



iROBOT
Robots for the Real World™
www.isr.com

Mobility Robot Integration Software User's Guide

Copyright and Liability Information

Copyright 2000, iRobot Corp. All Rights Reserved.

No part of this manual may be copied or distributed, transmitted, transcribed, stored in a retrieval system, or translated into any human or computer language, in any form or by any means, electronic, mechanical, magnetic, manual, or otherwise, including photocopying, or disclosed to third parties, without express written permission from iRobot Corp., 32 Fitzgerald Dr., Jaffrey, NH 03452, USA.

Limits of Liability

While every precaution has been taken in the preparation of this documentation, iRobot Corp. assumes no responsibility whatsoever for errors or omissions, or for damages resulting from the use of the information contained herein.

To the maximum extent permitted by applicable law, iRobot Corp., its officers, employees and contractors, and their suppliers disclaims all warranties, either expressed or implied, including, but not limited to, implied warranties of merchantability and fitness for a particular purpose, with regard to the hardware, software, and all accompanying or subsequently supplied written materials and documentation.

To the maximum extent permitted by applicable law, in no event shall iRobot Corp., its officers, employees or contractors, or their suppliers, be liable for any damages whatsoever (including without limitation, special, incidental, consequential, or indirect damages for personal injury, loss of business profits, or business interruption).

Special Note

Because some states/jurisdictions do not allow the exclusion or limitation of liability for consequential or incidental damages, the above limitations may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state/jurisdiction to state/jurisdiction.

Trademarks

ATRV, ATRV-Jr, ATRV-Micro, ATRV-Mini, B14, B14r, B21, B21r, FARnet, iRobot, K8, Magellan, Magellan Pro, Mobility, rFLEX, Robots for the Real World, TRANSIT, Urban Robot, and Urbie are trademarks of iRobot Corp.

Other product and company names mentioned may be trademarks or registered trademarks of their respective companies. Mention of third-party products is for informational purposes only and constitutes neither a recommendation nor an endorsement.

Contents

PREFACE	Preface
	Welcome To the World of iRobot Research Mobile Robotics ix
	Technical Support From iRobot ix
	Email Support x
	Web Page Support x
	Mailing List Support x
	Phone Calls x
	Documentation xi
	Using This Guide. xi
	Documentation Feedback xi
CHAPTER 1	Getting Started with Mobility Robot Integration Software
	Welcome to Mobility Robot Integration Software 1 - 1
	What is Mobility Robot Integration Software? 1 - 2
	Why did we develop Mobility? 1 - 4
	Getting Ready to Explore Mobility 1 - 5
CHAPTER 2	A Tour of Mobility With the MOM Graphical Interface
	Running MOM 2 - 1
	Viewing Sonar Sensor Output through MOM. 2 - 3
	Driving your Robot with MOM 2 - 4
CHAPTER 3	Examples of Mobility Programs
	Robot Programming Tips and Tricks 3 - 1
	A Simple Mobility Program: simple_follow 3 - 2
	Simple Follow Program. 3 - 4
	Notes On Simple Follow Program 3 - 7
	Getting More Sensor Readings: simple_follow_2 3 - 8
	Getting SICK-PLS Laser Scanner Readings 3 - 8
	Getting Base Sonar Readings 3 - 9
	A Note on Indexing on the B21r 3 - 9
	Getting Bump Panel Readings 3 - 10

The Program simple_follow_2	3 - 11
Makefile for simple_follow_2.....	3 - 19
Mobility In Action – More Realistic Examples	3 - 19
Square_and_Circle: A Mobility Sample Program.....	3 - 20
Square_and_Circle’s System Configuration	3 - 21
Filling Out Square_and_Circle’s Functionality.....	3 - 21
Wander: A More Interesting Mobility sample program.....	3 - 21
Wander’s System Configuration	3 - 21
Filling Out Wander’s Functionality.....	3 - 21
Designing a Mobility Program.....	3 - 21

CHAPTER 4

MOM — The Mobility Graphical Interface

MOM: An Overview	4 - 1
Understanding MOM’s Environment	4 - 3
iRobot Factory Pre-installed Configuration	4 - 5
Invoking the Naming Service.....	4 - 6
The Base Server	4 - 8
Starting Up MOM.	4 - 9
MOM’s Graphical Interface.....	4 - 10
Range (Sonar) View	4 - 11
The Object Hierarchy	4 - 12
Properties.....	4 - 13
Debug Output	4 - 14
Coming Soon.....	4 - 14
Creating and Running Objects	4 - 15
Active Objects	4 - 15
Hot-Pluggable Connections.....	4 - 15
Adding a Viewer	4 - 16
Loading Configurations.....	4 - 16
Saving Configurations.....	4 - 16

CHAPTER 5

Mobility Programming With Class Frameworks (C++ and Java)

The Mobility Class Framework Model (Language Dependent C++/Java) ..	5 - 1
Interface Methods	5 - 2
Helper Methods	5 - 2
Template Methods	5 - 3
Hook Methods	5 - 3

Building on the Mobility Class Framework 5 - 3
The Elements of Mobility Robot Integration Software 5 - 4
Mobility Tools: Robot Tools and User Interfaces 5 - 5
The Basics: Robot Components and Interfaces 5 - 5
 Interface Definitions 5 - 6
 Object Request Broker (ORB) 5 - 6
 O/S Abstraction Layers 5 - 7

CHAPTER 6

Mobility Robot Integration Software Overview

Mobility Robot Object Model (Language Independent) 6 - 1
 Robot Object Model Overview 6 - 1
An Example Mobility Robot Control System 6 - 3
 The Mobility Core Interfaces 6 - 5
 Contained Objects Interface 6 - 6
 Object Container Interface 6 - 7
 Property Container Interface 6 - 8
 ActiveObject Interface. 6 - 8
 Object Factory Interface 6 - 8
 Mobility Externalization Interfaces. 6 - 9
 MobilityComponents Module 6 - 9
 StateChangeHandler Interface. 6 - 9
 SystemComponent 6 - 10
 SystemComponentStatus. 6 - 10
 CompositeSystemComponent 6 - 10
 ActiveSystemComponent 6 - 10
 SystemModuleComponent 6 - 10
 StateObserver 6 - 11
 MobilityData Module 6 - 11
 DynamicObject Interface. 6 - 11
 Mobility StateComponents 6 - 11
 Managing System Configuration. 6 - 13
 Property Container Interface 6 - 13
 Putting the Components Together 6 - 13

CHAPTER 7

Mobility Building Blocks: Basic Robot Components and Interfaces

The Robot as a Hierarchy 7 - 1
 Robot Abstractions, Objects and Interfaces 7 - 2
 Sensor Systems 7 - 2

Sonar Sensing	7 - 3
How the Sonar Sensors Work	7 - 3
How The Sonar Sensors Can be Fooled	7 - 5
Infrared Sensing	7 - 6
Robotic Tactile Sensing	7 - 7
Odometry and Position Control: the RobotDrive Object	7 - 7
How Mobility Processes Encoder Data	7 - 8
Actuator System Abstractions	7 - 8
Robot Shape Abstractions	7 - 9
Behavioral Abstractions	7 - 9
Parallel Behaviors	7 - 9
Layers of Control.	7 - 9
Mobility Building Blocks for Extensibility	7 - 9
Keeping Track of Obstacles: Local Map.	7 - 10
The GUI Tools	7 - 10
The Programming Interface.	7 - 10
Playing Nice: Guarded Motion	7 - 10
The Programming Interface.	7 - 10
Getting to the Point: Pose Control.	7 - 10

CHAPTER 8

Sim: The Mobility Simulator

The Mobility Simulator	8 - 1
Sim's World Simulator Core	8 - 2
The GUI Tools	8 - 2
The Programming Interfaces	8 - 2
Sim's Robot Hardware Simulator Modules.	8 - 2
The Robot Simulator Visualization Interface	8 - 2
Web-Based Visualization Interface	8 - 2
RML 2.0 Interface	8 - 2
Running Other Objects with Sim	8 - 2

CHAPTER 9

Advanced Issues And Common Questions

How Do I...?	9 - 1
Work with a multi-robot team?	9 - 1
Write modules that handle multiple robots?	9 - 1
Deal with multiple threads in my modules?	9 - 1
Make my own interfaces and extend the robot object model?	9 - 1
Use my old BeeSoft programs with Mobility?	9 - 2
Use my old Saphira programs with Mobility?	9 - 2
Program Mobility from my LISP system?	9 - 2

Change a Mobility-defined interface?	9 - 2
Why Did You...?	9 - 2
Use CORBA 2.x as an interface standard?	9 - 2
Change from BeeSoft?	9 - 3
Support only C++ and Java?	9 - 3

APPENDIX A

Installing Mobility

Install Linux on the Robot's On-board PC and Prepare it for Mobility. . .	A - 2
For Red Hat Linux 5.1 only:	A - 3
Set Up a mobility Account.	A - 4
Download Mobility software.	A - 5
Install Mobility	A - 7
For Red Hat Linux 5.1 only:	A - 8
Configure an Off-board PC for Radio RS-232 Link	A - 8
Magellan.	A - 8
Install MOM Only on a Desktop PC.	A - 9
MOM-only on Linux	A - 9
MOM-only on Windows	A - 10
Install Base Server Only on a Robot PC.	A - 10

APPENDIX B

External Copyright Information

JaccORB and OmniORB2	B - 1
GNU Library General Public License	B - 1
Preamble	B - 1
Terms and Conditions for Copying, Distribution, and Modification	B - 4
No Warranty	B - 10
How to Apply These Terms to Your New Libraries.	B - 11

GLOSSARY

Glossary

Figures

FIGURE 1 - 1. The Mobility Environment in Context	1 - 3
FIGURE 2 - 1. MOM's Object Hierarchy View Window	2 - 3
FIGURE 2 - 2. MOM Drive View and Range View Windows.	2 - 4
FIGURE 4 - 1. MOM's Object Hierarchy View Window	4 - 10
FIGURE 4 - 2. Selecting MOM's Range View Window	4 - 11
FIGURE 4 - 3. MOM's Drive View Window and Range View Window .	4 - 12
FIGURE 5 - 1. Mobility in the Context of the C++ and Java Programming Environments	5 - 5
FIGURE 6 - 1. An Example Mobility Robot Software Setup	6 - 3
FIGURE 6 - 2. Mobility Class Diagram: Core and Components	6 - 5
FIGURE 6 - 3. Mobility Object States	6 - 7
FIGURE 6 - 4. Anatomy of a Mobility Base Server.	6 - 15
FIGURE 6 - 5. Data Flow in a Mobility Base Server.	6 - 16
FIGURE 7 - 1. How the Sonar Sensor Can Be Fooled: Ranging Errors .	7 - 5
FIGURE 7 - 2. How the Sonar Sensor Can Be Fooled: Angular Errors. .	7 - 6

Preface

Welcome To the World of iRobot Research Mobile Robotics

Welcome and congratulations! You have made a wise decision in purchasing your new robot from iRobot. iRobot is the acknowledged industry leader in the exciting field of cutting-edge mobile robotics. Everyone at iRobot is eager to help you get your robot up and running as quickly and easily as possible. Spend a few moments reading through the documents supplied with your new robot.

