: Projects: Enclosure Escape, Object Recognition and Multi-robot Coordination.

### 1.4.1 Assignments

The first three assignments are introductions to basic issues defining autonomous robotics and implementing robot controllers using either the Player or ROS robot middleware:

- Ch 4 Create Spotting: run experiments and analyze two built-in behaviors on the iRobot Create bases.
- Ch 5 Enclosure Escape: implement a reactive random traversal robot control policy to escape from an enclosure.
- Ch 6 Object Seeking: implement a control policy to continually seek and visit two objects in alternation.

The remaining projects explore different approaches to autonomous robot control in the context of robot soccer with various constraints on sensing and using heterogeneous robot hardware:

- Ch 7 Path Planning: given a map of the field and an overhead camera view of the environment, develop a deliberative planning-based robot client for visiting specific locations and pushing a ball into a goal.
- Ch 8 Monte Carlo Localization (MCL): same objective as path planning, except the camera view is onboard the moving robot.
- Ch 9 Subsumption: develop a goal scoring controller without maintaining any state (history or timing variables).
- Ch 10 Multi-Robot Coordination: given a team of one SmURV/Create and one unknown robot platform, develop a competitive multi-robot strategy and individual controllers.
- Ch 11 Learning: guide a fixed learning algorithm to play a control policy without explicit programming.

Figure 1.7: Path Planning Project.

### 1.4.2 Robot Soccer

The CS148 projects all center around developing controllers to play robot soccer. Robot soccer has been an active topic of research for over a decade. RoboCup is the worldwide effort to advance the state of the art in robot soccer through yearly competitions. Their goal is to field a team of robot soccer players capable of winning against the human World Cup champion by 2050 (Kitano).

This class considers a constrained version of this problem using the low-cost Brown iRobot Create platform and a highly structured soccer environment. The robots programmed for the course projects will likely not outplay any humans or RoboCup-level robots, but provide

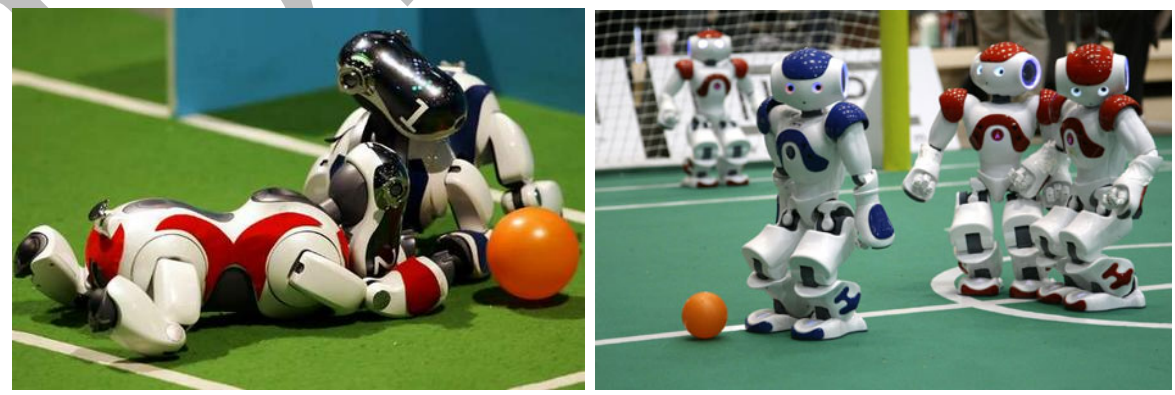


Figure 1.8: 4-legged League (RoboCup 2006); Standard Platform League (RoboCup 2009).


Figure 1.9: Create platform robot soccer.

a fun way to explore the basic topics in autonomous robot control.

## 1.5 Project Deliverables

Projects are graded assignments that consist of an implementation and a writeup. Projects are to be implemented by student teams with individually composed written reports. Implementations are demonstrated during scheduled class periods for grading by the course staff. Project writeups are meant to explain student's work for the project with respect to the scientific method. Due to the nature of robotics and programming in a nondeterministic world that is full of uncertainty, students will undoubtedly face many challenges implementing the projects. Given these challenges, experimentation and writing is essential for validating student's work.

### 1.5.1 Project Writeups

The purpose of the written reports is to familiarize the students with scientific writing and they should be written according to the scientific method. The students must propose a central hypothesis, design experiments based on measurable evidence to test the hypothesis, and confirm the validity or invalidity of the hypothesis based on what was observed. In this regard, they are not necessarily writing the document for themselves or the course staff, but rather to inform someone who is new to their work or the general topic. Science is about general knowledge that is valid without being specific to a given time, place, or technology fads.

## 1.6 How to use this book

This book should be used as a hands-on guide to grounding concepts contained in the robotics textbooks (listed below) into implementable course modules. Figure 1.10 illustrates a Venn categorization of robotics textbooks into, "Intro/Hobby", "Autonomous Robotics", and "Mechatronics". This book, as marked by the "CS148" in Figure 1.10, is meant to combine the accessibility of an "Intro/Hobby" (such as Martin's "Robotic Explorations" or Mataric's "Robotics Primer") with the concepts of an "Autonomous Robotics" textbook (such as "Principles of Robot Motion" by Thrun et al).

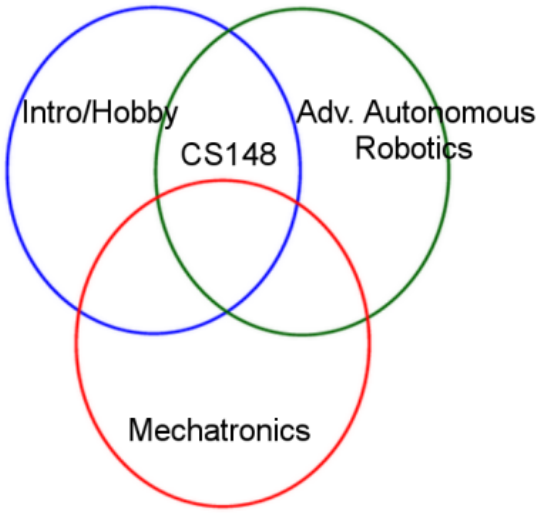



Figure 1.10: Types of robotic books and where CS148 falls within the spectrum.

### 1.6.1 References

- Matarić, *The Robotics Primer*, MIT Press, 2007.
- Wilson, *How to Survive a Robot Uprising*, Bloomsbury USA, 2005.

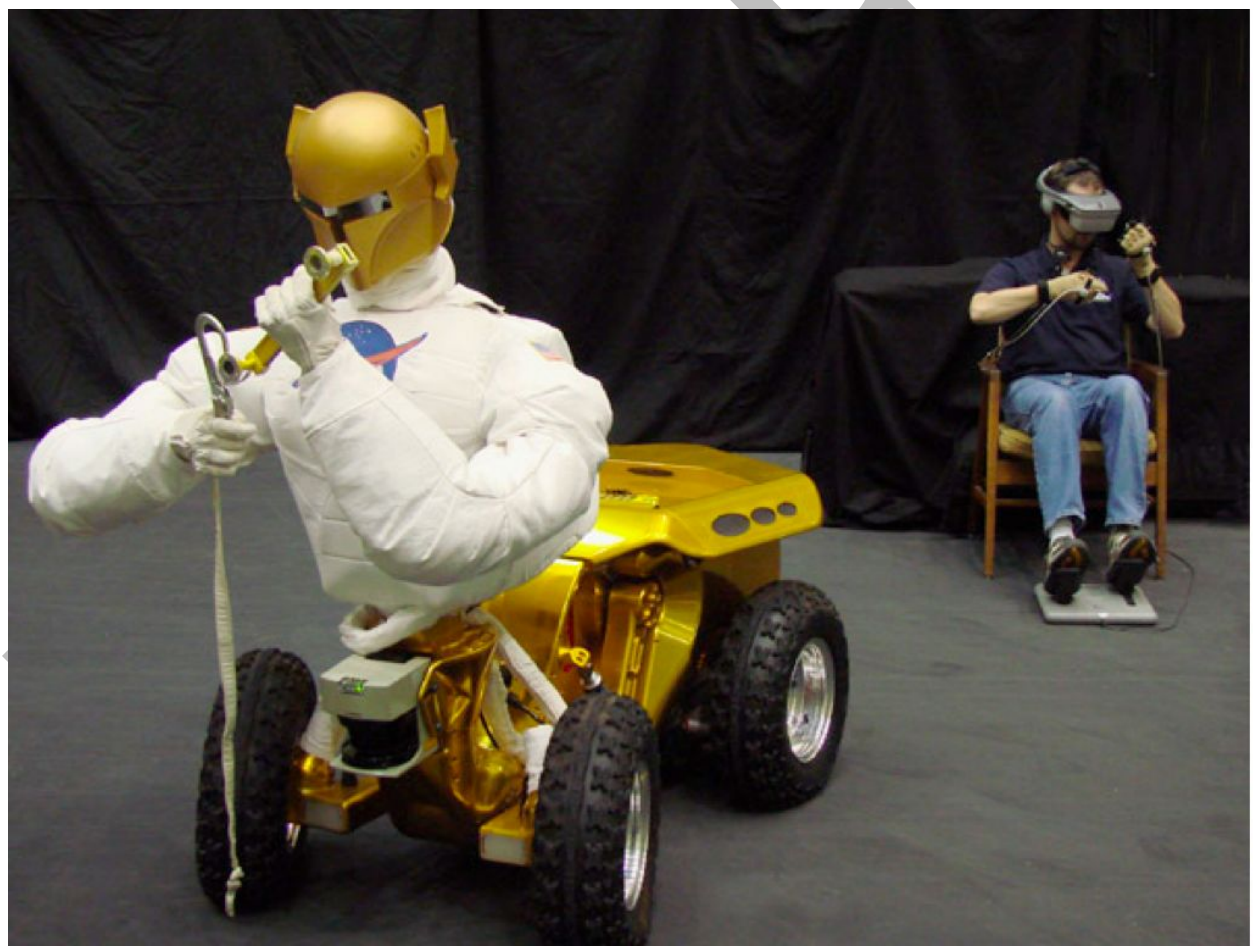

While Wilson's book has a humorous tone, it does cover many of the topics in CS148 in a substantive and accessible manner. The other readings below are optional readings used in class. Substantial material exists on web-accessible resources such as Wikipedia and the AI Topics Library (specifically from the Robots section).

- Siciliano and Khatib, *Springer Handbook of Robotics*, Springer, 2008.
- Arkin, *Behavior-Based Robotics*, MIT Press, Boston, 1998.

- Choset, Lynch, Hutchinson, Kantor, Burgard, Kavraki and Thrun, *Principles of Robot Motion: Theory, Algorithms, and Implementations*, MIT Press, Boston, 2005.
- Thrun, Burgard, and Fox: *Probabilistic Robotics*, The MIT Press, 2005.
- Martin: *Robotic Explorations: A Hands-On Introduction to Engineering*, Prentice-Hall, 2001.
- Spong, Hutchinson, and Vidyasagar, *Robot Modeling and Control*, Wiley, 2005.
- Craig: *Introduction to Robotics: Mechanics and Control (3rd Edition)*, Addison-Wesley, 1989.
- Bekey: *Autonomous Robots: From Biological Inspiration to Implementation and Control*, The MIT Press, 2005.

# Chapter 2

# Getting Started/Teleoperation



In this chapter, we walk through the basic steps for assembling and remotely controlling a low-cost mobile robot, which we call the "SmURV", from "commercial off-the-shelf" (COTS) components. At the completion of this chapter, you will have an assembled SmURV robot platform built with solderless commodity components. You will be able to remote control, or teleoperate, a mobile robot using one of two middleware packages we introduce in the chapter. Instructions for hardware assembly and software installation are specified step-by-step. This chapter refers to The SmURV Robotics Platform, which uses a previous version of the robot platform, and an updated version, Setting up Player 2.1.1: Asus EeePC and iCreate Robot. Please note, all the technology, instructions and hardware costs are as of Spring 2010.

The primary purpose of the SmURV (**Sm**all **U**niversal **R**obotics **V**ehicle) is to be a robot platform that emphasizes the computational aspects of robotics without requiring engineering expertise or low-level hardware hacking. That is, no soldering or low-level assembly programming is necessary to build a SmURV. However, we do assume the reader has user-level knowledge of Unix (e.g., Ubuntu Linux, OS X) and super-user abilities to install software packages, etc.

The SmURV platform, as seen on the previous page, is based on the iRobot Create mobile base with a sub-notebook computer, such as the Asus EEE PC, and a webcam-style camera. All of the hardware components for the SmURV can be readily purchased from online or brick-and-mortar stores, such as Target. The computer typically runs some distribution of Linux (Ubuntu in our case) with a robot middleware package to mediate communication between a client application and the the Create/USB camera hardware. We describe how to install robot middleware packages for Player and the Robot Operating System (ROS), although you will only need one. Robot middleware is discussed in detail in Chapter 3.