Technical Support From iRobot

iRobot wants your research robot to work for you. As you work with your robot, you may encounter questions or problems that are not adequately addressed in the documentation. When this happens, contact iRobot technical support. The best and most convenient way to contact technical support is by filling out the technical support request form at our website: www.rwii.com/rwi/rwisupport.html.

Email Support

iRobot technicians and engineers prefer email support dialogues. Email allows iRobot to preserve both problem descriptions and solutions for future reference.

If you have hardware or software questions not addressed in this User's Guide or other documentation, please email your question to support@rwii.com. Do not direct your email to a specific individual. This ensures that your question will receive attention from the engineer or technician best suited to help you. Be sure to include your name, university or business, robot serial number, a detailed description of your problem, and your phone number. An iRobot technician or engineer will get back to you, usually within 24 hours, and, in some cases, within minutes.

Web Page Support

Our web page, <http://www.rwii.com> (or <http://www.isr.com>) includes some support information. Authorized Mobility users can download the latest releases of the Mobility software and documentation from the software section of the site. As new support pages are brought up, iRobot will announce them via a mailing list.

Mailing List Support

The mailing list users@rwii.com provides users of Mobility and other iRobot research robotics products a forum to share ideas, concerns, and thoughts. iRobot will make important announcements through this list. These could include announcements, software updates and patch announcements, solutions to commonly experienced problems, and late-breaking iRobot and robotics news.

NOTE: This is a public list; all subscribers will see all posts. To subscribe, send a message to: users-request@rwii.com. In the subject line type "subscribe". It will ignore any text in the body of the email. iRobot uses a mailing list manager called SmartList.

Phone Calls

We understand that, occasionally, a phone call may be necessary. Call 603-532-6900 for support, or fax us at 603-532-6901. If you do call with a question please have your robot within easy reach when you call.

Documentation

Some iRobot research robots are equipped with hardware and software from other suppliers. Since the exact configuration of each robot may be different, documentation for third-party extensions is presented separately. Think of the third-party documents as appendixes to this User's Guide.

Using This Guide

This User's Guide, in combination with the other documents you have received, is your tool for getting started with your robot. The first two chapters show you how to set up radio communication between your robot and your base computer. The next chapters identify the important parts, connectors, switches, and other features of your robot and how to set it up for operation. The final chapters tell you how to drive your robot, and perform routine maintenance. The appendixes provide warranty information, a list of available accessories, rFLEX Control system details, and other related information.

Documentation Feedback

iRobot wants all its documents to be complete, accurate, friendly, and helpful. We welcome feedback to help us improve our documentation. If you find an error in a document, feel an explanation is unclear, or feel something is missing or incomplete, please send email to docs@rwii.com.

Getting Started with Mobility Robot Integration Software

Welcome to Mobility Robot Integration Software

Review this *Mobility Robot Integration Software User's Guide* to get acquainted with the basic philosophy and concepts of Mobility, and to get started writing your own Mobility programs, using Mobility-supplied tools.

To supplement this User's Guide, a complete set of reference documentation for all Mobility interfaces has been supplied as part of your Mobility distribution. It is in HTML-format, allowing easy online viewing and rapid cross-reference between sections while programming.

In this chapter, you will take a whirlwind tour of Mobility's object-oriented architecture and find out a little about the structure of the system.

In the following chapter, you will walk through a simple example Mobility program, "Simple-Follow," designed to introduce you to Mobility concepts and programming. Because Mobility is such a revolutionary robot software development package, even veteran programmers and experienced roboticists can learn much from studying this sample program.

Further chapters describe the Mobility robot integration software in greater detail.

The complete Mobility robot integration software package has been pre-installed on your system. There are only a few simple steps you need to take to get Mobility up and running in your own environment. You will learn how to do this in the next chapter.

But first, take a few minutes to read the overview of the Mobility system.

NOTE: If you are upgrading to (from BeeSoft, for example) or evaluating Mobility and did not purchase a fully configured computer system from iRobot, or, for some reason need to re-install Mobility, see Appendix A “Installing Mobility” for detailed instructions for installing Mobility.

What is Mobility Robot Integration Software?

Mobility robot integration software is a distributed, object-oriented toolkit for building control software for single and multi-robot systems. Mobility consists of the following:

- A set of software tools
- An object model for robot software
- A set of basic robot control modules
- An object-oriented class framework to simplify code development

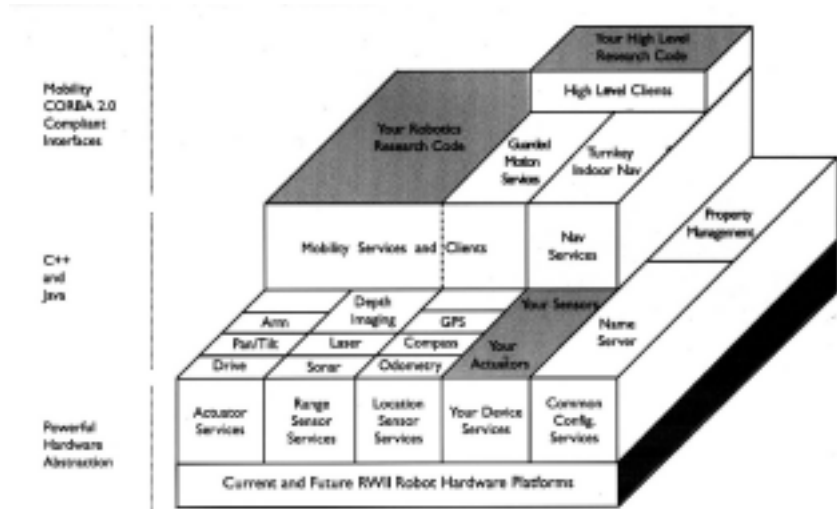


FIGURE 1 - 1. The Mobility Environment in Context

As Figure 1 - 1, “The Mobility Environment in Context,” on page 1 - 3 shows, Mobility defines the Mobility Robot Object Model using the Common Object Request Broker Architecture (CORBA) 2. X standard Interface Definition Language (IDL). By following the CORBA 2.x standard, Mobility supports many languages across many computing platforms. The Mobility Class Framework complements the Robot Object Model and greatly simplifies the development process by allowing you to reuse implementations of the basic system elements and only add what you need to implement your ideas. Appendix A “Installing Mobility” provides further information on CORBA itself

The Robot Object Model defines a robot system as a distributed, hierarchically organized set of objects. Each object is a separate unit of software with an identity, interfaces, and state. Objects represent abstractions of whole robots, sensors, actuators, behaviors, perceptual processes and data storage. Objects provide a flexible model of a robot system that can be reconfigured as new hardware, new algorithms, and new applications are developed.

The Mobility Class Framework mirrors the Robot Object Model and handles much of the grunt work involved in programming robot software. By deriving a new class from the Mobility Class Framework, you can easily add your own sensors, actua-

tors, behaviors, perceptual processes and data classes to the system. Mobility even allows you to extend the Robot Object Model itself by defining your own interfaces using CORBA 2.X standard Interface Definition Language (IDL). Because all objects in the Mobility environment support a common set of interfaces, they are called System Components.

Mobility supports both Java and C++, but Release 1.1 provides the class framework only for the C++ language. The Mobility tool set runs on Linux 2.x. Mobility is also compatible with Java 1.1 and uses Java to provide cross-platform user interfaces for configuration, management, testing and visualization of your robot software system in action.

Why did we develop Mobility?

In working with roboticists in labs all over the world, iRobot engineers couldn't help noticing that none of the several robot development systems in use really offered researchers the best, most capable development environment possible. After carefully studying the deficiencies in existing systems, iRobot engineers designed Mobility robot integration software from the ground up, specifically to support:

- Extensibility over time
- Multiple robot systems
- Integration among researchers
- High software component reuse
- Parallel and distributed processing for robot control
- Adherence to standardized protocol where feasible
- A degree of robot independence

The combination of these elements distinguishes Mobility robot integration software from other available systems. You might be used to having a library with a fixed set of functions (a “robot API”) as your primary abstraction for robot programming. Mobility gives you much more than this. Think of Mobility’s System Components as an “extensible API.”

NOTE: The online reference is the definitive reference for all the APIs within Mobility.

Mobility is specifically designed to address a critical issue in robotics research: system integration. Real World Interface wants to support the growing robotics

research and development community with the Mobility Development Environment so that they may more readily integrate their research ideas into working robot systems. Therefore, Mobility adheres to standardized protocols where applicable.

Getting Ready to Explore Mobility

Your robot system was delivered from iRobot with Linux and Mobility pre-installed.

NOTE: If Mobility was not pre-installed, or you need to re-install it, see Appendix A “Installing Mobility” for instructions. Install Mobility before proceeding.

A Tour of Mobility With the MOM Graphical Interface

The best way to get acquainted with Mobility is through its graphical user interface component, called the Mobility Object Manager, or just MOM. To use MOM, you must first have assigned your robot an IP address and a hostname, and have established communication with your robot.

After your quick initial tour with MOM, you'll want to study MOM's structure and capabilities more completely by reading Chapter 4 "MOM — The Mobility Graphical Interface".

NOTE: The information here assumes that your robot computer has a full Mobility installation.

If you want to run MOM on a computer that does not have Mobility installed, you can create "MOM-only" configuration and copy it to the display computer. Instructions for creating a "MOM-only" Mobility installation can be obtained by contacting iRobot support. MOM runs on Linux, Windows 95, and Windows NT.

Running MOM

This section describes a very simple way to run MOM that will work if your robot has an on-board PC with a Mobility 1.1 installation as shipped from Real World

Interface. (Chapter 4 “MOM — The Mobility Graphical Interface” explains other ways to set up MOM.)

Some robots, such as the Magellan, may have no on-board PC. They use a radio RS-232 link to communicate with a desktop PC running Linux. These instructions will also work once you have installed the Mobility 1.1 software onto the desktop PC connected to the radio RS-232 link. (See Appendix A “Installing Mobility” for detailed instructions.)

There are five steps to start up your robot and run MOM.

1. Make sure the robot is turned on and enabled for software control. Check the User's Guide for your robot and the *rFLEX Robot Control System User's Guide* to review correct robot startup and shutdown procedures.
2. Log in to the computer as user:mobility, password:mbyrwi
3. Run the Naming Service with the command
prompt> name -i
4. Run the base server with the command
prompt> base
5. Open another window on the robot computer and run MOM with the command
prompt> mom

This simple method allows you to check out your robot, Mobility, and MOM. However, there are certain disadvantages to running a graphical program such as MOM over a radio network to display on an X server on another computer. For MOM to display properly, the DISPLAY environment variable must be set correctly and the X server must allow MOM to access its display.

In most University and research settings, these details are taken care of by default. You may need to give the command:

```
prompt> xhost +
```

on your desktop computer.



FIGURE 2 - 1. MOM's Object Hierarchy View Window

CAUTION: There may be significant display and mouse-input performance degradation due to displaying MOM over the network. You must use extreme caution when driving your robot using MOM in this configuration, as there may be significant delays between when you make mouse movements and button events, and when the robot can respond to them,

Viewing Sonar Sensor Output through MOM

To see input from your robot's sonar sensors, click on <your robot>, then on "Sonar." Then, right-click on "Segment." A menu will pop up. Select "Range View." A new window will appear inside the framing MOM window, dynamically showing the sonar data coming from your robot.

The range viewer polls the robot for a new reading once every second.

Driving your Robot with MOM

To drive your robot through MOM, click on <your robot>, then on “Drive.” Then, right-click on “Command” and select “Drive View.” A new window will appear inside the MOM window.



FIGURE 2 - 2. MOM Drive View and Range View Windows

This interface operates essentially the same as the joystick. Until you have some experience driving your robot with the MOM interface, be especially careful to move slowly and carefully.

Move the mouse to the center of the Drive View window. Press and hold down your mouse button. While holding the mouse button:

1. Slide the mouse up to drive the robot forward
2. Slide the mouse down to drive the robot in reverse.
3. Move the mouse right on the screen to turn the robot to the right.
4. Move the mouse left on the screen to tun the robot to the left.

NOTE: Note: The further from the center of the window, the higher the robot's velocity.

This concludes your brief tour of Mobility using MOM. We hope you're enjoying your new Real World Interface robot! To learn more about MOM and to its many features and abilities, turn to Chapter 4 “MOM — The Mobility Graphical Interface”.

Examples of Mobility Programs

After studying this chapter, you'll know how to write your own components using the standard Mobility interfaces and the Mobility Class Framework. Here at iRobot, we've used the same procedures described here to develop all the components included in Mobility robot integration software.

If you're not a seasoned robot programmer, read the next section on Tips and Tricks before you do too much robot programming. But because Mobility is so revolutionary, even veteran programmers and experienced roboticists might learn something here.

Robot Programming Tips and Tricks

Our robots are, simply, self-contained, self-propelled mobile units equipped with sensors for making observations about the environment in which they operate, and with mechanisms for moving around in, and affecting, that environment. For example, an iRobot robot can roll over to a workbench, stop just short of it, pick up an object, turn in another direction, roll to another workbench, and put the object down. One of our robots has even been trained to roam the corridors of a museum exhibit and interact with visitors.

Training a robot to accurately observe and reliably interact with its environment might seem simple, especially with the advantages of Mobility robot integration software. But robots are often called upon to operate in noisy, messy, unpredictable, rapidly-changing environments. Robot programmer must always be cognizant of such factors as inexact observations, imprecise movements and undetected errors. Responsible roboticists take pride in their safety precautions, prudent lab management policies and in the special care they take to prevent uniquely robotic calamities.

As a robot programmer, you need to be always aware of the unique characteristics of your robot's mechanisms and its surroundings. For example, when conventional software detects an error, it either recovers automatically or sends a notice about the exact nature of the error. A robot, on the other hand, might simply stand there, roll away in the wrong direction or enter some undefined state.

So, robot programmers can't simply follow the algorithm: "Observe, act and, if no errors are reported, exit and pursue the next task in line." Rather, the robot programmer must take special care to try to increase the likelihood that the action requested does, in fact, occur. A more iterative algorithm increases the odds: "Observe, act, observe again to discern the results of acting, act again to correct inaccuracies in the previous action, observe again..." and so on. Such an approach is built right into Mobility. The example Mobility programs presented later in this chapter illustrate this design approach.

CAUTION: The following sample programs will cause your robot to move under the control of software. Make sure you disconnect all cables dangling from the robot, place the robot in a clear, flat, open area and familiarize yourself with the location of all robot emergency stop buttons before running any sample programs.

Remember you can halt sample programs by pressing return.
Remember you can halt your robot at any time by pressing any emergency stop button.

A Simple Mobility Program: simple_follow

Before too long, you'll be on your way to writing a full-fledged Mobility program with its own components, active objects and properties. But first, take a few

moments to go through this simple sample program that lets you use the components in a Mobility base server in a client-only manner. This example, called `simple_follow`, is a robot control program that reads the sonar sensors of your robot and drives the robot forward until it is within about 1 meter of the nearest obstacle, at which point the robot stops. Run the example and then look through the source code to learn how it works.

NOTE: Some robots require additional steps to enable robot motion after the base server has begun executing. Check the documentation for your own robot for more information on this important safety feature.

To get started easily, start up MOM. (See Chapter 2 “A Tour of Mobility With the MOM Graphical Interface”.)

Start up the base server for your robot. (If you remember the name you gave the base server, you can just type it in for the sample program, or use MOM to look at the objects running in your base server to get the robot name.)

While you're running `simple_follow`, view the robot's sonars with MOM.

Run the sample program by typing:

```
prompt> simple_follow -robot <your robot name here>
```

You'll see the banner for the program start and then (after a few moments) a scrolling list of range numbers. Your robot will start moving, striving to establish a 1-meter distance between itself and obstacles in its environment. If the robot senses that it is too close (about 30cm) to an obstacle, it will stop moving. Terminate `simple_follow` at any time by pressing the enter key. If you're running MOM, you'll be able to view the sensor feedback while your sample program is running.

The source code for the `simple_follow` sample program is available in your Mobility distribution as the file:

```
src/mby/examples/simple_follow/simple_follow.cpp
```

The file is also included here, so you can follow along and see how such a program is constructed.

Simple Follow Program

```
//
// Simple test program for the "user library" framework
// classes of Mobility.
// This program is designed to exercise and test the function-
// ality of the
// simplified environment provided by the user library.
//
// Robert Todd Pack, IS Robotics, Real World Interface Divi-
// sion.
//
#include "userlib.h"
#include "mobilityutil.h"
int main (int argc, char *argv[])
{
    char *modulename;
    char *robotname;
    SystemModule_i::init_orb(argc,argv);
    // Pick module name arguments.
    modulename = mbyUtility::get_option(argc,argv,"-
name","UserTest");
    mbyBasicModule *m_pModule = new mbyBasicModule(module-
name,argc,argv);
    // Pick robot name arguments.
    robotname = mbyUtility::get_option(argc,argv,"-robot","ATRV-
Jr");

    mbyBasicRobot *m_pRobot = new mbyBasicRobot("RobotX",robot-
name);
    // Use this robot in our module.
    if (m_pModule->add_new_component(m_pRobot) < 0)
        fprintf(stderr,"Module add error.\n");
    // Turn the module on, initializes robot objects and con-
// nects to servers.
    m_pModule->start_module();
    // We're going to look at some range data so make a place for
// it.
    MobilityGeometry::SegmentData ranges;
    unsigned int index;
    // These are parameters for our "follow" algorithm.
    float haltdist = 0.2; // 20cm halts
    float followdist = 0.9; // 0.9m follow distance.
    float mindist; // Computed minimum sensor distance.
    float minfrontdist; // Computed minimum front distance.
```

```
float tempdist;           // Computed temp dist value compared
to min/minfront.
float translate,rotate; // Command velocities.
float minx;              // X coordinate of minimum reading.
// Now we're ready to control the robot. Do whatever we want
in this loop,
// call other functions, access the robot object. The inter-
nal threads
// of the Mobility C++ class framework handle all the low-
level details
// for you.
while (1)
{
    m_pRobot->get_range_state(ranges); // Get latest range
data we have.
    // Process sonar data (you get back a set of line seg-
ments).
    // This shows how you can loop through all the segments
you
    // get back from the sonar source.
    mindist = 100000.0; // Set to "really large" dis-
tances.
    minfrontdist = 100000.0;
    minx = 0;
    for (index = 0; index < ranges.end.length(); index++) {
        tempdist = sqrt((ranges.org[index].x -
ranges.end[index].x)*
                        (ranges.org[index].x -
ranges.end[index].x)+
                        (ranges.org[index].y -
ranges.end[index].y)*
                        (ranges.org[index].y -
ranges.end[index].y));
        // Find the minimum length value.
        if (tempdist < mindist)
            mindist = tempdist;
        // Find the minimum front distance value. (Only things
in front
        // of the robot count).
        if ((ranges.end[index].x - ranges.org[index].x) > 0.2) {
            if (tempdist < minfrontdist)
            {
                minfrontdist = tempdist;
                minx = ranges.end[index].x;
            }
        }
    }
}
```

```
    }
  }
  // Show what you found.
  fprintf(stderr,
    "Sonar Values: Minimum Dist: %f Min Front Dist:
%f\n",
    mindist,
    minfrontdist);
  // Compute new drive command based on these distances.
  if (mindist < haltdist) {
    translate = 0.0;
    rotate = 0.0;
  }
  else if ((minfrontdist < (followdist + 0.2)) &&
    (minfrontdist > (followdist - 0.2))) {
    translate = 0.0;
    rotate = 0.0;
  }
  else if (minfrontdist > (followdist + 0.2)) {
    translate = 0.15; // Drive forward.
    if (minx < -0.3)
      rotate = -0.1;
    else if (minx > 0.3)
      rotate = 0.1;
    else
      rotate = 0.0;
  }
  else if (minfrontdist < (followdist - 0.2)) {
    translate = -0.15; // Drive backward.
    rotate = 0.0;
  }
  // Show our command.
  fprintf(stderr, "CMD: %f %f", translate, rotate);

  // Send a command based on what we've seen on sensors.
  if (m_pRobot->send_velocity_command(translate, rotate) <
0)
    fprintf(stderr, "Command error.\n");

  // Check for keypress to terminate the program.
  if (mbyUtility::chars_ready() > 0) {
    // Stop robot before we exit.
    m_pRobot->send_velocity_command(0.0, 0.0); // Zero
velocity is stop.
    break;
  }
}
```

```
    }
    else // Wait a small time before the next loop. 0.2 sec-
ond, keep going.
    {
        // This is an portable thread library call to sleep
your loop.
        omni_thread::sleep(0,200000000);
    }
}
// Turn module off, disconnects from servers and releases
resources.
m_pModule->end_module();
// Clean up memory.
delete m_pRobot;
delete m_pModule;
return 0;
} // End of main.
```

Notes On Simple Follow Program

The things to remember from this example are:

Mobility locates components by pathnames (like the sonar sensor object and the drive command object).