At the end of this chapter you will be able to:

- Assemble a low-cost mobile robot using COTS components.
- Install all the necessary software for running a robot client.
- Teleoperate a mobile robot.

## 2.1 Hardware Components

### 2.1.1 iRobot Create

The iRobot Create, produced by iRobot, is used as the basic robot unit. It is a mobile robot that is similar to the iRobot Roomba, except it does not contain a vacuum and is instead designed for robotics development. The Create is equipped with a serial port; the Open Interface Specification (OIS) for the Create explains what commands can be sent to the Create over the serial port in order to read sensor data and send motor commands to the robot. However, we do not teach students how to write programs by sending low level commands to the robot. Instead we utilize robot middleware (as explained in Chapter 3), as a hardware/software abstraction for transparency and portability.


### 2.1.2 Accessories for the iRobot Create

- Serial Cable that is packaged with the Create.
- Serial to USB adapter cord. CS148 uses FTDI US232R-10 "evaluation cables" which can be ordered from DigiKey for roughly USD $20. Note: this cord needs to be of sufficient quality; cheaper adapters will overheat and cause problems.
- (recommended) Rechargeable Advanced Power System (APS) Battery.
- (optional) iRobot Virtual Wall which creates an invisible barrier that the Create will not cross by emitting infrared signals that the Create detects with its IR receiver.
- (optional) Self Charging Home Base which enables the Create to automatically charge its iRobot rechargeable battery.

### 2.1.3 Asus EeePC subnotebook

The Asus EeePC subnotebook is the computational brain for this robot platform. The EeePC communicates with the Create base through a USB-Serial connection. Programs running on the EeePC will control the robot's actuators and receive sensory information. Thus the programmer does not have to worry about sending low level commands to the Create's serial port. We recommend the 7" EeePC model since larger EeePCs will stick out over the Create base.

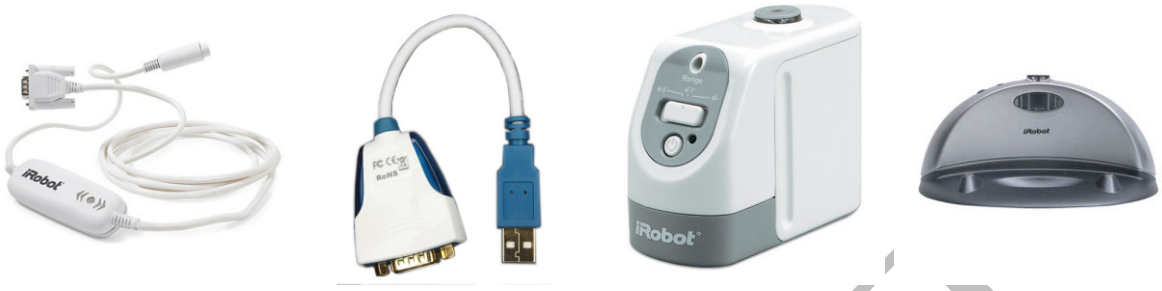



Figure 2.1: iRobot accessories include a serial cable, a serial to USB adapter, a virtual wall and a self charging home base.

### 2.1.4 USB Camera

Just having the bump, cliff and IR sensors built into the Create is not necessarily a lot of fun. Adding a camera to the robot however is! You can either go with a cheap USB Webcam or a more expensive Firewire camera if having uncompressed high quality images is a requirement. CS148 previously utilized a Unibrain Fire-I camera. However with our newest platform we recommend a USB webcam, preferably a newer, Video4Linux2 (V4L2) version.

### 2.1.5 Assembling the Robot Platform

1. Connect the Create's serial cable to the USB to serial adapter. Connect this to the Create and the EeePC, as seen in Figure 2.2, left image.

2. Connect the USB webcam to the EeePC.

3. Tuck the cords into the Create and set the EeePC on top. The assembled Create platform is seen in Figure 2.2, right image.

4. Optionally, add Velcro tape to secure the EeePC to the top of the Create and to secure the camera to the EeePC.


Figure 2.2: Connected Create and EeePC; Assembled Create Robot Platform.

### 2.1.6 Costs

Costs are as of Spring 2010 and are specified in USD.

| Part | Name | Cost |
|---|---|---|
| Robot (Basic) | iRobot Create Programmable Robot | 129.99 |
| Robot (Package) | iRobot Create Premium Development Package; includes APS, 2 virtual walls, home base, remote and command module | 299.99 |
| Subnotebook | Asus EeePC 7" 2G Surf | 240.00 |
| USB Camera | Logitech Webcam Pro 9000 | 99.99 |
| Serial to USB adapter | FTDI US232R-10 "evaluation cables" | 22.00 |

## 2.2 Software Components

### 2.2.1 Operating System

The EeePC comes with a GNU Linux OS, however we recommend installing Easy Peasy (formerly Ubuntu-Eee), which is a custom version of Ubuntu designed especially for the EeePC. The EeePC comes with a version of Xandros Linux which has Asus-modified libraries that are incompatible with the Player middleware system (see Section 2.2.2), thus it is recommended to simply install a more standard OS. Easy Peasy was chosen because it is one of the more supported custom Eee Linux distributions available and it is one of the few that fits on the EeePC 2G Surf model. Note: other distributions are available and outlined on Installing Operating Systems, but the rest of the instructions are assuming an Easy Peasy installation.

### Install Easy Peasy

1. You will need:
   - Another computer to download Easy Peasy on and to run UNetbootin (Universal Netboot Installer) for creating a bootable USB flash drive. This computer must be either Windows or Ubuntu/Linux. If using Linux, you will need administrative rights to install the UNetbootin application.
   - An empty USB stick (of 1GB or more, formatted to FAT32) to do the initial Ubuntu boot and installation.
2. From the secondary computer, go to the Easy Peasy download site and click on "Download".
3. Select the "Download" link to download Easy Peasy and choose "Save File" (Figure 2.3).

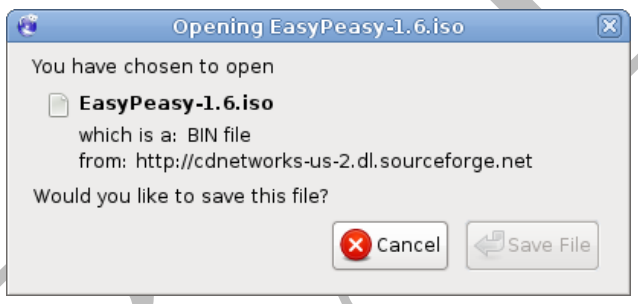



Figure 2.3: Easy Peasy download screenshot.

4. Download UNetbootin, a helper application used to move Easy Peasy to a USB stick. On the UNetbootin site, select either "Download (for Windows)" or "Download (for Linux)" depending on your operating system. Select "Save File" (Figure 2.4). This step of creating a bootable drive is necessary since the EeePC does not have a CD-ROM drive.

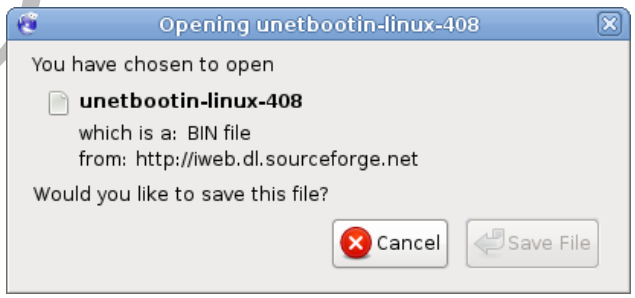



Figure 2.4: UNetbootin download screenshot.

5. Make sure the USB stick is formatted to FAT32, otherwise the installation of Easy Peasy on the EeePC will hang and be unsuccessful. Note: formatting the stick will cause all data to be lost.

   - To format the USB stick in Windows, insert the stick and right click on the USB icon. Press "Format" in the menu. Select "FAT32" in the File system drop-down list and press "Start" (Figure 2.5).

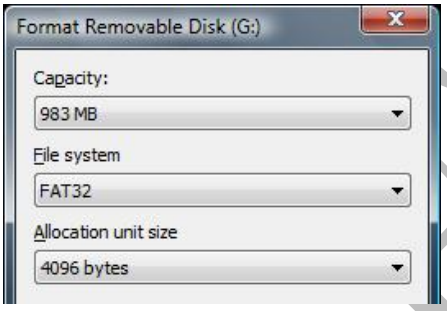



Figure 2.5: Formatting to FAT32 in Windows.

   - To format the USB stick in Linux, root permissions are necessary. Insert the USB stick.

     (a) Determine if the USB is already formatted to FAT32. Run:
     `sudo fdisk -l`

```
obot@brainiac:~$ sudo fdisk -l

Disk /dev/sda: 16.1 GB, 16139354112 bytes
255 heads, 63 sectors/track, 1962 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x000ea510

   Device Boot      Start         End      Blocks   Id  System
/dev/sda1   *           1        1875    15052800   83  Linux
/dev/sda2            1875        1963      705537    5  Extended
/dev/sda5            1875        1963      705536   82  Linux swap / Solaris

   Device Boot      Start         End      Blocks   Id  System
/dev/sdc1               1        3936     1007600    b  W95 FAT32
obot@brainiac:~$
```

Figure 2.6: List USB device and current formatting.

     If FAT32 is listed under "System" (Figure 2.6), you do not need to reformat. If anything else is listed, continue.

     (b) Determine which device (sda, sdb, sdc, etc) Linux assigned the USB drive to. Running the above `fdisk` command prints this information under "Device" (Figure 2.6). In our case the device assignment is to "/dev/sdc1".

(c) Run fdisk, a partition table manipulator:
`sudo fdisk /dev/sdX`
`sdX` should be the name of the device as determined in the previous step.

(d) Enter "d" to delete all existing partitions.

(e) Enter "n" to create a new partition.
Enter "p" for the command action.
Enter "1" for the partition number.
Hit Enter twice to keep default settings.

(f) Enter "t" to impose the partition's system type as FAT32.
Enter "b" as the partition type hex code, which is the id for type FAT32.

(g) Enter "w" to save changes and write the partition table to the USB stick.

```
obot@brainiac:~$ sudo fdisk /dev/sdc

WARNING: DOS-compatible mode is deprecated. It's strongly recommended to
         switch off the mode (command 'c') and change display units to
         sectors (command 'u').

Command (m for help): d
Selected partition 1

Command (m for help): n
Command action
   e   extended
   p   primary partition (1-4)
p
Partition number (1-4): 1
First cylinder (1-3936, default 1):
Using default value 1
Last cylinder, +cylinders or +size{K,M,G} (1-3936, default 3936):
Using default value 3936

Command (m for help): t
Selected partition 1
Hex code (type L to list codes): b
Changed system type of partition 1 to b (W95 FAT32)

Command (m for help): w
The partition table has been altered!
```

Figure 2.7: Steps (c) - (g).

(h) Unmount the device: `sudo umount /dev/sdX1`

(i) Format the drive to FAT32:
`sudo mkfs.vfat -F 32 /dev/sdX1`
This step is necessary in order to format the newly created partition.

(j) Unplug and insert the USB stick into the computer again.

6. Install UNetbootin:[1]

- Run the UNetbootin executable.
  - If using Windows, click on the UNetbootin file to launch the application (Figure 2.9).

[1]The following are the same instructions as under "Installation & Screenshots" on the Unetbootin site

```
obot@brainiac:~$ sudo umount /dev/sdc1
obot@brainiac:~$ sudo mkfs.vfat -F 32 /dev/sdc1
mkfs.vfat 3.0.7 (24 Dec 2009)
obot@brainiac:~$ sudo fdisk -l

   Device Boot      Start         End      Blocks   Id  System
/dev/sdc1               1        3936     1007600    b  W95 FAT32
```

Figure 2.8: Steps (h) - (i).