- Mobility components support many interfaces, and you can ask for the interface you need (like the SegmentState interface).
- Mobility components provide different, portable abstractions of robot hardware, and you can ask for the abstraction you need (like the objects contained under Sonar).
- You can use MOM to check on your robot software while it 's running and see it in operation in real-time (like when you watched the sonar readings when you ran simple_follow).
- The simple_follow example makes a fairly unintelligent robot. (We'll fix this as we go along!)

In the following section, you'll find out how to extend simple_follow to get readings from some of your robot's other sensors.

Getting More Sensor Readings: simple_follow_2

Mobility provides uniform interfaces to many different types of hardware. The code in the simple_follow sample program described above can be modified and extended to obtain range readings simply by changing the name of the object that is the data source.

Getting SICK-PLS Laser Scanner Readings

For example, let's say you want to get the readings from your robot's optional SICK-PLS laser scanner.

NOTE: This option is not available on all robot models. If your robot is equipped with a SICK-PLS laser scanner, you have a program called pls-server that is a Mobility hardware server for the laser. If you start up the robot base server and pls-server as well, you'll find both of them viewable through MOM, the Mobility Object Manager.

```
simple-follow:
// Build a pathname to the component we want to use to get
sensor data.
    sprintf(pathName,"%s/Sonar/Segment",robotName); // Use robot
name arg.
// Locate the component we want.
    ptempObj = pHelper->find_object(pathName);
change to:
// Build a pathname to the component we want to use to get
sensor data.
    sprintf(pathName,"Pls/Laser/Segment");
// Locate the component we want.
    ptempObj = pHelper->find_object(pathName);
```

Then, when you get the segment state object, the data source will be the on-board laser scanner. You could add a new segment state object and use them both in the program. (They both provide the same kind of data, that is, range data segments.)

new code:

```
// Find the laser data (we're going to use sensor feedback).

// The XX_var variable is a smart pointer for memory manage-
ment.

    MobilityGeometry::SegmentState_var pLaserSeg;
```

```
MobilityGeometry::SegmentData_var pLaserSegData;

// Request the interface we want from the object we found

try {

    pLaserSeg = MobilityGeometry::SegmentState::_narrow(ptem-
pObj);

}

catch (...)

{

    return -1; // We're through if we can't use sensors.

}
```

Later in your loop:

```
    // This actually samples the laser state. Looks just
like
    // the one for accessing sonar.
pLaserSegData = pLaserSeg->get_sample(0);
```

This code lets you get data from your on-board sonars and your SICK-PLS laser scanner at the same time.

NOTE: Future releases of Mobility will include a framework to make such things much simpler and more straightforward!

Getting Base Sonar Readings

Use the same approach to get base sonar sensor data as well, simply by using the name of the base sonar object (look at the server using MOM to find the object names you want) instead of "Sonar" above. If your robot is a B21r, read the note below for some robot-specific details on indexing rotation and base-skirt sensors.

A Note on Indexing on the B21r

Mobility is aware that the base skirt of the B21r does not rotate with the robot. After one full rotation of the robot, the skirt index is updated and the positions of bump panels and sonars on the base are available in correct robot coordinates.

There is no "automatic" indexing, because it is usually too annoying for people. If you need the skirt sensor readings to be valid and in robot coordinates, have the robot do a slow spin when you first start the base server. Upon completion of the spin, it will be correctly indexed. Mobility will subsequently transform your sensor readings into robot coordinates correctly.

Getting Bump Panel Readings

The bump-panels of the robot server use a different data type, called PointState. The bump-sensors are a collection of points and a set of flags indicating whether the bump switch at a given point is closed. They work in the same manner as the other sensors, with a very similar interface, the only differences are the types of data and the source object names.

You can get ahold of the bump data, for example, in the robot's base, using this code:

```
// Build a pathname to the component we want to use to get sensor data.
    sprintf(pathName, "%s/BaseContact/Point", robotName); // Use robot name
arg.
// Locate the component we want.
    ptempObj = pHelper->find_object(pathName);
    MobilityGeometry::PointState_var pBumpPoint;
    MobilityGeometry::Point3Data_var pBumpData;
    // Request the interface we want from the object we found
    try {
        pBumpPoint = MobilityGeometry::PointState::_narrow(ptempObj);
    }
    catch (...)
    {
        return -1; // We're through if we can't use sensors.
    }
```

Later in your loop:

```
        // This actually samples the laser state. Looks just like
        // the one for accessing sonar and laser
    pBumpData = pBumpPoint->get_sample(0);
        // You use bump data like this:
        // The BumpData will be a sequence of Point3 types
```

```
        // just like the sonar data is a sequence of Segment
types.
    pBumpData->point[1..k].x
    pBumpData->point[1..k].y
    pBumpData->point[1..k].z
    pBumpData->point[1..k].flags == 1 if the bump is hit.
```

The base server transforms all sensor readings into robot coordinates. Lengths and distances are in meters. Orientations and angles are in radians. There is a "right-handed" robot coordinate system fixed to the drive system of the robot. The +X direction is robot forward. The +Y direction is to the left of the robot. The +Z direction is up.

The Program simple_follow_2

/* This is the simple follow 2 sample program for Mobility 1.0:

The purpose of this example is to show how to use other sensors in a simple follow program.

This program is a stepping stone to a full fledged Mobility program that provides the same functionality, but is built from Mobility components, rather than a simple main program loop.

NOTE: This program is not an example of good robot program design. It is intended as a simple introduction to using some of the functions and interfaces in Mobility.

```
    Real World Interface, Inc.      Robert Todd Pack
*/
// These includes pull in interface definitions and utilities.
#include "mobilitycomponents_i.h"
#include "mobilitydata_i.h"
#include "mobilitygeometry_i.h"
#include "mobilityactuator_i.h"
#include "mobilityutil.h"
#include <math.h>

// Start of main program.
int main (int argc, char *argv[])
{
    //
    // This framework class simplifies setup and initialization
for
    // client-only programs like this one.
```

```
//
mbyClientHelper *pHelper;
// These variables are "environments" which are used to pass
around
// complex (more than success/fail) error information.
// CORBA::Environment env,env2;

// This is a generic pointer that can point to any CORBA
object
// within Mobility.
CORBA::Object_ptr ptempObj;
// This is a smart pointer to an object descriptor. Automa-
cially
// manages memory.
// The XXX_var classes automaticall release references for
// hassle-free memory management.
MobilityCore::ObjectDescriptor_var pDescriptor;
// This is a buffer for object names.
char pathName[255];
// Holds -robot command line option.
char *robotName;
// ACE_OS is a class that provides a portable wrapper around
lots
// of standard OS/C library functions like fprintf.
fprintf(stderr,
"***** Mobility Simple-Follow-2 Example*****\n");
// Look for robot name option so we know which one to run.
robotName = mbyUtility::get_option(argc,argv,"-robot");
if (robotName == NULL) {
    fprintf(stderr,"Need a robot name to use.\n");
    return -1;
}
fprintf(stderr, "Connect Sonar.\n");
// All Mobility servers and clients use CORBA and this ini-
tialization
// is required for the C++ language mapping of CORBA.
pHelper = new mbyClientHelper(argc,argv);
// Build a pathname to the component we want to use to get
sensor data.
sprintf(pathName,"%s/Sonar/Segment",robotName); // Use robot
name arg.
// Locate the component we want.
ptempObj = pHelper->find_object(pathName);
// Find the sonar data (we're going to use sensor feedback).
```

```
    // The XX_var variable is a smart pointer for memory manage-
ment.
    MobilityGeometry::SegmentState_var pSonarSeg;
    MobilityGeometry::SegmentData_var pSegData;
    // Request the interface we want from the object we found
    try {
        pSonarSeg = MobilityGeometry::SegmentState::_narrow(ptem-
pObj); //,env);
    }
    catch (...)
    {
        return -1; // We're through if we can't use sensors.
    }
    /
    *****/
    /* New Stuff Here: Optional laser scanner sensor inputs*/ /
    *****/
    fprintf(stderr, "Connect Laser.\n");
    // Build a pathname to the component we want to use to get
laser
    // scanner data. Only works using standard pls-server for
SICK-PLS.
    sprintf(pathName, "Pls/Laser/Segment");
    // Locate the component we want.
    ptempObj = pHelper->find_object(pathName);
    // Find the laser data (we're going to use sensor feedback).
    // The XX_var variable is a smart pointer for memory manage-
ment.
    MobilityGeometry::SegmentState_var pLaserSeg;
    MobilityGeometry::SegmentData_var pLaserSegData;
    int use_laser;
    // Request the interface we want from the object we found
    try {
        pLaserSeg = MobilityGeometry::SegmentState::_narrow(ptem-
pObj);
        if (pLaserSeg != MobilityGeometry::SegmentState::_nil())
            use_laser = 1;
    }
    catch (...)
    {
        use_laser = 0;
        pLaserSeg = MobilityGeometry::SegmentState::_nil();
        fprintf(stderr, "We didn't get the laser. Bummer!");
    }
}
```

```
 /
*****
*****/
/* New Stuff Here: Optional Contact sensor inputs for B21
*/
 /
*****
*****/
    fprintf(stderr, "Connect Base Bump.\n");
    // Build a pathname to the component we want to use to get
    sensor data.
    sprintf(pathName,"%s/BaseContact/Point",robotName); // Use
    robot name arg.
    // Locate the component we want.
    ptempObj = pHelper->find_object(pathName);

    MobilityGeometry::PointState_var pBumpPoint;
    MobilityGeometry::Point3Data_var pBumpData;
    int use_bump = 0;
    // Request the interface we want from the object we found
    try {
        pBumpPoint = MobilityGeometry::PointState::_narrow(ptem-
pObj);
        if (pBumpPoint != MobilityGeometry::PointState::_nil())
            use_bump = 1;
    }
    catch (...)
    {
        fprintf(stderr,"We didn't get base bumps. Bummer!");
        pBumpPoint = MobilityGeometry::PointState::_nil();
        use_bump = 0;
    }

    // Build pathname to the component we want to use to drive
    the robot.
    sprintf(pathName,"%s/Drive/Command",robotName); // Use robot
    name arg.
    // Locate object within robot.
    ptempObj = pHelper->find_object(pathName);
    // Find the drive command (we're going to drive the robot
    around).
    // The XX_var is a smart pointer for memory management.
    MobilityActuator::ActuatorState_var pDriveCommand;
    MobilityActuator::ActuatorData      OurCommand;
```

```
// We'll send two axes of command. Axis[0] == translate,
Axis[1] == rotate.
OurCommand.velocity.length(2);
// Request the interface we need from the object we found.
try {
    pDriveCommand = MobilityActuator::ActuatorState::_duplicate(
        MobilityActuator::ActuatorState::_narrow(pTempObj));
}
catch (...)
{
    return -1;
}

// Now, here is a loop that continually checks robot sensors,
// and sends new drive commands to make the robot follow.
// There are lots slicker ways to do this, but this is the
// simple example, so bear with us until later examples.
unsigned long index1; // Counts through sensor readings.
float haltdist = 0.2; // 20cm halts
float followdist = 0.9; // 0.9m follow distance.
int bumped = 0;
int seg_count = 0;
float mindist; // Computed minimum sensor distance.
float minfrontdist; // Computed minimum front distance.
float tempdist; // Computed temp dist value compared
to min/minfront.
fprintf(stderr,
    "***** Mobility Simple-Follow-2: Main Loop
*****\n");
while(1)
{
    if (pSonarSeg != MobilityGeometry::SegmentState::_nil())
        pSegData = pSonarSeg->get_sample(0);
    /
    *****/
    /* New Stuff Here */
    /
    *****/
    if (pLaserSeg != MobilityGeometry::SegmentState::_nil())
        pLaserSegData = pLaserSeg->get_sample(0);
    /
    *****/
}
```

```
        /* New Stuff Here */
        /
        *****/
        if (pBumpPoint != MobilityGeometry::PointState::_nil())
            pBumpData = pBumpPoint->get_sample(0);
        /
        *****/
        /* Now we have sonar+laser+bump data */
        /
        *****/

        // Process sonar data (you get back a set of line seg-
ments).
        // This shows how you can loop through all the segments
you
        // get back from the sonar source.
        mindist = 100000.0; // Set to "really large" dis-
tances.
        minfrontdist = 100000.0;
        seg_count = 0;
        // Find minimum front distance and minimum overall dis-
tance.
        for (index1 = 0; index1 < pSegData->org.length();
index1++) {
            // Compute segment lengths.
            tempdist = sqrt(
                (pSegData->org[index1].x - pSegData-
>end[index1].x)*
                (pSegData->org[index1].x - pSegData-
>end[index1].x)+
                (pSegData->org[index1].y - pSegData-
>end[index1].y)*
                (pSegData->org[index1].y - pSegData-
>end[index1].y));
            // Find the minimum length value.
            if (tempdist < mindist)
                mindist = tempdist;
            // Find the minimum front distance value. (Only
things in front
            // of the robot).
            if ((pSegData->end[index1].x - pSegData-
>org[index1].x) > 0.2) {
                if (tempdist < minfrontdist)
                    minfrontdist = tempdist;
            }
        }
    }
```

```

        seg_count++;
    }
    /
    *****/
    /* New Stuff Here */
    /
    *****/
    // Do the same thing for our new laser data.
    // Find minimum front distance and minimum overall distance.
    if (use_laser == 1)
    {
        for (index1 = 0; index1 < pLaserSegData->org.length(); index1++) {
            // Compute segment lengths.
            tempdist = sqrt(
                (pLaserSegData->org[index1].x-
                 pLaserSegData->end[index1].x)*
                (pLaserSegData->org[index1].x-
                 pLaserSegData->end[index1].x)+
                (pLaserSegData->org[index1].y-
                 pLaserSegData->end[index1].y)*
                (pLaserSegData->org[index1].y-
                 pLaserSegData->end[index1].y));
            // Find the minimum length value.
            if (tempdist < mindist)
                mindist = tempdist;
            // Find the minimum front distance value. (Only
things in front
            // of the robot).
            if ((pLaserSegData->end[index1].x-
                 pLaserSegData->org[index1].x) > 0.2) {
                if (tempdist < minfrontdist)
                    minfrontdist = tempdist;
            }
        }
        seg_count++;
    }
    /
    *****/
    /* New Stuff Here */
    /
    *****/
    // Are we bumping something?
    bumped = 0;

```

```

    if (use_bump == 1)
    {
        fprintf(stderr,"BMP");
        for (index1 = 0; index1 < pBumpData->point.length();
index1++) {
            if (pBumpData->point[index1].flags == 1)
                bumped = 1;
        }
    }
    // Show us what was found.
    fprintf(stderr,
        "Range Values: %d Minimum Dist: %f Min Front
Dist: %f\n",
            seg_count,
            mindist,
            minfrontdist);
    // Compute new drive command based on these distances.
    // Halt if we're too close or if we're bumped.
    if ((mindist < haltdist)|| (bumped != 0)) {
        OurCommand.velocity[0] = 0.0;
        OurCommand.velocity[1] = 0.0;
    }
    else if ((minfrontdist < (followdist + 0.2))&&
        (minfrontdist > (followdist - 0.2))) {
        OurCommand.velocity[0] = 0.0;
        OurCommand.velocity[1] = 0.0;
    }
    else if (minfrontdist > (followdist + 0.25)) {
        OurCommand.velocity[0] = 0.15;
        OurCommand.velocity[1] = 0.0;
    }
    else if (minfrontdist < (followdist - 0.25)) {
        OurCommand.velocity[0] = -0.15;
        OurCommand.velocity[1] = 0.0;
    }
    fprintf(stderr,"CMD: %f %f",OurCommand.velocity[0],
        OurCommand.velocity[1]);
    pDriveCommand->new_sample(OurCommand,0);
    // Was there a character pressed?
    if (mbyUtility::chars_ready() > 0) {
        // Stop robot.
        OurCommand.velocity[0] = 0.0;
        OurCommand.velocity[1] = 0.0;
        pDriveCommand->new_sample(OurCommand,0);
    }

```

```
        return 0;
    }
    else // Wait a small time before the next loop. 0.1 second, keep going.
    {
        omni_thread::sleep(0,100000000);
    }
}
return 0;
}
```

Makefile for simple_follow_2

```
#-----
-----
#
#       Makefile for Mobility example: simple_follow_2
#
#-----
-----
# Do you want every command printed, or just errors? (Q=Quiet)
ifndef Q
Q = @
#Q =
endif
# Sources to compile
CPP_SRCS =
C_SRCS =
# Libraries used by these test programs
#LDLIBS +=
# define the name of the executables (up to 3)
PROG = simple-follow2
# include the master Mobility library makefile
include $(MOBILITY_ROOT)/etc/scripts/cppexec.mf
```

Mobility In Action – More Realistic Examples

NOTE: The sample programs “square_and_circle” and “wander” discussed in the following sections are not yet fully developed and tested for Mobility 1.1, and may not be included in your distribution.

Let's say you have a mobile robot with some sonars, some infrared sensors and some tactile sensors. When a sonar detects its ping bouncing back to it off an object in its environment, the robot can ascertain the distance between itself and that object. The infrared and tactile sensors return similar information, based on their own characteristics.

In the following sections, you'll work with a robot demonstration program called `wander` that lets your robot wander about while avoiding running into obstacles within an 180-degree-arc heading of itself. A function within `wander` examines the current situation at any given instant, determining if anything is in the way, and modifying speed and heading accordingly.

But to actively avoid obstacles within a particular distance, calling this function just once is clearly insufficient. New obstacles may come into range at any time. You might call the function periodically, to continuously scan the environment and take evasive action as appropriate. Or, you might decide to call the function only when the robot has received new readings from its sonar, infrared or tactile sensors. To implement either scheduling scenario, use Mobility's `ActiveComponent`.

With `ActiveComponent`, you define a set of instructions to various components of the robot, and specify under what conditions each is to be invoked:

- on a schedule based on the elapsed time between invocations;
- upon triggering event, such as a sonar reading; or,
- using a simple timer.

The `ActiveComponent` takes care of invoking the function at the specified times. `ActiveComponent` will keep on invoking the function, on schedule, until either a module explicitly asks it to stop, or when you interrupt the program. The active component replaces the simple "main loop" you used in the `simple_follow` example.

In your robot system, many tasks must all be invoked repeatedly. With `ActiveComponent` directing traffic, though, your program can manage multiple asynchronous tasks invoked by multiple modules.

Square_and_Circle: A Mobility Sample Program

This sample Mobility program demonstrates how to use the `PoseController` to drive your robot in a square or circular route on the floor of your lab. It shows how to configure Mobility robot integration software for use with the low-level `PoseCon-`

troller and with Mobility’s built-in collision avoidance component called GuardedMotion. The combination of these two components offers a reactive, high-level interface to your robot. The organization of these components follows a layered model for combining software for robots. The PoseController and GuardedMotion together form a “layer” between higher level commands and the robot hardware by incorporating sensor data and generating commands based on reactive principles.

The PoseController component provides an interface of target points that lets you command the robot to head towards a set of target points (“waypoints”) in sequence. The PoseController considers only the target points and the current location of the robot. Without the help of GuardedMotion, the PoseController would run into obstacles. Working together, the PoseController and GuardedMotion provide a simple but effective low-level robot controller.

Square_and_Circle’s System Configuration

Filling Out Square_and_Circle’s Functionality

Wander: A More Interesting Mobility sample program

Wander’s System Configuration

Filling Out Wander’s Functionality

Designing a Mobility Program

A Mobility program consists of System Components, some written by you and some provided by iRobot and others. Mobility modules let you collect information from the robot’s hardware. Mobility components take care of managing robot resources and communication among all onboard programs and processors, so you don’t have to deal with that part of robot programming.

To write a Mobility program:

1. Derive some new component class(es) from the Mobility Class Framework, and add to or override built-in functions to make the robot do something new and

different from the default framework behavior. (This should not be difficult, as the default framework behavior does precisely nothing.) When you derive these new components, you specify their behavior by filling in functions that are called automatically by the class framework when certain user-defined computations are needed. You also write the initialization functions that set the initial state of your components.

2. Write a main program to create the `SystemModule` component for your new program. Add the new components you've written to the `SystemModule`. You might also want to set some default property values for your components when you create them.
3. Write code to initialize your `SystemModule`, which will initialize any components you've inserted into the module.
4. Write code to activate your `SystemModule`. This will initialize the ORB library and connect your program with the top-level naming service for Mobility so your program can be viewed and managed from the Mobility System Manager. This will also activate any `ActiveComponents` that are part of your new code.