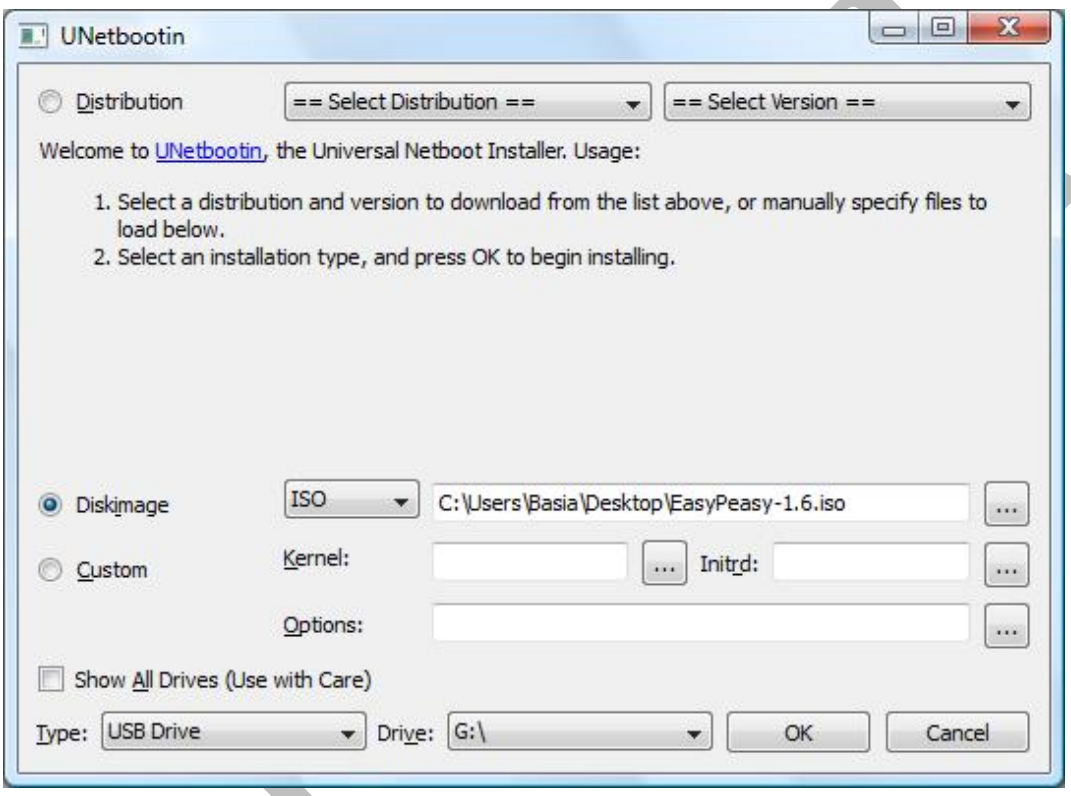



Figure 2.9: UNetbootin installation prompt for Windows.

– If using Linux:

* Install Unetbootin dependencies:
  `sudo aptitude install mtools p7zip-full`
* Make the file executable:
  `chmod +x unetbootin-linux-*`
* Run the application (Figure 2.10):
  `sudo ./unetbootin-linux-*`

- Select Distribution/Select Version may be ignored. The Disk Image format should be "ISO"; enter "EasyPeasy-*.iso" (the downloaded Easy Peasy file) as the Disk Image. This specifies which file to create the bootable drive from. Finally select

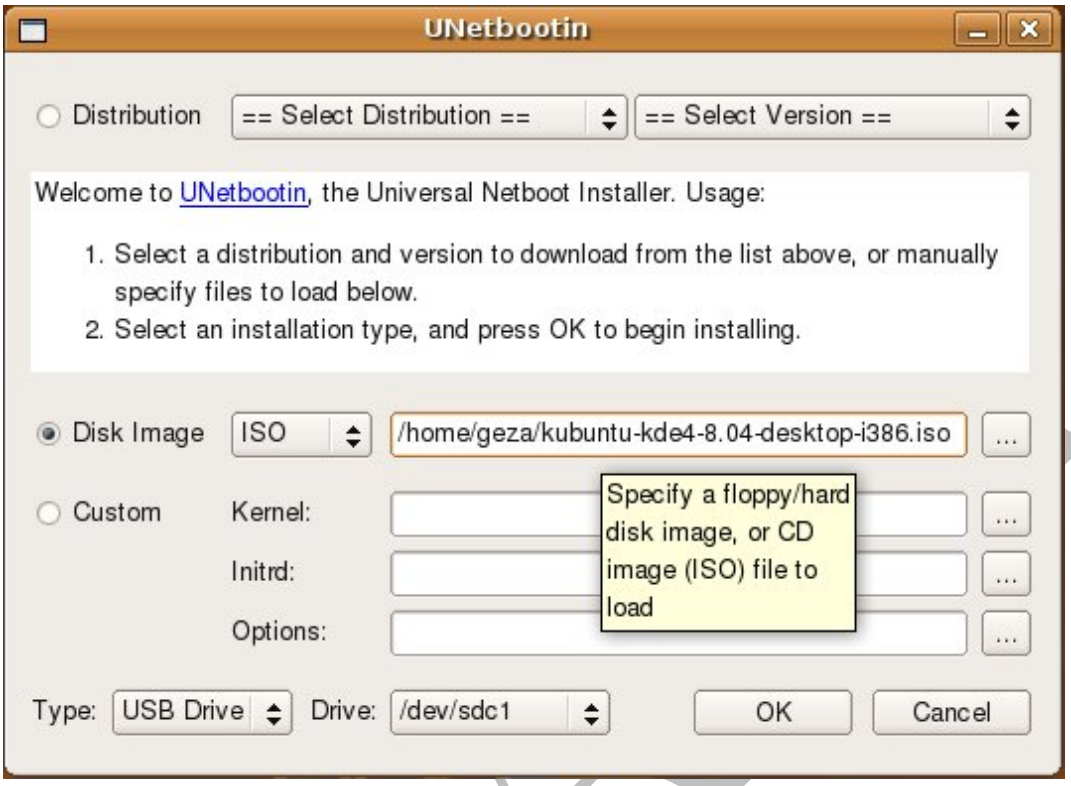



Figure 2.10: UNetbootin installation prompt for Linux.

"USB Drive" as the Type and specify the target Drive. Press "OK".

- If using Windows, refer to Figure 2.9.
- If using Linux, refer to Figure 2.10. By default, the correct USB drive should be automatically selected. To be certain, the instructions to find the name of the USB drive are outlined in step 5b.

- After installation is complete (prompted by the screenshot in Figure 2.12), do not reboot and instead select "Exit" (unless you are installing Easy Peasy on the same EeePC from which you create the bootable USB drive). Unmount the disk. Easy Peasy is now installed on the USB stick.

7. Insert the USB stick into the EeePC[2]

8. (Re)start the EeePC. Press ESC a few times while the Asus boot screen is displayed.

9. Select the USB drive from the list of bootable devices in the boot menu.

10. Select "Default" from the UNetbootin menu.

[2]It is suggested to use the USB port on the left side of the EeePC; some EeePC's seem unable to boot from the USB ports on the right side.

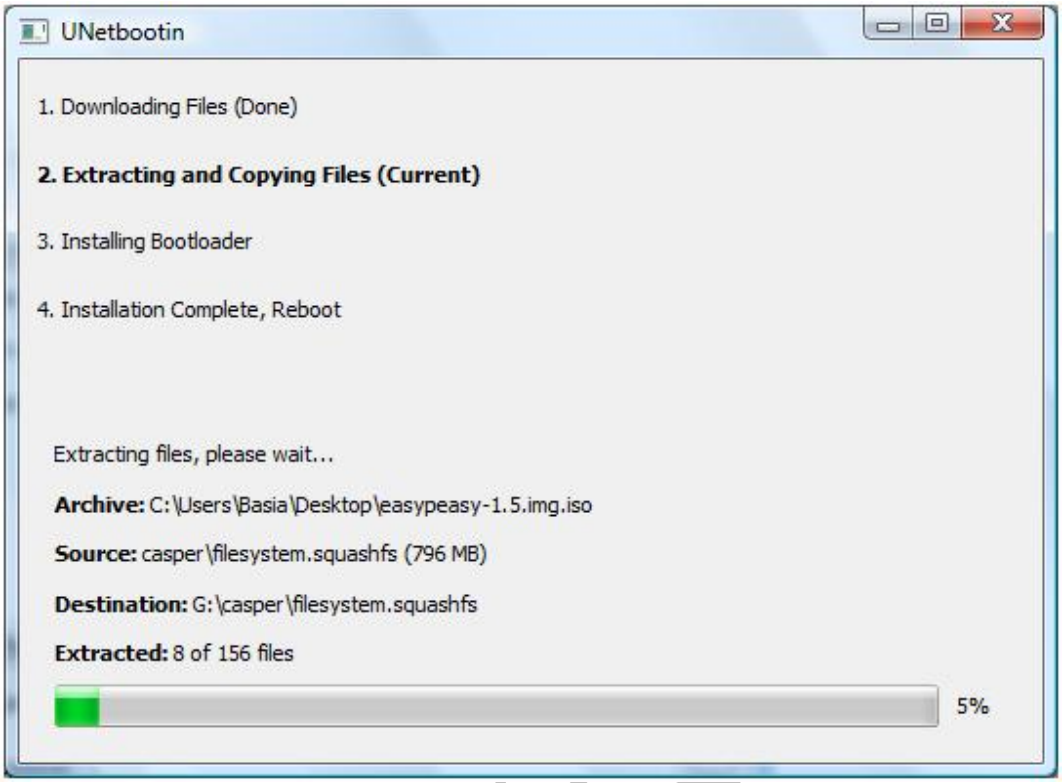



Figure 2.11: UNetbootin installation screenshot.

11. Install Easy Peasy by selecting the "Install EasyPeasy 1.*" icon, which should be in the Desktop folder.

12. You will be prompted for your language preference, time zone, etc. Follow all the installation instructions.

13. If your EeePC is the 2G Surf model, be aware that there is only 2GB of hard drive and the OS takes up 1.8GB when you install. Tips to prevent wasted time reinstalling:

    - Choose "No Localization" on the first install screen.
    - Do a manual install, with an ext2 format and no swap drive.
    - Once installed, the 2GB internal drive will be nearly full. To free more than 300mB, uninstall Open Office, Skype and other unnecessary applications using the following command:
      `sudo apt-get remove <package_name>`
      where `<package_name>` is any package, such as `openoffice.org*`. Note, to list all installed packages: `dpkg --list`

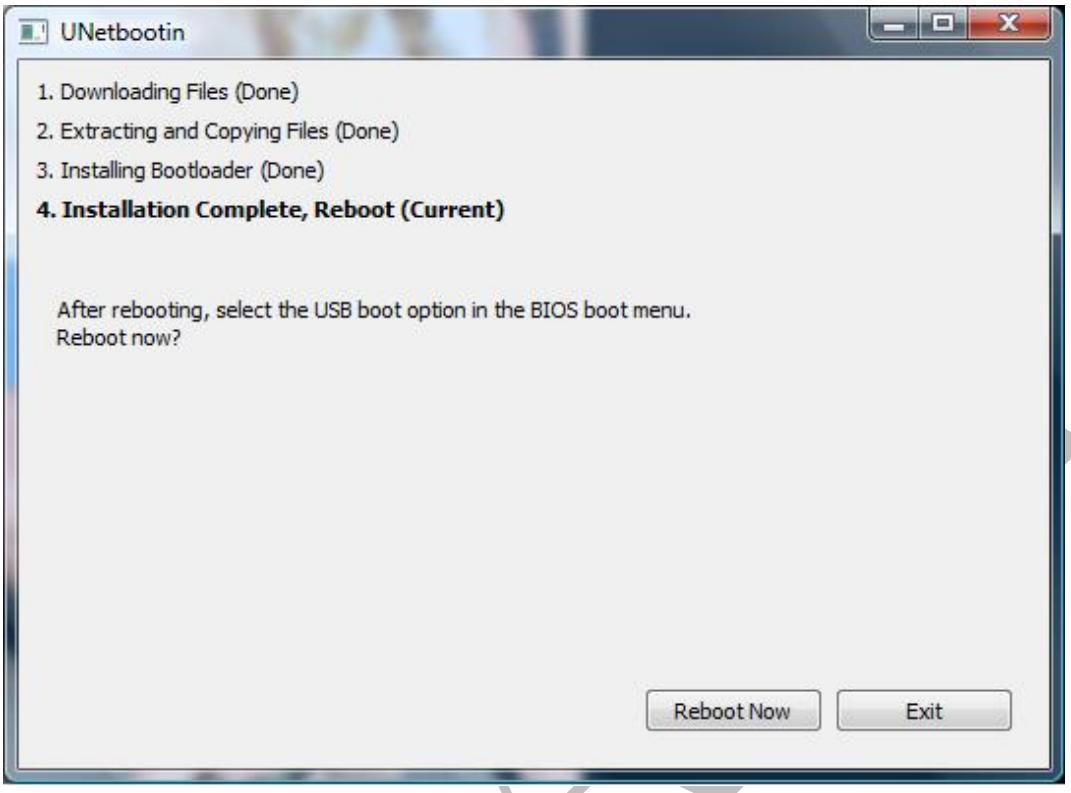



Figure 2.12: UNetbootin installation complete screenshot.

14. (Optional) Enable the Regular Desktop mode instead of the Netbook Interface mode by following these instructions.