MOM — The Mobility Graphical Interface

The Mobility Development Environment offers a set of tools that let you control Mobility-supplied objects, and your own objects, through a set of common graphical interfaces. These tools also let you visually debug your ideas and algorithms faster and more easily than ever before. The Mobility tool you'll use most often is also the core of the Mobility tool interface. It's called MOM, the Mobility Object Manager.

Mobility also provides direct support of multiple robots, and with MOM, you can easily visualize several robots running at the same time — finally getting a handle on those tricky multi-robot coordination problems.

NOTE: Mobility fully supports distributed, decentralized robot control software. However, Mobility supports a centralized management structure to simplify the process of configuration, management and development of multi-robot systems.

MOM: An Overview

MOM is a graphical user interface that lets you observe, tune, configure, and debug the components of your Mobility robot control programs -- as they are running. With MOM, you can launch programs, create objects, edit object properties, connect and configure objects and control which objects are active. MOM also lets you

launch a variety of object viewers that provide visualization of your robot's actuators, sensors, algorithm outputs and debugging information, all from a central management point.

MOM is your primary tool for interacting with your Mobility robot programs. For example, you can use MOM to teleoperate your robot. MOM is written in Java and communicates with your Mobility components through CORBA.

MOM depends on a few "background" programs running on computers that participate in the Mobility system. The scope of a Mobility system is contained within objects that are accessible from a top-level Naming Service. The location of this naming-service is not fixed; there are no hard-coded "port numbers" or multicast schemes in Mobility.

Because MOM and the Mobility objects controlling your robot are a truly distributed object system, there are many ways to arrange your programs. At least the Mobility base server must run on the computer physically connected to your robot hardware. Other programs containing Mobility objects may run on the same or other computers. MOM itself usually achieves its best performance running directly on your desktop or lab computer, although it can also run on another computer and display on your desktop by using the X window protocol. The Naming Service can run on the robot computer, your desktop or lab computer, or any other convenient computer.

You can easily and simply run multiple Mobility systems on the same set of computers at the same time without fear of conflicts. How? You simply start each system with a separate workspace of distributed objects. Each workspace is defined by its own separate Naming Service.

Understanding MOM's Environment

Three key aspects of MOM's environment are explained in this section:

1. environment variables;
2. the Naming Service; and
3. some of the different ways Mobility programs can be distributed.

Many of Mobility's programs and scripts need to access files and tools from the Mobility distribution. While Mobility is normally installed in `~mobility/mobility-b-1.1.0/` (the exact release letter and numbers vary, of course), you may install it elsewhere.

Mobility programs and scripts read the `MOBILITY_ROOT` environment variable to find the root directory of the Mobility software distribution. In much of the iRobot-supplied Mobility documentation, we refer to directories and files in the Mobility distribution by writing `MOBILITY_ROOT` in place of the actual installation directory.

There are several important environment variables that must be set properly in order to develop and run software using Mobility. iRobot provides a Bash script: `MOBILITY_ROOT/etc/setup` that sets them all.

You must use the `source` command to read this script. The script expects the environment variable `MOBILITY_ROOT` to be set already, and it sets `JAVA_HOME`, `LD_LIBRARY_PATH`, `CLASSPATH`, and `PATH`. The setup script is designed to work properly if `source`'d multiple times.

In the CORBA distributed object environment, one object can communicate with another object only if it has the other's `Object Reference`. A program that creates several objects can give them one another's references, but for objects in separate programs (such as the robot base server and MOM) there is a bootstrapping problem: How does an object reference for any object in one program first get to another program?

The CORBA standard solves this problem with the `Naming Service`. Important objects in each program register themselves with the `Naming Service`. Any program can query the `Naming Service` to get the object reference registered with a (string) name. MOM queries the `Naming Service` for all registered objects. However, each program still has to obtain the object reference to the `Naming Service`.

When a CORBA object reference is written out, it is called an IOR (Interoperable Object Reference). An IOR is about 250 bytes of hexadecimal. When the Naming Service program starts up, the IOR of the Naming Service is written to a file. In order for Mobility programs to interoperate, they must all obtain the IOR of the same Naming Service. In a distributed, networked environment with some nodes (the robots) connected by radio, shared filesystems are not very reliable. Instead, Mobility programs use a URL to obtain the file containing the IOR. Mobility programs read the URL for the CORBA Naming Service from the environment variable `$MOBILITY_NS`. By convention, the URL takes the form

```
http://<hostname>/~mobility/NamingService
```

For example, if the Naming Service runs on the host `robbie`, using Bash you would give the command:

```
prompt> export MOBILITY_NS=http://robbie/~mobility/NamingService
```

prior to running any Mobility program.

It often makes a great deal of sense to set the `MOBILITY_ROOT` and `MOBILITY_NS` environment variables, and to source `MOBILITY_ROOT/etc/setup` from your home directory's `.bashrc` file. The `mobility` user account on iRobot factory-installed computers is set up this way, as described in the next section.

The base server must execute on the computer that is physically connected to your robot. Most iRobot robots have a single on-board PC running Linux, and that is the computer that must run the base server. On the iRobot Magellan robot, the computer may be a desktop computer connected to the robot by a radio RS-232 link.

In the case of a B21r or ATRV robot with more than one computer, check the User's Guide for your robot to determine which one is connected to the base.

NOTE: If Mobility is not installed on the robot computer, it is possible to create a “base-only” configuration and copy it to the robot computer. See Appendix A “Installing Mobility” for details.

This physically-connected computer that executes the base server is called the robot computer.

The base server and MOM need not execute on the same computer. The computer that MOM runs on is the display computer.

- MOM must run on a computer with a graphical display suitable for an interactive GUI program, such as a desktop computer.

- MOM must be able to contact the base server via the Internet. They may run on the same computer, or they may run on computers on separate continents. It is likely to be problematic if there is a firewall between them.
- If you want to run MOM on a computer that does not have Mobility installed, for example, a desktop computer running Windows 95 or Windows NT, you can create a “MOM-only” configuration and copy it to the display computer as described in Appendix A “Installing Mobility”.

NOTE: MOM can actually run on one computer and display on another by using X Windows. The “display computer” referred to in this document is always the one that is actually running MOM, not the one to which MOM’s display happens to be directed.

iRobot Factory Pre-installed Configuration

When Mobility is installed at iRobot's factory, the mobility user account on the robot computer has its environment set up automatically in its `.bashrc` file. It expects the Naming Service to be run on the robot computer. Its `.bashrc` contains something like the following lines:

```
export MOBILITY_ROOT=~/.mobility-b-1.1.1
export MOBILITY_NS=http://`hostname -s`/~mobility/NamingService
alias base=b14rserver
if [ -f $MOBILITY_ROOT/etc/setup ]; then
    source $MOBILITY_ROOT/etc/setup
fi
```

The first line sets the `MOBILITY_ROOT` environment variable to the root of the tree of installed Mobility software and tools. The second line sets the `MOBILITY_NS` environment variable to the conventional URL on the robot computer. The third line creates the command “base” to run the base server for the specific robot model. The fourth through sixth lines source the Mobility setup script, but won’t cause an error if the Mobility software is not where it is expected.

The instructions in Chapter 2 “A Tour of Mobility With the MOM Graphical Interface” tell you how to run the Naming Service, the Mobility base server, and MOM all on your robot computers. (In the terminology of the previous section, the robot computer and the display computer are the same.) This configuration will work, but

most likely with poor performance because of the X Window traffic running over the radio ethernet from the robot computer to your desktop where MOM displays.

To get good performance from MOM, it must run on the computer upon whose screen it displayed. Appendix A “Installing Mobility” describes in detail how to install Mobility — either the full release or just MOM — on your desktop computer. Then, on your display computer, set `MOBILITY_NS` to the same URL that the robot computer uses and run MOM again.

Invoking the Naming Service

The Naming Service daemon program is invoked with the `name` command. It is a script in `MOBILITY_ROOT/etc`.

It reads the environment variable `MOBILITY_NS` as well as the ones set by the `setup` script. `name` has several command-line options:

- `-i` print out the IOR of the Naming Service
- `-r` kill, then restart the Naming Service daemon
- `-k` kill the Naming Service daemon

Unless `-r` or `-k` is given, `name` checks to see whether the Naming Service daemon is already running on this computer. If so, it prints a message to that effect and exits. Otherwise, it starts the Naming Service daemon. You must run `name` on the same computer that is named in the `MOBILITY_NS` URL.

Occasionally, when a robot program aborts, its objects do not unregister themselves from the Naming Service. The Naming Service will then hand out an IOR to a non-existent object. The Java ORB is still somewhat experimental and does not handle this error gracefully. If your Naming Service has this kind of garbage in it, MOM will most likely not find the actual objects in the Naming Service. iRobot recommends three possible ways of dealing with this problem:

1. Start the aborted program again. It will register a new object with the same name, replacing the invalid object reference.
2. Shut down all Mobility programs using the Naming Service and restart the Naming Service with the `name -r` command.

3. Try running the `nsclean` Mobility utility. This utility is intended to remove invalid object references, but it doesn't always work. It takes no arguments and reads the `MOBILITY_NS` environment variable.

If you're working with multiple robots together, all the robots (actually, all the Mobility programs) must use a single Naming Service. In this situation it is usually best to run the Naming Service on a computer that is not in one of the robots, so that there is only one radio link hop between any program and the Naming Service.

NOTE: In a robot laboratory with multiple independent researchers working concurrently with multiple robots, it is sometimes beneficial to isolate the separate development efforts from each other. Run a separate instance of the Naming Service daemon for each development effort.

While the `name` script reads and uses `MOBILITY_NS`, it's not able to distinguish multiple Naming Service daemons running on a single computer even if they write their IORs to different files. In the unlikely event that you need to do that, you will have to directly invoke the Naming Service daemon yourself. All other Mobility software accesses the Naming Service by its IOR, so as long as multiple instances of the Naming Service daemon write their IORs to different locations, only the `name` script would get confused.

If you are running multiple Naming Services on computers that share filesystems, different URL's might end up referring to the same actual file. Make sure that each Naming Service ends up writing its IOR to a different physical file.

The following example uses Bash shell syntax, assumes Mobility was installed in `/home/mobility/mobility-b-1.1.0`, and assumes you want the Naming Service to write its IOR to `http://computer/~mobility/NamingService`.

```
prompt> export MOBILITY_ROOT=/home/mobility/mobility-b-1.1.0
prompt> export MOBILITY_NS=http://computer/~mobility/Naming-
Service
prompt> source $MOBILITY_ROOT/etc/setup
prompt> name -i
```

If the Naming Service is already running, `name` will print a message to that effect and harmlessly exit.

The Base Server

The base server for your robot is a program located in `$MOBILITY_ROOT/bin`, named after your robot model as follows:

TABLE 4 - 1. Mobility Base Server Program For Type of Robot

Robot Type	Mobility Base Server Program Name
ATRV	atr2server
ATRV-Jr	atrjserver
ATRV-micro	uatrvserver
B21r	b21rserver
B21	b21server
Magellan	magellanserver
B14r	b14rserver
B14	b14server
Pioneer AT	pioneerserver
Pioneer	pioneerserver

NOTE: Early ATRV robots use a different base server, “atrserve”.

Custom robots generally have custom base servers.

When Mobility is installed at the iRobot factory, the `mobility` user account arranges that the proper base server will run if you give the ‘base’ command.

The base server prints out a lot of diagnostic messages. Often these messages give the appearance of an error even during normal operation. The most common situations are:

1. When the base server attempts to register with the Naming Service, if a previous run of the base server exited uncleanly leaving an invalid object reference, an error message is printed but the base server goes on to replace the invalid reference with a reference to itself.

2. When the base server starts communicating with the robot firmware (in most models, the rFLEX controller), there are often a few error messages until the two computers get synchronized.
3. In some robot models, it is normal for there to be occasional communication errors.

All base servers to date communicate with the robot hardware through a single RS-232 connection. The Linux device for this connection can be specified on the command line. This option is primarily useful on models like the Magellan with no on-board computer. The RS-232 radio modem may be connected to any serial port on the PC, but the base server must be informed of the corresponding Linux device.

The following example runs a Magellan base server, assuming the robot computer's name is "robot" and the radio modem is connected to `/dev/cur2`.

```
prompt> export MOBILITY_ROOT=/home/mobility/mobility-b-1.1.0
prompt> export MOBILITY_NS=http://robot/~mobility/NamingService
prompt> source $MOBILITY_ROOT/etc/setup
prompt> magellanserver -deviceport /dev/cur2
```

Starting Up MOM

Like the base server, MOM must be given the URL of the Naming Service. This can be done in the same way as the base server. The following commands will run MOM on the computer named "desktop", with the Naming Service running on the computer named "robot" and the Mobility software installed in `/home/projects/mobility`.

```
prompt> export MOBILITY_ROOT=/home/projects/mobility
prompt> source $MOBILITY_ROOT/etc/setup
prompt> export MOBILITY_NS=http://robot/~mobility/NamingService
prompt> mom
```

It takes a few moments for the MOM screen to come up as the Java classes load, the ORB initializes and connects to the Naming Service, and the MOM object hierarchy view appears.

MOM's Graphical Interface

To start the MOM interface, review the instructions in Chapter 2 “A Tour of Mobility With the MOM Graphical Interface”.

When you start up MOM, the Hierarchy View Window appears (see Figure 4 - 1, “MOM’s Object Hierarchy View Window,” on page 4 - 10:



FIGURE 4 - 1. MOM’s Object Hierarchy View Window

The object hierarchy view, or just “object view” for short, is very similar to familiar GUI programs that show directory hierarchies. The single root node you see in the object view represents the base server program and is named for the model of your robot.

Click on the dot to the left of a node or double-click on the node itself to toggle the display of its children. Explore the hierarchy below the root node to see the components of the base server program. Individual components can be named, just like files in a directory hierarchy, with a full path name from the root node. For example, in a Pioneer base server, the component object that monitors the RS-232 port to the robot is named

`/Pioneer/Hardware.`

Clicking the right mouse button on a component object pops up a menu of all the viewers known to MOM that can display something about that object. The two most useful for robot teleoperation are described next.

Range (Sonar) View

To see graphically the live output of your robot's sonar sensors, expand the object hierarchy:

```
/<robot>model>/Sonar/Segment
```

Click your right mouse button on Segment and select Range View from the popup menu. You'll see a dynamic, scaled display of the sonar sensors. The sonar cones are drawn at 50 pixels per meter. The Java program polls the base server for sonar data. Buttons across the top of the MOM Window allow you to control the polling behavior.



FIGURE 4 - 2. Selecting MOM's Range View Window

CAUTION: Safety First! When running your robot, always be prepared to press the Emergency Stop button to prevent unintentional damage to the robot, other lab equipment, or personal injury. Be aware that a radio link problem could cause the robot not to receive a software “stop” command.

To tele-operate your robot, expand the object hierarchy so you can see the node

```
/<robot>model>/Drive/Command
```

Click the right mouse button on Command and select Drive View from the popup menu. The Drive View and Range View windows will display (see Figure 4 - 3, “MOM’s Drive View Window and Range View Window,” on page 4 - 12).



FIGURE 4 - 3. MOM’s Drive View Window and Range View Window

The grid area in the center operates much like an iRobot joystick control. The vertical axis of the grid represents translation: up for forward, down for reverse, with no movement in the center. The horizontal axis represents rotation: left to rotate left and right to rotate right. Place the cursor at or very near the center of the grid and hold the left button down.

Slowly and carefully move the cursor while holding the left button down, and the robot will move. Let up on the mouse button and the robot will stop.

The Object Hierarchy

The object view in MOM shows the hierarchy of your Mobility software component objects. A root node in the default tree represents a program, its children represent its major software components, and their children represent subcomponents, and so forth.

For example, in the base server program the sonar sensing system is a major component, with subcomponents representing three forms that the data from each sonar sensor can take:

Raw	A distance reading given in meters.
Point	An (X, Y, Z) point in robot coordinates of the detected object.
Segment	An (X, Y, Z) origin and (X, Y, Z) endpoint of the region of free space from the sonar sensor to the detected object.

Through these three objects, the sonar data is available in three different formats. A component that requires sonar data obtains it from the object that provides it in the desired format. The Sonar component is responsible for updating the data in the three subcomponents as new readings become available.

Normally, all the components communicate with one another through the CORBA interfaces, so in principle each component could be alone in its own program. However, components within the same program interact with no communications overhead. Thus, performance is typically what drives the assignment of component objects to programs.

The robot control software you write can act as a client to the provided Mobility software components, or, you can write full-fledged components of your own. A client-only program will be invisible to MOM, whereas components can take advantage of numerous features of MOM such as properties, active objects, and hot-pluggable connections.

Properties

A Mobility component can have a set of properties that are visible to MOM. If you implement all the tuning parameters of your algorithms as Mobility properties of your software components, you can tune the parameters at runtime and (in a forthcoming release of Mobility) save and restore them using Mobility tools. The MOM Property Viewer allows you to access property values at runtime. There are three ways you can use properties: read-only, initialization-writable, and fully dynamic.

Read-only: A component can use properties to provide additional information to its clients. For example, the standard sonar sensing and laser range sensing components provide the divergence property that gives the angle of sensor beam divergence in radians, as a floating point number. Such a property is only readable by clients. The Range View client uses this property value to correctly depict the free-space cones.

Initialization-writable: Active components have properties that may be set before the component starts running, but cannot be modified thereafter. For example, you

could write a metric-grid-based navigation system to have a settable grid resolution, but for performance reasons you choose not to allow the resolution to be adjusted once the navigation system starts running. Such a property would be writable only during component initialization and read-only thereafter.

Fully-dynamic: A dynamic property can be set at any time and the new value takes effect immediately. For example, a wall-following algorithm could use a dynamic property to define the distance it maintains from the wall.

Debug Output

By convention, most Mobility objects that can contain other objects (in other words, objects that support the `ObjectContainer` interface) contain an object named `Debug`. You've probably already seen one or two of these in the MOM object hierarchy viewer. These `Debug` objects are very much like the objects representing sensor input data. They are updated periodically with new data you can observe in MOM with a viewer. They implement the same underlying `DynamicState` interface.

A `Debug` object's data is in the form of strings of characters, (Other sensor data could also be character strings, but that would be unusual.) What makes `Debug` objects unique is that they are used by software components to output debugging information. Instead of writing to UNIX standard error or a DOS console, Mobility software components write debugging information to their `Debug` object. The MOM viewer for a `Debug` object is a scrollable text window in which you can read its messages. You can create and use `Debug` objects from software components that you write.

Coming Soon

The following sections highlight some of the features you can look forward to in future releases.

NOTE: Release 1.1 of Mobility does not completely support many designed interfaces and capabilities, in both the Class Framework and in MOM.

- Creating and Running Objects
- Active Objects
- Hot-Pluggable Components
- Adding a Viewer
- Loading Configurations

- Saving Configurations

NOTE: Do you have an idea for a useful feature that is not included in the current release of MOM or Mobility? As always, iRobot welcomes your feedback to help guide our development efforts. Real World Interface appreciates user feedback and will consider for future releases, all submitted ideas for enhancements and features. Just drop an email to support@rwii.com.

Creating and Running Objects

The MOM can communicate with objects that support process management as well as with object factory interfaces. Thus, you can dynamically add new elements to your running robot system from MOM. For example, you could configure a set of robots for a demonstration, create and set parameters of a demo program object all from the Mobility Object Manager interface, and then activate the demo object to show off your work.

Active Objects

Active Mobility components each have their own thread of control. Before an active object gets its own thread, its initialization-only properties can be written. Once it is activated, those properties become read-only and it receives a thread of its own with which to carry out its computation. A fully implemented active object can be deactivated, properties and connections modified, and subsequently reactivated any number of times, all through the MOM. This lets you rapidly experiment with combinations of several modules or robot behaviors simply by switching them on or off from your MOM console.

The robot base servers contain active objects that monitor the robot hardware interface(s) and that push sensor data to their other major software components that each manage a kind of sensor data.

Hot-Pluggable Connections

A Mobility component can “advertise” that it expects to use another component of a particular type. This usage can be optional or required, and can be either a source or a sink of data.

Adding a Viewer

Additional, user-written viewers can be added to MOM without modifying or recompiling MOM. This feature will be improved and documented in a subsequent release of Mobility. If you find it necessary to add a viewer of your own to MOM in the meantime, please contact iRobot software support for advice.

Loading Configurations

Because all Mobility objects support a common set of core interfaces, you can dynamically load a configured robot system from persistent storage. MOM lets you select a robot system configuration file and load that file into a robot system. The load process automatically launches robot servers, uses factory interfaces to create objects and initializes the objects from the data stored in the configuration file.

Saving Configurations

Because all objects in the Mobility system support a common set of core interfaces, MOM can ask each object in the system to save its persistent state into a file by using the externalization interfaces. A saved system configuration can be restored later. After spending several hours tuning the robot demo, it's very convenient to be able to store this information for a presentation of your work later.

The combination of dynamically editable properties and centralized configuration management provide an environment that encourages a wide range of experimentation with algorithm parameters while providing a straightforward way to manage and compare multiple configurations of your system.