15. Connect the EeePC to the internet (necessary for the next step).

16. Enable all of the repositories in the `/etc/apt/sources.list` file. Ubuntu uses APT for package management; `/etc/apt/sources.list` contains a list of sources from which software repositories can be obtained.

    - Backup the configuration file:
      `sudo cp /etc/apt/sources.list /etc/apt/sources.list.backup`
    - Open the file for editing: `sudo nano /etc/apt/sources.list`
    - Enable all software repositories by uncommenting any commented-out apt repository lines (lines which begin with either "deb" or "deb-src"). The following are two apt lines:
      `deb http://us.archive.ubuntu.com/ubuntu/ lucid main restricted`
      `deb-src http://us.archive.ubuntu.com/ubuntu/ lucid main restricted`
    - Obtain the updated package lists from the newly enabled repositories:
      `sudo apt-get update`

If you get the error "Failed to fetch...", make sure the EeePC is connected to the Internet!

- If you are outside the US and receive a "Failed to fetch..." error, try changing all references from "http://us.archive.ubuntu.com" to "http://archive.ubuntu.com" in the `sources.list` file.

17. Install others software packages that will be utilized:
`sudo apt-get install cmake subversion wget python g++`
If using only Player, none of these are essential for teleoperation, but cmake and subversion are at least highly recommended. If using ROS, all four of these are necessary.

### 2.2.2 Player 2.1.3

#### Install Player 2.1.3

After installing Easy Peasy, the EeePC is now ready to install Player.

1. Download `player-2.1.3.tar.gz` source tarball from Player Sourceforge Site.

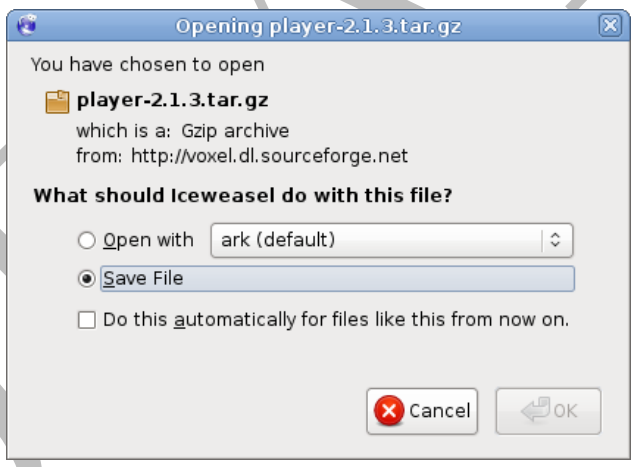



Figure 2.13: Download Player 2.1.3 screenshot.

2. Uncompress and expand the downloaded file: `tar xzvf player-2.1.3.tar.gz`

3. Install additional libraries: `sudo apt-get install <library_name>`
Replace `<library_name>` with each of the following libraries:
libgdk-pixbuf2, libgtk2.0-dev, libjpeg62-dev.
`apt-get` should have all of these libraries if every repository in `/etc/apt/sources.list` has been enabled (refer to step 16 of Section 2.2.1).

   - If running `apt-get` gives the error: "Couldn't find package libgdk-pixbuf2":

  - Add the following lines to the `/etc/apt/sources.list` file:
    ```
    deb http://us.archive.ubuntu.com/ubuntu/ jaunty universe
    deb-src http://us.archive.ubuntu.com/ubuntu/ jaunty universe
    deb http://us.archive.ubuntu.com/ubuntu/ jaunty-updates universe
    deb-src http://us.archive.ubuntu.com/ubuntu/ jaunty-updates universe
    ```
  - Run: `sudo apt-get update`

- If you are using a fireware camera and not a USB webcam, you should also `apt-get install` the following libraries:
  libraw1394-dev, libavc1394, libdc1394-13, libdc1394-13-dev.

4. Change to the Player source directory: `cd player-2.1.3`
5. Configure Player: `./configure`
   Note: Player will install without throwing any errors regardless if libraries are missing. If the above mentioned libraries were not installed, Player will still install but either Player or accessing camera data may not work.
6. To enable Player to have more functionality beyond using the Create, the sensor drivers, a USB or camera, the blobfinder proxy, and playercam/playerjoy, it may be necessary to install more libraries. The libraries needed are located in the `player-2.1.3/config.log` file, which was created by configure. `config.log` is a huge file that contains all the messages created by configure, including missing packages, library dependencies and a list of what Player has and has not installed.
7. Compile Player: `make`.
8. Install Player: `sudo make install`.
9. Restart the EeePC.
10. Reopen the terminal and run Player: `player`
    - The output should be similar to Figure 2.14.
    - If the following message is displayed:

      player: error while loading shared libraries: libplayerdrivers.so.2.2: cannot open shared object file: No such file or directory

      Run `ldconfig` to fix the error. Player sometimes does not link its libraries completely when installing.
    - If the "Registering driver" message outputs from Player, installation is successful and the EeePC is now ready to be hooked up to the robot.

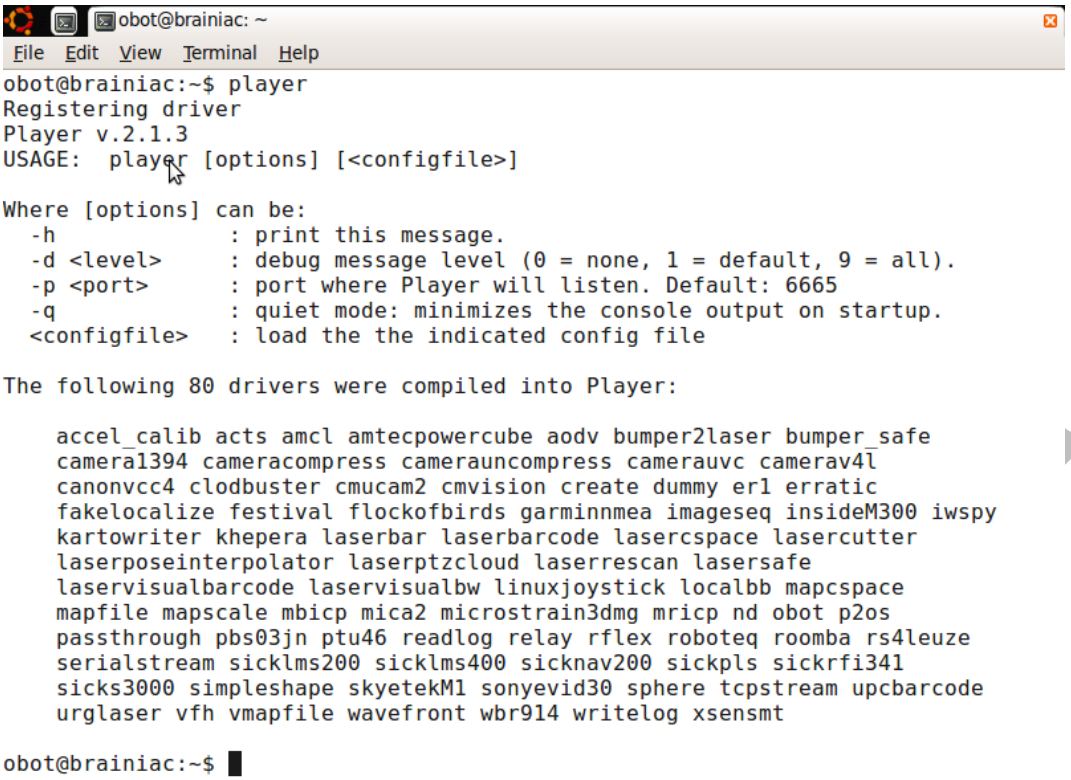



Figure 2.14: Output from starting Player (Step 10 in "Install Player 2.1.3" instructions).

### Run and Test Player on the Create

To test the robot's vision and to teleoperate the Create, in this section we will be running two Player utilities that are included with the Player distribution: playercam and playerjoy. playercam is a GUI client that visualizes the images captured by the webcamera. playerjoy provides mobile control of the robot. The playerjoy client uses input from the keyboard to drive the robot around; only a forward and rotate speed can be manipulated on the Create.

1. You will need:
   - The assembled Create Robot Platform.
   - The EeePC installed and working with Player.
   - Ideally (but not necessary to test) a wireless network and another computer with Player installed.
2. Create a configuration file, which is required by the Player server to tell Player what robot systems to load drivers for. Configuration files are text files that have a `.cfg` extension. Below is `create.cfg`, a basic configuration file for the Create with position and bumper proxies specified along with camera/blobfinder drivers. The following

configuration file assumes the robot is attached to the "/dev/ttyUSB0" port. If the robot is actually attached to a different port, reflect this change in the configuration file.

```
driver
(
        name "create"
        provides ["position2d:0" "bumper:0" "power:0" "ir:0"]
        port "/dev/ttyUSB0"
)

driver
(
        name "camerauvc"
        provides ["camera:0"]
)

driver
(
        name "cmvision"
        provides ["blobfinder:0"]
        requires ["camera:0"]
        colorfile "colors.txt"
)
```

You will pass the name of this configuration file as a command line argument when starting the Player server. Please refer to Section 3.1.3 for an explanation of Player configuration files and the specific format.

Note: if your EeePC has a built-in webcam, you will have to specify the external camera in the config file.

- Under the line: provides ["camera:0"] add the line: port "/dev/video0".
- Check whether your external cam is video0 or video1 by listing the contents of the `/dev` directory with the camera plugged in and unplug the camera to see which listing disappears.

3. Create a color calibration file for the blobfinder proxy. This color file is specified as `colors.txt` in the above configuration file, thus this file must reside in the same directory as `create.cfg`. Below is a generic uncalibrated color file.

```
[Colors]
(255,  0,  0) 0.000000 10 Red
```

```
(  0,255,  0) 0.000000 10 Green
(  0,  0,255) 0.000000 10 Blue

[Thresholds]
( 25:164, 80:120,150:240)
( 20:220, 50:120, 40:115)
( 15:190,145:255, 40:120)
```

The following is a sample color file that has been calibrated for four different colors.

```
[Colors]
(255,  36,  0) 0.000000 10 Orange
(  0,255,  0) 0.000000 10 Green
(255,105, 180) 0.000000 10 Pink
(255, 255,0) 0.000000 10 Yellow

[Thresholds]
( 100:135, 101:115, 170:214)
( 20:220, 50:120, 40:115)
( 99:132, 123:131, 168:207)
( 124:174, 90:105, 137:148)
```

Note, in a color calibration file, there must be a line space in between the Colors and the Thresholds sections. Details on what the numbers in the color file represent and how to calibrate for colors will be explained in Chapter 6.

4. Put a charged battery in the Create base, turn on both the Create and the EeePC. The Led lights on the Create will blink and eventually turn off; even if the lights stop blinking the Create may still be turned on.

5. Start the Player server. Open a terminal and `cd` to the directory where your configuration file is and type: `player create.cfg`
   If all is successful you should get a similar message displayed in Figure 2.15.

   The Player server is now running on your Create Robot.

6. Run playercam. In a second terminal on the EeePC, run: `playercam`
   Running this command will open up a window on the screen with the camera's output, as seen in Figure 2.16. If the EeePC is on a wireless network and you want to run playercam from another computer on the network, run:
   `playercam -h <robot IP> -p 6665`
   `<robot IP>` is the IP address of the EeePC. The port number could be different that `6665` if you have changed from the default.

```
obot@brainiac: ~
File  Edit  View  Terminal  Help
obot@brainiac:~$ player create.cfg
Registering driver
Player v.2.1.3

* Part of the Player/Stage/Gazebo Project [http://playerstage.sourceforge.net].
* Copyright (C) 2000 - 2006 Brian Gerkey, Richard Vaughan, Andrew Howard,
* Nate Koenig, and contributors. Released under the GNU General Public License.
* Player comes with ABSOLUTELY NO WARRANTY.  This is free software, and you
* are welcome to redistribute it under certain conditions; see COPYING
* for details.

Opening connection to Create on /dev/ttyUSB0...Done.
listening on 6665
Listening on ports: 6665
```

Figure 2.15: Output from running the Player server with a config file (Step 5 in "Test and Run Player" instructions).

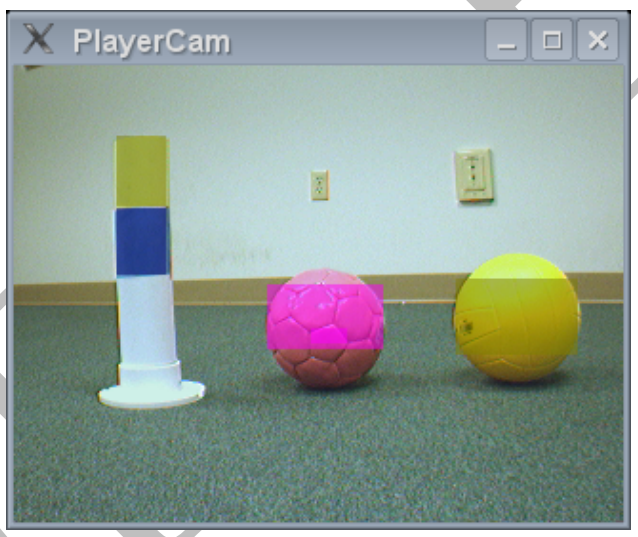



Figure 2.16: Playercam output.

If there are solid color rectangles over regions of the screen, the Player blobfinder works and is successfully detecting colors.

7. Run playerjoy to move the robot using keyboard commands. Preferably from a client computer on the network, run: `playerjoy <robot IP>:6665`. If running locally, just run: `playerjoy`. The terminal window output from running playerjoy is displayed in Figure 2.17.

   Possible issues:

   - If the robot does not move, make sure the Create base has not turned off and that the terminal with playerjoy running is selected/on top.

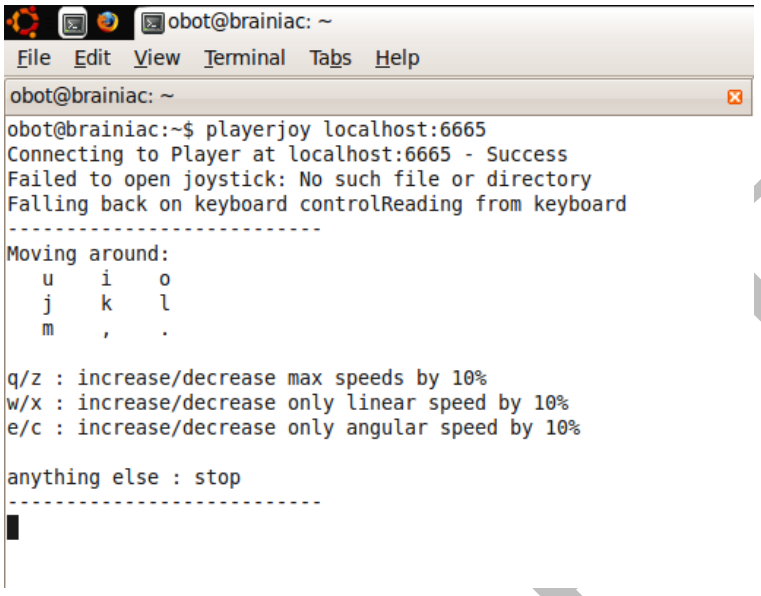



Figure 2.17: Output from running playerjoy.

- If the Create does turn off, both the player server and the client application (playerjoy or playercam) must be restarted.

### 2.2.3 ROS

The ROS website has installation instructions for Ubuntu that we will follow in addition to installing Brown's ROS packages. Brown University has a repository for ROS packages at brown-ros-pkg. Please check this site under the "New Users" section for the latest instructions and updates on the software resources available.

#### ROS Setup

- During the installation of Easy Peasy, make sure wget, Python, CMake and Subversion are all installed on the EeePC. If not, for any missing package needed:
  `sudo apt-get install cmake subversion wget python`

- Setup sources.list:
  `sudo sh -c 'echo "deb http://code.ros.org/packages/ros/ubuntu karmic main" > /etc/apt/sources.list.d/ros-latest.list'`

- Set up your keys:
  `wget http://code.ros.org/packages/ros.key -O - | sudo apt-key add -`

- Run: `sudo apt-get update`

- Install: `sudo apt-get install ros-boxturtle-base`
  Alternatively, for the latest release, run:
  `sudo apt-get install ros-latest-base`

- Brown has a ROS install script available which retrieves versions of the ROS source, tested to be compatible with brown's packages, and automatically retrieves the latest brown packages. To retrieve the install script, on the EeePC run:
  `wget http://brown-ros-pkg.googlecode.com/svn/tags/getros/getros.py`

- Run the script to install: `python getros.py`

- To setup the correct environment variables, from the ROS root directory run:
  `source rosenv`
  You must do this everytime you open up a new terminal.

- Build the basic ROS packages with: `rosmake roslite`
  Each brown-ros-pkg package can be build with appropriate calls to rosmake, such as: `rosmake cv_capture`. Note, the `rosmake` command can be run from any directory within the ROS directory structure.

**Run and Test the Brown ROS Create Driver**

- The Brown ROS Create Driver is located in the "ros-1.0.0/pkg/irobot_create_2_1" directory.

- python-serial is the only external dependency of the driver and must be installed.
  - Go to pyserial to download.
  - Unpack the archive and install the package:
    ```
    tar zxvf pyserial-*.tar.gz
    cd pyserial-*
    python setup.py install
    ```

- Build: `rosmake irobot_create_2_1`

- In its own terminal run: `roscore`
  This command launches the ROS Master, which must be running for other ROS nodes to locate each other and communicate.

- The ROS Create node assumes the robot is attached to "/dev/ttyUSB0". If it is not, run: `rosparam set /brown/irobot_create_2_1/port PORT`
  PORT should be the port to which the robot is actually attached to.

- Run the create driver: `rosrun irobot_create_2_1 driver.py`
  This command runs the Create driver from the Brown ROS package. The purpose of

this node is to provide other nodes with basic sensor information (e.g. whether or not a bumper has been pressed), as well as to provide basic movement control to other nodes for the Create.

- Teleoperate the Create using the "teleop_twist_keyboard" package.
  - Compile: `rosmake teleop_twist_keyboard`
  - Run: `rosrun teleop_twist_keyboard teleop_twist_keyboard.py` This teleoperates the robot using keyboard commands (Figure 2.18).

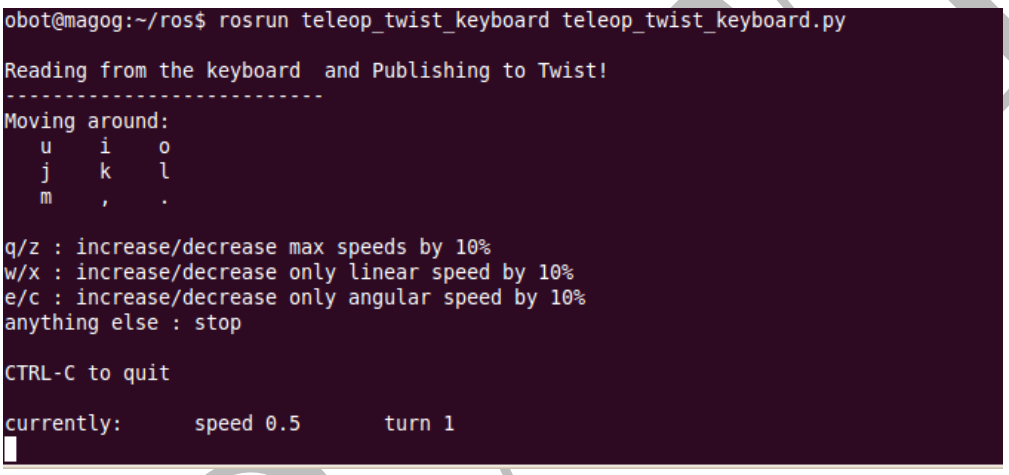



Figure 2.18: Teleoperating the Create using ROS

# Chapter 3

# Robot Middleware and Controller Development

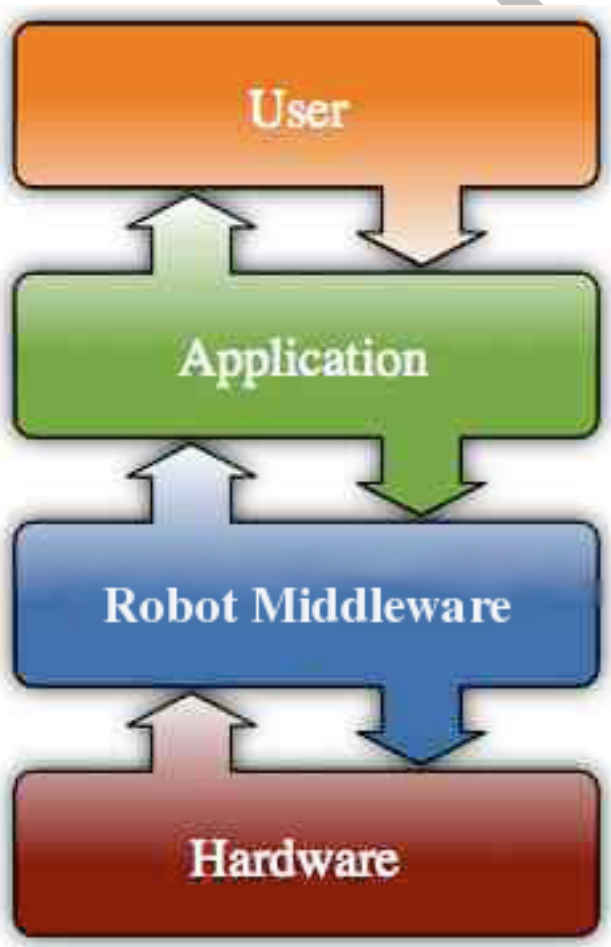



Figure 3.1: Middleware layers.


Given a robot embodiment and a program that controls the robot, middleware serves as an abstraction between the hardware and the robot client that allows for writing robot controllers not dependent on any specific platform, as seen in Figure 3[1]. The need for robot middleware arises out of the question of how to program our robots to perform desired tasks. The straightforward approach is to write and compile a program that directly sends low level commands to the robot base in order to perform the robot's "cognitive" functions. This one-off solution may be sufficient for programming the robot to perform one specific task on top of one specific robot platform. However, the following challenges arise from applying this approach to robotic application development:

- Portability to other robot platforms or new devices.
- Scalability to growth in functionality.
- Reusability for other projects.
- Interoperability between functions and languages.

To enhance the application development for robots, middleware services exist to provide an abstraction layer between computation and embodiment, similar to the hardware abstraction layer in operating systems. Middleware allows any robot controller to run on any robot platform, thus the controller software is not specific to the hardware. Between the hardware at the lowest layer and a client program at the highest layer is the middleware, which provides an interface to the robot's devices and marshals the high level commands of the client program to the low level commands understandable by the robot hardware. Middleware aims to overcome the above listed challenges by simplifying the development process, enhancing software integration and reuse, supporting interoperability, providing efficient utilization of robot components and resources, and providing interfaces that hide the low-level hardware complexity.

Although the Create platforms we are using in CS148 are relatively simple since we only need access to a handful of sensors and actuators, many robotic applications today are complex assemblages, consisting of many heterogeneous hardware components build by different manufacturers, and multiple software modules written in different languages. Given such a platform, the need for middleware is essential, allowing for modular design while coordinating the communication among the components efficiently and providing interoperability, portability and robustness.

[1]Source: Golftheman, http://en.wikipedia.org/wiki/File:Operating_system_placement.svg

There are several existing middleware packages for robotics, such as LCM, Orocos, YARP, CARMEN, Microsoft Robotics Studio and JAUS. This course focuses on two: Player and the Robot Operating System, although there is mention of other middleware systems as well.

At the end of this chapter you will be able to:

- Understand the Player and/or ROS robot middleware system as an abstraction between the robot hardware and software.
- Understand the basic concepts of a handful of other middleware systems.
- Know how to send commands to and receive sensory information from the robot using either Player or ROS.
- Write a basic robot client application to control the robot.

## 3.1 Player/Stage/Gazebo

PSG is a robot interface and simulation software suite that expose drivers through a simple interface for robotic applications. The core of PSG is the Player robot interface, which is a framework for robot control consisting of devices, robot servers and robot clients.

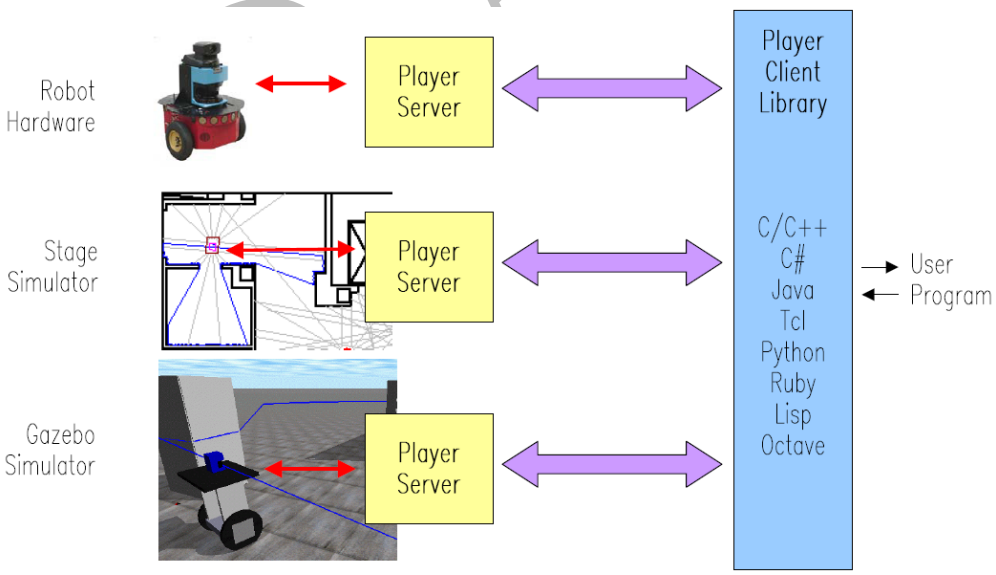



Figure 3.2: Player middleware abstraction.

Running on your robot, Player provides a clean and simple interface to the robot's sensors and actuators over a TCP/IP (or UDP/IP) network. Client programs talk to Player over a TCP socket, read data from sensors, write commands to actuators, and configure devices

on the fly. Figure 3.1[2] illustrates the Player middleware abstraction between hardware and client applications.

One of the major advantages of Player as a robot server is its independence from a particular client-development language. The interaction between Player and a client program is done completely over a TCP/IP (or UDP/IP) network connection. The communication between the Player client and Player server are illustrated in Figure 3.3[3]. Thus, any language with libraries that supports Player functionalities can be used to develop robot clients. The best-supported client languages are C and C++ (supported through libplayerc and libplayerc++). Client programs can run on any machine that has a network connection to your robot and can be written in any language that supports TCP sockets. Additionally Player is designed to be platform independent; it supports a variety of robot hardware and Player's modular architecture makes it easy to add support for new hardware.

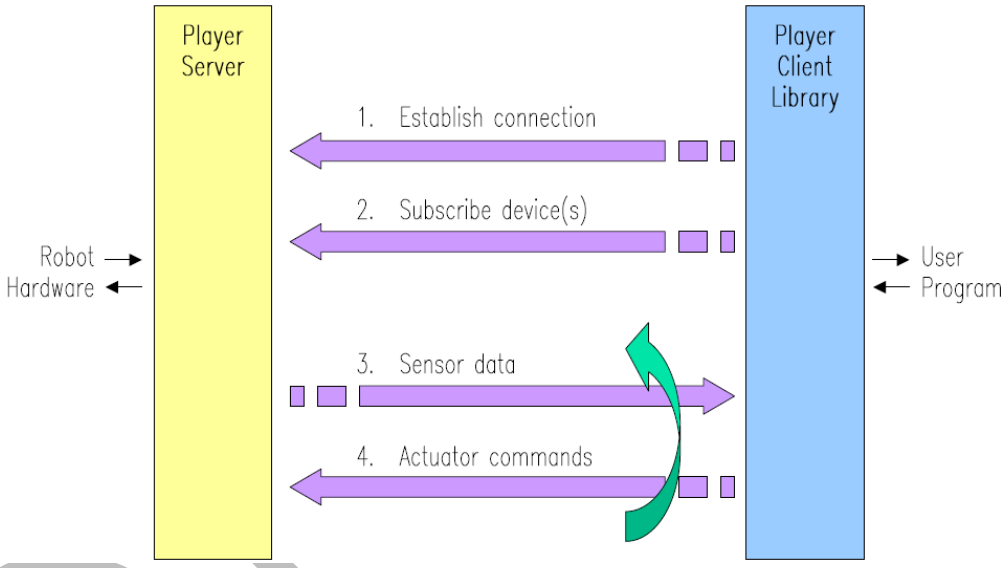



Figure 3.3: Client/server communication.

Player is complemented by two simulation systems, Stage and Gazebo. Stage simulates a population of mobile robots, sensors and objects in a two-dimensional bitmapped environment. Stage is designed to support multi-agent autonomous systems, so it provides fairly simple, computationally cheap models of lots of devices rather than attempting to emulate any device with great fidelity. Gazebo is a physically simulated multi-robot simulator. Like Stage, it is capable of simulating a population of robots, sensors and objects, but does so in a physically dynamic three-dimensional world using the Open Dynamics Engine. It generates both realistic sensor feedback and physically plausible interactions between objects with higher computational overhead. These simulation systems are often an excellent means to prototype robot controllers. However, these simulators do not have the same uncertainty

[2]Source: http://www.ki.inf.tu-dresden.de/ wiki/Robotics/Creatures/PlayerTutorial.pdf
[3]Source: http://www.ki.inf.tu-dresden.de/ wiki/Robotics/Creatures/PlayerTutorial.pdf

and nondeterminism as faced in the real world. We strongly recommend to students to thoroughly testing their controllers on a real robot before demoing or submitting assignments.

### 3.1.1 Client/Server Architecture

Player is a device server that provides an interface to robots, sensors and actuators. The components of the Player model is illustrated in Figure 3.4[4].

Devices (e.g., a laser, a camera, or a complete robot) are the actual hardware in the real world or simulated hardware that exists in a virtual environment maintained by Stage or Gazebo. Hardware resides at the lowest level in this model.

A robot server (e.g., Player) is the information interface between the robot and any program that requests information from or sends commands to the robot. Regardless of whether a device is real or simulated, the robot server provides the same interface to the robot for client programs. Thus, controllers developed on a simulated device will immediately run the equivalent real robot device. (Note: A robot's **behavior** is a function of both its controller and environment. A robot controller working in one environment does not necessarily imply the same controller will yield the same behavior in new or similar environments, given PSG support for the device.)

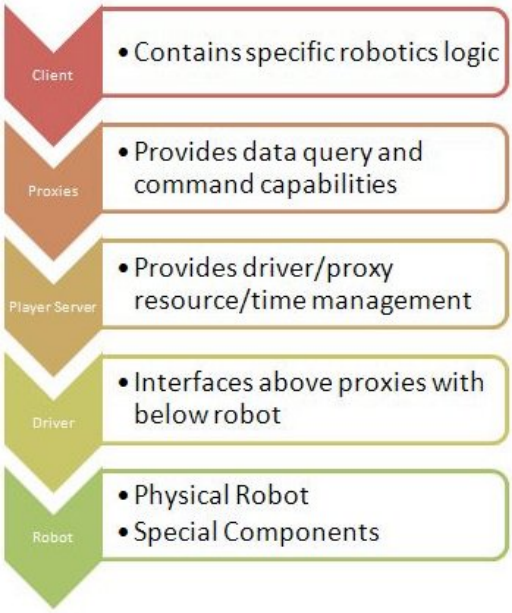



Figure 3.4: The Player model's layers of abstraction.

A robot client, at the highest level in this model, is a user-developed program that accesses robot functions through the robot server. The robot client will first establish a connection to the robot's server and then command the robot by reading data from the server and sending appropriate control command back. The job of the developer is to write control programs that produce commands that yield desired behavior from information received from the robot server[5]

Proxies are used by the client to retrieve information from a sensor or control an actuator; they expose interfaces to both hardware devices and existing implemented algorithms.

Drivers reside on top of the robot and provide access to specific functions of the robot.

Player uses a TCP socket-based client/server model, illustrated in Figure 3.3. The client is a given robot application which interacts with the server running on the robot. The server exposes devices as proxies and publishes data continuously. The client first establishes a network connection with the robot server and subscribes to robot proxies. The client then

[4]Source: http://psurobotics.org/wiki/index.php?title=Player/Stage_Drivers

[5]Referring to the lecture on autonomous control, the description of the robot client should remind you of feedback control (i.e., commands $[u_t]$ that produce desired state $[x_d]$ given observations $[y_d]$).

runs in a continual loop, receiving data from and sending actuator commands to the server. Because Player follows a publish/subscribe messaging paradigm to control the robot over a network, there is no dependency on which programming language robot applications are written in[6] and clients can execute from any computer on the network.

### 3.1.2 Proxies

The server continually publishes proxy information and the client sets variables to subscribed proxies to control the robot. The information being sent over the network between the client and server is only with respect to the proxies the client explicitly initialized.

The proxies that are of interest to applications written in CS148:

- position2d: controls mobile robot bases in 2D. Client sends motor commands for velocity control by setting a forward and rotate speed to the robot; allows access to the odometric information.
- bumper: returns data from the bumper device. This proxy accepts no commands.
- ir: provides access to an array of infrared (IR) range sensors. The ir proxy controls two sets of ir devices on the Create. The first is a small infrared on the tip of the Create that senses virtual walls; the second set are four ir sensors on the bottom of the Create for detecting cliffs on the ground.
- camerauvc: allows access to a USB camera and retrieves images from a camera device.
- blobfinder: provides access to devices that detect blobs in images used for color recognition.

The full list of device proxies can be found here.

### 3.1.3 Player Quick Start with Create

The following are instructions on how to use Player as a middleware abstraction to send commands to a robot. This is an example of client application in C++ that controls an iRobot Create to continually rotate in place and read bumper information from the robot.

1. Create a `client.cpp` program using the code below.

```
#include <stdio.h>
#include <libplayerc++/playerc++.h>
#include <signal.h>
```

[6]Given that the programming language has a Player client library to provide the appropriate messages, such as libplayerc for the C language.

```
//Use the player namespace, for cleanliness
using namespace PlayerCc;

//declare Ctrl-C-related variables
bool endnow = false;
//function to receive signals
void stopper(int x)
{
  endnow = true;
}

int main(int argc, char** argv)
{
  //Player setup
  PlayerClient *client = new PlayerClient(argv[1],6665);
  client->SetDataMode(PLAYER_DATAMODE_PULL);
  client->SetReplaceRule(true,-1,-1);

  // Create a position2d proxy (device id "position2d:0") and susbscribe
  // in read/write mode
  Position2dProxy *posProxy = new Position2dProxy(client,0);
  //set the robot to be stationary
  posProxy->SetSpeed(0,0);

  // Create a bumper proxy and subscribe
  BumperProxy *bumperProxy = new BumperProxy(client, 0);

  //tell the function 'stopper' to receive all SIGINT signals.
  signal(SIGINT,stopper);
  while(!endnow)
  {
    // read from the proxies
    client->Read();

    double forward = 0.0;
    double rotate = 0.3;

    printf("Forward is %g, rotate is %g.\n",forward,rotate);
    printf("Left: %d     Right: %d\n",
    bumperProxy->IsBumped(0),bumperProxy->IsBumped(1));
    // send commands to control the motors
    posProxy->SetSpeed(forward,rotate);
```

```
  }
  posProxy->SetSpeed(0,0);

  //cleanup
  delete posProxy;
  delete bumperProxy;
  delete client;

  return 0;
}
```

This program uses `libplayerc++`, which is a C++ client library for the Player server. Because libplayerc++ is used in this example, the position2d and bumper device proxies are not directly accessed. Instead the Position2dProxy and BumperProxy classes are used which control the position2d and bumper devices, respectively. Please see the Position2dProxy Class Reference and theBumperProxy Class Reference for the public member functions of each class.

This client first creates a PlayerClient proxy and establishes a connection to the server on the IP specified in the command-line argument (which can just be "localhost") listening on port 6665. PositionProxy and BumperProxy objects are created, using the client proxy object to establish access to the position2d and bumper devices. The program enters a loop (that may be exited using Ctrl-c) that spins the robot in place by sending commands to control the actuators by setting the velocity via the position2d proxy. Furthermore the client accesses the sensory information from the bumper proxy to determine if the bumper is currently pressed.

2. As described in 2.2.2, a Player configuration file must reside in a directory on the robot where you will start the Player server. Configuration files are used to inform the Player server which drivers need to be instantiated. The config file consists of driver sections, declared by the keyword `driver()`, which configure drivers. A driver section is composed of option-value pairs that are whitespace separated. Both the `name` and `provides` options are required in a driver section. The `provides` keyword specifics the device address(es) through which the driver can be accessed. The `requires` keyword specifies the device address(es) to which the driver will subscribe.

   Below is a sample `create.cfg` used for this exercise:

```
driver
(
        name "create"
        provides ["position2d:0" "bumper:0" "power:0" "ir:0"]
        port "/dev/ttyUSB0"
)
```

```
driver
(
        name "camerauvc"
        provides ["camera:0"]
)

driver
(
        name "cmvision"
        provides ["blobfinder:0"]
        requires ["camera:0"]
        colorfile "colors.txt"
)
```

Please refer to the configuration files link and the tutorial writing configuration files to learn more about config files.

3. pkg-config, a tool that searches for installed libraries when compiling, must know where libplayerc++ is installed. By default the pkg-config files for Player are installed in /usr/local/lib/pkgconfig and this directory should be in the search path for pkg-config. Run: `pkg-config --libs playercore`
   If the output is similar to "-lplayercore -lltdl -lpthread -lplayererror", you should be fine. Otherwise you will have to set the PKG_CONFIG_PATH environment variable to include the /usr/local/lib/pkgconfig path[7]:
   `export PKG_CONFIG_PATH=/usr/local/lib/pkgconfig:$PKG_CONFIG_PATH`

4. Compile the client application:
   `g++ -o client ‘pkg-config --cflags playerc++‘ client.cpp ‘pkg-config --libs playerc++‘`
   (Best to copy and paste this line; note the above single quotes are a different ASCII character then the typical single quote and you will get a compile error if this is not correct).

5. Turn on the Create base and the EeePC.

6. Start the Player server on the robot: `player create.cfg`

7. Run the client application.

   - If locally executing the client: `./client localhost`.

[7]Source: http://playerstage.sourceforge.net/doc/Player-2.1.0/player/install.html

- If the robot is on a wireless network, run the following from another computer on the network: `./client <robot IP>` where `<robot IP>` is the IP address of the robot.

Please refer to the Player website for a simple Player C example, and also for another Player C++ example.

## 3.2 ROS

ROS is a robotics software framework that provides an interface to the robot's sensors and actuators over an IP network. ROS enables a peer-to-peer network of distributed processes that process data together and use the ROS communication infrastructure. According to the system's developers:

> Robot Operating System (ROS), a Willow Garage software platform, is an open-source, meta-operating system for your robot. It provides the services you would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers.

### 3.2.1 Peer-to-Peer Messaging Model

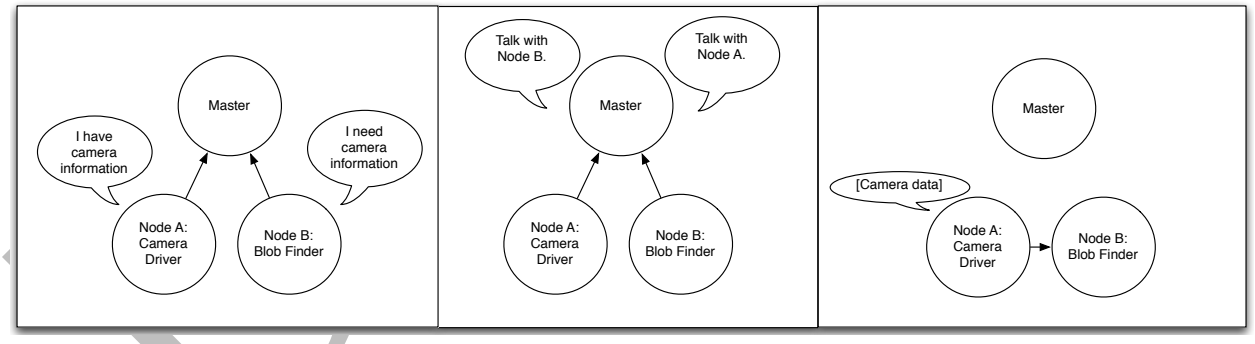



Figure 3.5: A conceptual view of a higher-level node calling a lower-level node. The master node is responsible for matching nodes to enable communication between them (source unknown).

ROS uses a peer-to-peer architecture over a TCP network, allowing for various styles of communication. The basic building block in ROS is a *node*. Each node is a modular process with some specialized purpose. A particular node, for example, might implement a robot's basic motor control, or publish information from a camera. Furthermore, these simple nodes may be combined in various ways to form more complex nodes governing more sophisticated

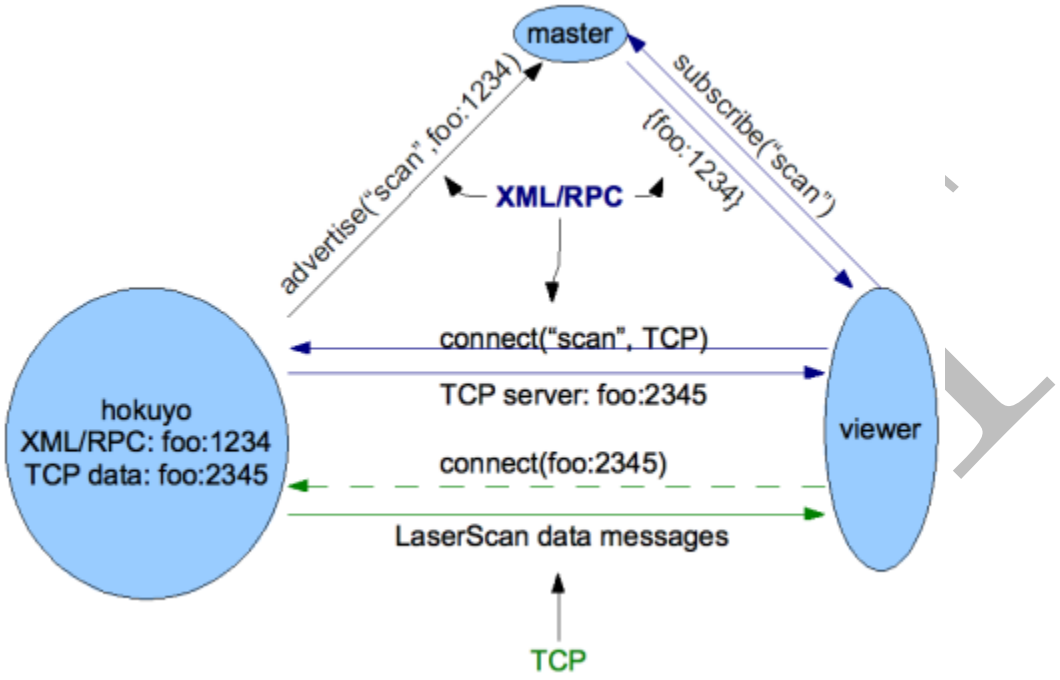



Figure 3.6: A diagram of controlling a Hokuyo laser range-finder. Details can be found at the ROS website.

behaviors. For instance, an obstacle-avoidance node may combine information from a robot's sensors, actuators and higher-level control architecture to compute a maneuver. A primary motivation of ROS is to promote code reuse, and the modularity and platform-independence of nodes helps serve this purpose.

Nodes communicate with each other by publishing messages to *topics*, which are the basic message structure for broadcasting data. Nodes publish messages to a topic as well as subscribe to a topic to receive messages, as seen in Figure 3.7[8]. Thus topics are intended for unidirectional, streaming communication. In general, nodes are not aware of who they are communicating with. Instead, nodes that are interested in data subscribe to the relevant topic; nodes that generate data publish to the relevant topic. Figure 3.6[9] provides an illustration of communication between nodes; the "hokuyo" node publishes messages on the "scan" topic and the "viewer" node subscribes to the "scan" topic. The two nodes are decoupled and need not know of each other.

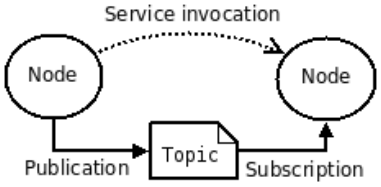



Figure 3.7: ROS messaging.

Nodes that need to perform remote procedure calls, i.e. receive a response to a request,

[8]Source: http://www.ros.org/wiki/ROS/Concepts
[9]Source: http://www.ros.org/wiki/ROS/Technical%20Overview

should use *services* instead. Unlike the publish/subscribe communication paradigm that topics provide, services allow for Remote Procedure Call (RPC) request/reply interactions which are often required in a distributed system. Request/reply is done via a Service, which is defined by a pair of Messages: one for the request and one for the reply.

A complex program could potentially consist of tens or hundreds of nodes, and these nodes need some efficient way to communicate with each other. Thus, every program in ROS is run with a special node called the *master node*. Conceptually, this node is a matchmaker: it is responsible for listening to node offers and requests, and then introducing relevant nodes to each other so that they can communicate. The master nodes provides naming and registration services to the rest of the nodes in the ROS system. They track publishers and subscribers to topics as well as services. Thus the role of the master nodes is to enable individual ROS nodes to locate one another. Once these nodes have located each other they communicate with each other peer-to-peer.

ROS client libraries for various programming languages, (eg `roscpp, rospy, rosjava`) exist that facilitate the developer to program ROS nodes, publish and subscribe to topics, and write and call services.

In addition to being a robot middleware system, ROS provides package management along with an integrated build system. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers. Finally ROS allows for integration of external packages, repositories and third-party software such as OpenCV, OpenRAVE and even Player.

### 3.2.2 ROS Quick Start with Create

The Brown ROS package is a set of nodes that provides basic sensing and movement functionality for the iRobot Create. This section demonstrates creating an `irobot_create_controller` node and writing a controller that provides a simple method to determine whether the robot's left or right bumper is pressed, as well as to set the robot's speed and rotation. This controller will rotate the Create in place until Ctrl-c is pressed.

- From the ROS root directory on the EeePC: `cd ros-1.0.0/pkg`
  (Don't forget to `source rosenv` from the ROS root directory).

- Create a ROS package for our controller:
  `roscreate-pkg irobot_create_controller roscpp std_msgs irobot_create_2_1`
  The `roscreate-pkg` command-line tool automatically creates a new ROS package called `irobot_create_controller` with dependencies on three other packages (`roscpp`, `std_msgs` and `irobot_create_2_1`). It generates a number of files for building the new package.

- Create a `client.cpp` file in `irobot_create_controller/src` and add the following code to `client.cpp`:

```
#include <ros/ros.h>
#include <irobot_create_2_1/SensorPacket.h>
#include <geometry_msgs/Twist.h>
#include <signal.h>

ros::Publisher create_pub;
ros::Subscriber create_sub;

// Declare Ctrl-C related variables
bool endnow = false;
void stopper(int x)
{
  endnow = true;
}

// Master node calls this function everytime a new message arrives
void createListen(const irobot_create_2_1::SensorPacketConstPtr& msg)
{
  printf("Left: %d, Right: %d\n", msg->bumpLeft, msg->bumpRight);
}

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

  // Initialize ROS
  ros::init(argc, argv, "controller");
  // Create a handle to this process node
  ros::NodeHandle n;
  // Publish messages of type geometry_msgs/Twist on topic "cmd_vel"
  create_pub = n.advertise<geometry_msgs::Twist>("cmd_vel", 1);
  // Subscribe to the "sensorPacket" topic.
  create_sub = n.subscribe("sensorPacket", 0, createListen);

  // Tell the function 'stopper' to receive all SIGINT signals
  signal(SIGINT,stopper);

  while(!endnow)
    {
      double forward = 0.0;
      double rotate = 0.3;

      // Create and publish a Twist message
      geometry_msgs::Twist twist;
```

```
      twist.linear.x = forward; twist.linear.y = 0; twist.linear.z = 0;
      twist.angular.x = 0; twist.angular.y = 0; twist.angular.z = rotate;
      create_pub.publish(twist);

      ros::spinOnce();
    }

  // Stop the robot's movement
  geometry_msgs::Twist twist;
  twist.linear.x = 0; twist.linear.y = 0; twist.linear.z = 0;
  twist.angular.x = 0; twist.angular.y = 0; twist.angular.z = 0;
  create_pub.publish(twist);

  printf("Done.\n");

  return(0);
}
```

The Create driver publishes a single message type, `SensorPacket.msg`, which exports all of the robot's sensory information. The message file exists in:
ros-1.0.0/pkg/irobot_create_2_1/msg/SensorPacket.msg
For online documentation, visit the Brown ROS Create Driver site.

Additionally, use this ROS tutorial as a helpful resource for writing simple Publish and Subscriber nodes in C++.

- Edit `irobot_create_controller/CMakeLists.txt` and add the following line to the end of the file:
  `rosbuild add executable(client src/client.cpp)`
  This causes an executable named `client` to be created whenever you run the command `rosmake irobot_create_controller`. For more information on CMake, refer to Section 3.5, however most of the work in creating CMakeLists is done automatically by ROS.

- Compile: `rosmake irobot_create_controller`

- Turn on the Create base.

- Run the ROS master: `roscore`

- Run the Create driver: `rosrun irobot_create_2_1 driver.py`

- Run the controller client: `rosrun irobot_create_controller client`

## 3.3 Other Middleware Packages

There are many suitable robot middleware packages beyond Player and ROS, some of which are described in brief below. For an excellent comparison of robot middleware, we refer the reader to the survey and analysis conducted by Huang et. al.[10]

### 3.3.1 LCM

Lightweight Communications and Marshalling (LCM) is "a library for message passing and data marshalling targeted at real-time systems where high-bandwidth and low latency are critical. It provides a publish/subscribe message passing model and an XDR-style message specification language with bindings for applications in C, Java, and Python. It was originally designed and used by the MIT DARPA Urban Challenge Team as its message passing system. LCM is designed for tightly-coupled systems connected via a dedicated local-area network. It is not intended for message passing over the Internet. LCM has been developed for soft real-time systems: its default messaging model permits dropping messages in order to minimize the latency of new messages."[11]

Features include:

- Low-latency inter-process communication
- Efficient broadcast mechanism using UDP Multicast
- Provides type-safe message marshaling that automatically detects most types of errors (such as version mismatches between different modules)
- User-friendly logging and playback
- Essentially unlimited packet size
- No centralized "database" or "hub" – peers communicate directly
- No daemons
- Supports C/C++, Java, Python, and MATLAB

### 3.3.2 Orocos

The Open Robot Control Software project aims to provide a modular framework for robot and machine control.[12]

[10] Huang, Olson and Moore, *Lightweight Communications and Marshalling for Low Latency Interprocess Communication*, Technical Report MIT-CSAIL-TR-2009-041, 2009.

[11] code.google.com/p/lcm

[12] www.orocos.org

### 3.3.3 YARP

Yet Another Robot Platform is "an open-source framework that supports distributed computation with an eye at robot control and efficiency. YARP supports building a robot control system as a collection of programs communicating in a peer-to-peer way, with a family of connection types that meet the diverse, sometimes contradictory, and always changing needs of advanced robotics."[13]

### 3.3.4 CARMEN

The Carnegie Mellon Robot Navigation Toolkit (CARMEN) is "an open-source collection of software for mobile robot control. CARMEN is modular software designed to provide basic navigation primitives including: base and sensor control, logging, obstacle avoidance, localization, path planning, and mapping."[14]

### 3.3.5 Microsoft Robotics Studio

Microsoft Robotics Studio is "a Windows-based environment for hobbyist, academic and commercial developers to create robotics applications for a variety of hardware platforms."[15]

### 3.3.6 JAUS

Joint Architecture for Unmanned Systems "was originally an initiative by the US Department of Defense to develop an open architecture for the domain of unmanned systems."[16]

## 3.4 Version Control using Subversion

Version control systems are a class of software that keeps track of revisions to files as they are made and edited. One popular open source version control system is Subversion. From `man svn`:

> Subversion is a version control system, which allows you to keep old versions of files and directories (usually source code), keep a log of who, when, and why changes occurred, etc., like CVS, RCS or SCCS. Subversion keeps a single copy of the master sources. This copy is called the source "repository"; it contains all the information to permit extracting previous versions of those files at any time.

CS148 requires students to use Subversion to submit the code for finished assignments. In addition to helping students keep track of the revisions they make for their source code,

[13] eris.liralab.it/yarp
[14] carmen.sourceforge.net
[15] msdn.microsoft.com/en-us/robotics
[16] en.wikipedia.org/wiki/JAUS

Subversion facilitates collaborative development on source code. Anyone in a project group can check out code, modify it, and commit their changes to a common repository. A student can then update the code they are working on with their changes without having to trade whole files back and forth. Additionally, code can be checked out directly to one of the class robots, allowing for greater portability when switching between robots.

Our aim is for students who have not used a version control system before will gain familiarity with version control and learn how to use Subversion to guide their group's workflow, while those who have will be able to start working with Subversion right away. Version Control with Subversion is a very comprehensive book freely available, which gives extensive documentation on Subversion.

### 3.4.1 Repository Setup by Admin

The course staff is responsible for setting up a Subversion repository for each group. We want students to be able to check out code directly onto the EeePCs. Because Subversion repositories cannot be accessed directly via SSH, the course staff runs a network service (*svnserve*) to allow for this. This requires an additional machine that both runs the service and hosts the repositories, as well as a wireless network that the robot EeePCs can connect to. *svnserve* is a small and lightweight server program; it is advantageous because there is no need to create system accounts on the server and passwords are not passed over the network. However by default, *svnserve* does not provide encryption or logging, and the server stores clear text passwords. Overall, *svnserve* is easy to setup and suits our purposes. For an overview and comparison of other svn server configurations options, please see Chapter 6 of "Version Control with Subversion", which is available online.

The following are instructions for an admin to create a student group repository, set up the repository structure, run *svnserve*, and test checking out. All of this should be done on another machine connected to the wireless network. In this example, COURSE_HOME is the root of your course directory and "148student" is the name of the student repository.

- Create a student repository.
```
cd COURSE_HOME
mkdir svn/148student
svnadmin create COURSE_HOME/svn/148student
```

- Create a repository structure with /tags, /trunk and /branches directories inside the repository. To do this, we first locally checkout the newly created repository, create each directory, commit and cleanup.
```
cd COURSE_HOME
svn checkout file://COURSE_HOME/svn/148student tmp
cd tmp
mkdir tags trunk branches
svn add tags trunk branches
svn commit -m "Created initial Subversion repository structure"
```

```
cd ..
rm -rf tmp
```

- In `svn/148student/conf/svnserve.conf`, uncomment/edit the following lines:
  ```
  anon-access = none
  auth-access = write
  password-db = passwd
  authz-db = authz
  realm = 148student repository
  ```

- Add the following lines to `svn/148student/conf/authz`:
  ```
  [groups]
  students = 148student
  [/]
  @students = rw
  ```

- Set the user password. In `svn/148student/conf/passwd`:
  ```
  [users]
  148student = MyPassword
  ```

- To disable password caching, in `/.subversion/config` on each EeePC, add/uncomment the line: `store-passwords = no`
  By default, Subversion caches passwords in plain text on disk. Since students will most likely share EeePCs and have access to every EeePC, it is necessary to disable password caching on the EeePC so students cannot check out each other's code.

- Run svnserve as a standalone daemon process:
  ```
  svnserve -d
  ```

- Test checking out from an EeePC connected to a wireless network.
  `svn checkout svn://<IP>/148student/trunk <dirname>`
  <IP> should be the IP address of the computer running svnserve.

- For more resources, read the manpages for `svnadmin`, the administrative command for Subversion. Also refer to Chapters 5 and 6 of "Version Control with Subversion".

### 3.4.2 Repository Use by Students

#### Concepts

Subversion may be thought of as a file versioning system that sits on top of a filesystem. With it, one can take a set of files, apply changes to them, commit the new changes, and continue working. At any point in time, one can revisit his or her files as they were at an earlier revision. What follows is a brief glossary of terms.

- Repository: a central location where your project resides.
- Working copy: a file tree you have checked out from the repository.
- Revision: every time you commit code to your repository, a new revision is created and given a number. You can refer to the repository as it was at a specific commit using that commit's revision number.
- Trunk: a file tree that contains your main body of code.
- Tags: a tag is a special name that you give to some snapshot of code in the repository. In Subversion, this is a copy of some file tree that you give a special name in the top-level directory tags.
- Branches: a branch is a section of material that is being worked on in separately from other code in order to isolate one set of changes from others. In Subversion, this is a copy of some file tree that you have given a special name in the top-level directory branches.

When accessing the repository for the first time, students will be prompted for a username/password pair. To specify a particular username to svn, use the –username argument.

### The Repository Structure

At their top level of a student's Subversion repository are the directories `trunk`, `tags`, and `branches`. Typically `trunk` should be used for whatever is currently being worked on, while `tags` and `branches` hold tags and branches of the work, respectively.

### Subversion commands

The command-line Subversion client is `svn`. Invocations of the client use this syntax:

```
svn <action> [arguments]
```

- Browsing the repository

  To examine the contents of the repository before checking anything out: `svn ls svn://host/groupname/svn`

  svn has analogues to the Unix `ls`, `cat`, `mv`, `rm`, and cp commands that can all work directly on the repository.

- Checking out

  To get started, one should check out a new working copy of a repository. This only needs to happen once, unless for some reason multiple working copies of your code exist. To check out your trunk directory, run:

  `svn checkout svn://host/groupname/trunk <dirname>`

  One should end up with a new directory with the name <dirname> where the command was run. One can treat this directory as if it were any other and develop projects in it. Future example commands with svn will assume that the commands are run from within the working copy that was just checked out. As you start out, you should see using `svn ls` that you have a few `branches` in branches and nothing else in any of the other top-level directories.

- Updating

  It is important to update a working copy of the repository whenever another group member commits any of their work. To update a working copy, run:

  `svn update`

  This will download any changes that other group members have made and then apply them to the current working copy. If there are conflicts between changes made in the current working copy and changes that have been committed to the repository, they will need to be resolved.

  To update to an older version of the repository:

  `svn update -r <revision>`

  See svn help update for information about what <revision> might be. If svn update is ran to look at old versions, it is important to svn update again later to keep updated with the latest repository version

- Examining the state of a working copy

  To see what changes have been made to a current working copy relative to what files in the repository, run:

  `svn status`

  See `svn help status` for information about what the output means. It is good practice to run this before committing, in order to see what changes have been made.

  To see specific changes made, use `svn diff`. This can be run with a specific file argument to display the changes made to that file.

- Manipulating files in a working copy

The `svn cp`, `svn mv`, and `svn rm` commands mentioned above may be used directly on a repository, and they should also be used to manipulate files in a working copy. To rename:

```
svn mv <file1> <file2>
```

This will rename the file in a manner that Subversion is aware of it and will track it when committing this change.

- Committing changes

  Committing work to the group repository is known as "checking in". Always run `svn update` before a commit.

  To mark the new files that should be committed as "added" before committing them to the repository, use `svn add`.

  ```
  svn add <file1> [file2] ...
  ```

  View the changed status of files with `svn status`.

  Once the new files have been added as files to commit, run:

  ```
  svn commit
  ```

  An editor will pop up asking for a message describing the commit. It is good practice to write something informative, maybe even detailing what semantic changes were made to each file, save the message, and then exit the editor.

- Resolving conflicts

  It does happen that, while working on one section of a project, someone else will commit something that overlaps with what is being worked on in your working repository. One may resolve the conflict immediately using an interactive prompt, or will need to look at the files with conflicting sections, edit them appropriately, and then mark them with:

  ```
  svn resolved <file1> [file2]
  ```

- Other helpful commands: `svn help <action>` to get usage for a specific action.
  `svn help` lists all of the Subversion commands.
  `svn revert` to undo changes to a file and reset it to a just-checked-out state. `svn log` to view a history of commits and their messages.

- Merging from branches

  We provide students with skeleton code for the first assignment. Before beginning work on the first assignment, students should pull its respective skeleton code into their group's trunk. Students should then proceed to work in their own trunk.

  Conceptually, what you need to do is merge the code in the branch onto whatever is currently in your trunk. This is very easy for assignment 1, as you are starting with

an empty trunk. Once you have checked out your trunk (with `svn checkout`), the command to do this is:

`svn merge svn://host/groupname/branches/asgn1_skel`

You then must commit the merged changes with `svn add` and `svn commit`.

See `svn merge help` for further explanation of this command. Once again, the section of the Subversion book on branching and merging gives a comprehensive treatment of the subject.

### 3.4.3 Assignment Submissions

To submit assignments, we expect students to "tag" their current working trunk directory; the course staff then checks out the current assignment folder within each group's tags directory. Remember, tagging is simply taking a snapshot in the repository. Students should follow the instructions below to submit for an Assignment 0:

```
svn add <files>
svn commit -m "Tagging assignment 0"
svn cp svn://host/groupname/trunk svn://host/groupname/tags/asgn0
```

## 3.5 Building with CMake

CMake is an open-source build system that we encourage students to use as an alternative option to the make build system. CMake tries to get around some of make's shortcomings (such as whitespace-sensitivity) by providing an easy way of declaring how your project is compiled. From the cmake manpage:

> CMake is a cross-platform build system generator. Projects specify their build process with platform-independent CMake listfiles included in each directory of a source tree with the name CMakeLists.txt. Users build a project by using CMake to generate a build system for a native tool on their platform.

Please refer to CMake's documentation as an additional resource.
CMake builds makefiles by default. Below is a "Hello World" example with CMake:

- In an empty directory, save the following as `hello.cpp`:

```
#include <stdio.h>
int main() {
  printf("hello world.\n");
  return 0;
}
```

- Save the following as `CMakeLists.txt`:

```
cmake_minimum_required(VERSION 2.0)
project(MyProject)
add_executable(hello hello.cpp)
```

- Run: `cmake .`
  This reads your CMakeLists.txt and builds your makefiles. In addition to Makefile (the actual makefile), CMake generates CMakeCache.txt, cmake_install.cmake, and the directory CMakeFiles. These can be ignored.

- Run: `make`
  This reads the makefiles written by cmake and builds your project. The next time you edit any of your existing files, you do not have to run `cmake .` again; you only need to `make` again to recompile the project. However if you add or remove files from your project or decide to change how it is built (such as by changing your compiler options or adding a library dependency), you do have to run `cmake .` again to regenerate a Makefile.

Player is not dependent on CMake, however ROS uses CMake as their core build system. If you use ROS you will not have to worry about creating CMakeLists.txt files because this is automatically done for you, however you will have to add lines to CMakeLists.txt for adding new libraries or executables to your ROS package.