Mobility Programming With Class Frameworks (C++ and Java)

The Mobility Class Framework Model (Language Dependent C++/Java)

The Mobility Class Framework (MCF) is a set of language-specific classes that provide the implementation for the core functionality of your Mobility system. The MCF provides base classes that implement most of the interfaces needed for integration with Mobility robot integration software. The class frameworks provide the basic implementation of the MobilityCore, MobilityData and MobilityComponents interfaces for all Mobility-defined robot objects and interfaces.

NOTE: Release 1.1 of Mobility provides the Class Framework in C++ only.

The class framework provides much of the implementation required to build your project using the Mobility Core Object Model. You can reuse that code by inheriting from the framework base classes. The class frameworks help eliminate redundant programming effort in writing your own Mobility object implementations

The class framework provides a “working skeleton” of the system you can customize to create you own Mobility hardware servers, behaviors, perceptual routines and data objects. From the framework base classes, you derive your own classes to specialize or extend the functionality of the system for your application.

NOTE: Your derived classes will have access to some of the state of the framework base classes, but the base classes allocate storage and manage this state. Therefore, if you interfere with this internal base-class data structure, you might create problems.

In addition to implementations for standard interface methods, framework classes define additional state and methods that are specific to the framework implementation. These data types and methods make it easier to write your own derived classes.

The Mobility Class Framework provides four methods by which your derived classes may extend or specialize the operation of the framework:

1. Interface Methods
2. Helper Methods
3. Template Methods
4. Hook Methods.

Think of these methods as the interface between Mobility-supplied base classes and your own code.

Interface Methods

Interface Methods are virtual methods that directly implement the interface signatures defined in the Mobility Robot Object Model (MROM) by the IDL files. Your derived classes may override these methods to directly implement certain functions required of Mobility objects. However, most of these functions are implemented by the MCF and the implementations of these functions use the other three methods to provide more portable, safer extensions for base class functionality. Most of the time you don't have to override or implement these methods directly.

Helper Methods

Helper Methods are called by derived classes to access and/or manage part of a base class state. They are essentially utility functions used to simplify method implementations for derived class implementations. Helper methods are defined and implemented by the base classes in the Mobility class framework itself.

Helper methods provide accessors for state that is managed by the base classes so that derived classes can safely access base class data without directly manipulating the internal base-class owned storage. Each Mobility class framework base class

defines additional helper methods that correspond to the new state managed by that base class. The implementation of base classes does not normally override helper methods, but rather calls these base class methods directly.

Template Methods

Template Methods are virtual methods called at particular points within the base class implementation of Interface Methods to obtain data or allow customization of handling from within derived classes. Derived classes normally provide implementations that override the template methods of their base class. Using Template Methods is like “filling in a form” for your implementation.

Hook Methods

Hook Methods are virtual methods invoked by the base class implementation to provide an alternate implementation of an Interface Method. When the framework calls Hook Methods it is delegating nearly all the functionality required for the implementation of an Interface Method to a derived class. The default implementation of base class methods generally provides a kind of “null operation” for hook methods. If you want to override the behavior of the framework, provide new implementations for the hook methods in your derived objects. Think of using Hook Methods as a way to extend the processing of the framework to cover new cases by deriving classes from the framework base classes.

Some of the data structures visible to derived classes are lists of data whose storage is owned by base classes. Base classes also manage the state of the lists. These lists are implemented using language-specific list mechanisms.

In the case of C++ these lists are templates that provide an iterator for searching through the list. While it is possible to directly use the lists maintained by base classes in a derived class implementation, you shouldn’t manipulate the lists directly. Instead, use a combination of a Helper Method and a Template Method to “iterate through the list” and perform operations on each element of base class state.

Building on the Mobility Class Framework

Mobility’s object-oriented class framework is a set of classes that provides base implementations of elements of the Robot Object Model.

To build robot software for Mobility, simply:

1. Write a subclass of one of the framework objects; and
2. Extend and/or override the built-in behavior of that class to implement the object for a new sensor, actuator, behavior, data store or perceptual process.

Simple programming tasks thus remain simple within the relatively complex Mobility software system.

For each Mobility Component there is a framework base class that implements the core interfaces for that component. The framework class has implementations for every standard interface on the component itself. When you derive a class from this framework base class you can provide new implementations to replace the standard functionally, or rely on the base class for the implementation.

The implementation of the class framework is separated from the definition of the object model. Thus, you can program every aspect of robot software yourself, conforming only to the Robot Object Model, and still remain completely compatible with the other elements of your Mobility system. You can develop Mobility-compatible software modules in any programming language for which a CORBA 2.x IDL interface is available.

The Elements of Mobility Robot Integration Software

Mobility robot integration software comprises the following:

- Basic Mobility Tools (Java User Interface Programs)
- Basic Mobility Objects (C++ Programs)
- A C++ Class Framework (C++ shared library and headers)
- Interface definitions (IDL files and shared libraries)
- Support Tools and Utilities (C/C++ Programs)
- An Object Request Broker (basic shared libraries)

- An OS Abstraction Layer (basic shared libraries)

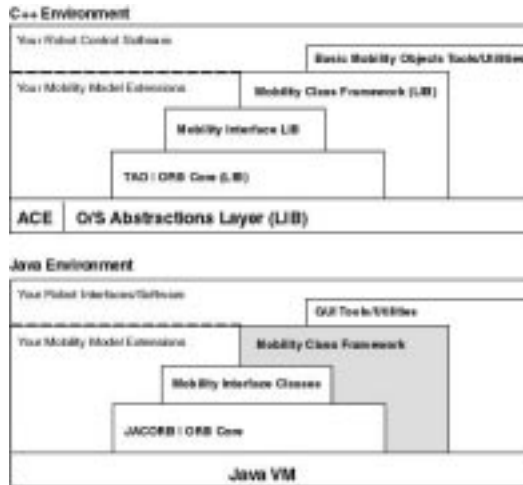


FIGURE 5 - 1. Mobility in the Context of the C++ and Java Programming Environments

Mobility Tools: Robot Tools and User Interfaces

Mobility's basic toolkit provides a GUI, or graphical user interface, to the basic Mobility objects. This GUI is called MOM, the Mobility Object Manager. Using MOM, you can drive your robot under teleoperation, view sensor feedback, manage the state of objects, configure and tune the parameters of objects, view debugging feedback from objects and generally keep a handle on your robot system. See Chapters 3 and 5 of this Manual for more details on MOM.

The Basics: Robot Components and Interfaces

In Mobility robot integration software, robot hardware, behaviors, plans, and internal representations are all represented and programmed as software components. These components each support a common subset of the Mobility interface and are called Mobility Components. Some Mobility Components support additional interfaces that allow them to contain other objects, execute on their own thread, or per-

form other functions. Specific Mobility Components may support new special interfaces that pertain to individual pieces of functionality.

You may be familiar with the concept of client-server computing. Several available robot development environments take the client-server approach. But Mobility is different. Everything in Mobility, including each program you write, is both a full client and a full server. A Mobility object is both a client of other objects and a server for other objects.

In distributed object systems like Mobility the terms “client” and “server” apply only to the invocation of a particular method. During that invocation the object that invoked the method is the client of the method and the object that is executing the method is the server.

Interface Definitions

Mobility’s interface definitions are a set of files written using the CORBA standard interface definition language (IDL). This language allows definition of an object system and interfaces for that system in a way that is independent of particular computing platforms or programming languages. Of course, the actual implementation of these interfaces depends on a particular platform and programming language, but by defining interfaces using standard IDL we can ensure that other implementations will work with objects that we implement, as long as they are based on the same interfaces.

In release 1.1 of Mobility, iRobot supports tools for C++. There are CORBA standard ORB implementations available for LISP, so LISP users can use Mobility robot integration software by compiling the Mobility IDL files using a LISP-based ORB.

Object Request Broker (ORB)

Mobility uses CORBA to communicate among objects. The implementation of CORBA used by Mobility has a shared library called the Object Request Broker (ORB). The ORB is a communication management library that allows transparent access to objects in different address spaces on the same or other computer systems. Mobility also includes interface libraries that are compiled versions of the interfaces that define the Mobility Core Object Model.

Mobility’s interface libraries use the ORB libraries to allow your programs to communicate with other programs, for example, with the robot manager. When you use

objects local to your process, there is no additional overhead for CORBA communications; the ORB library is not invoked in this case

O/S Abstraction Layers

Mobility is based on an ORB that is, in turn, based on an OS abstraction layer. This OS abstraction layer provides a high degree of portability to many of the core elements of Mobility, including the ORB itself and the Mobility C++ class framework.

Mobility's portability layer will allow support of a wide variety of operating systems, including Windows/NT.

NOTE: In Release 1.1 of Mobility, iRobot supports only the Linux OS

Mobility Robot Integration Software Overview

Mobility Robot Object Model (Language Independent)

The Mobility Robot Object Model defines the interfaces and objects needed to represent and manage robot software as a set of concurrently executing, distributed software components. These components represent software abstractions of robot hardware and behavior. The Mobility Robot Object model (MROM) is defined using an object-oriented approach that is in turn based on the CORBA 2.x object model. The interfaces of the MROM are defined in a programming-language neutral interface language called Interface Definition Language (IDL).

Robot Object Model Overview

The Robot Object Model describes a robot as:

“a hierarchical collection of object instances that provide interfaces to each component of the robot system.”

The top of the Robot Object Model is the CORBA 2.x standard naming service. This naming service (also referred to as a name server), allows your software to access the many elements of multi-robot software systems. The top level name server contains a directory of robot objects and shared support objects, such as maps shared by a robot team or a blackboard for multi-robot coordination.

Each robot has its own SystemModuleComponent in the Robot Object Model. The SystemModuleComponent provides a set of interfaces that allow organized access to all the components of each robot.

Each SystemModuleComponent contains a set of SystemComponents. SystemComponents serve as handy abstractions of

- robot hardware;
- robot software;
- robot behaviors;
- data stores; and
- perceptual processes.

Typical System Components a robot might include odometry, tactile sensors, sonar sensors and actuators. If your robot has a laser scanner, its SystemModuleComponent would be configured with an additional laser sensor component. If your robot has an arm, its SystemModuleComponent would be configured with an arm actuator component.

Through the interfaces provided by the SystemModuleComponent, you can add and remove services, and discover services dynamically at runtime.

Mobility's definition of a robot as “a collection of dynamically connected objects” makes it easy to reconfigure robot systems with new hardware and to integrate a wide variety of modules into the overall software system. And because all low-level robot hardware abstractions are explicitly specified, your robot hardware simulators are guaranteed 100% code compatibility between actual robots and simulations.

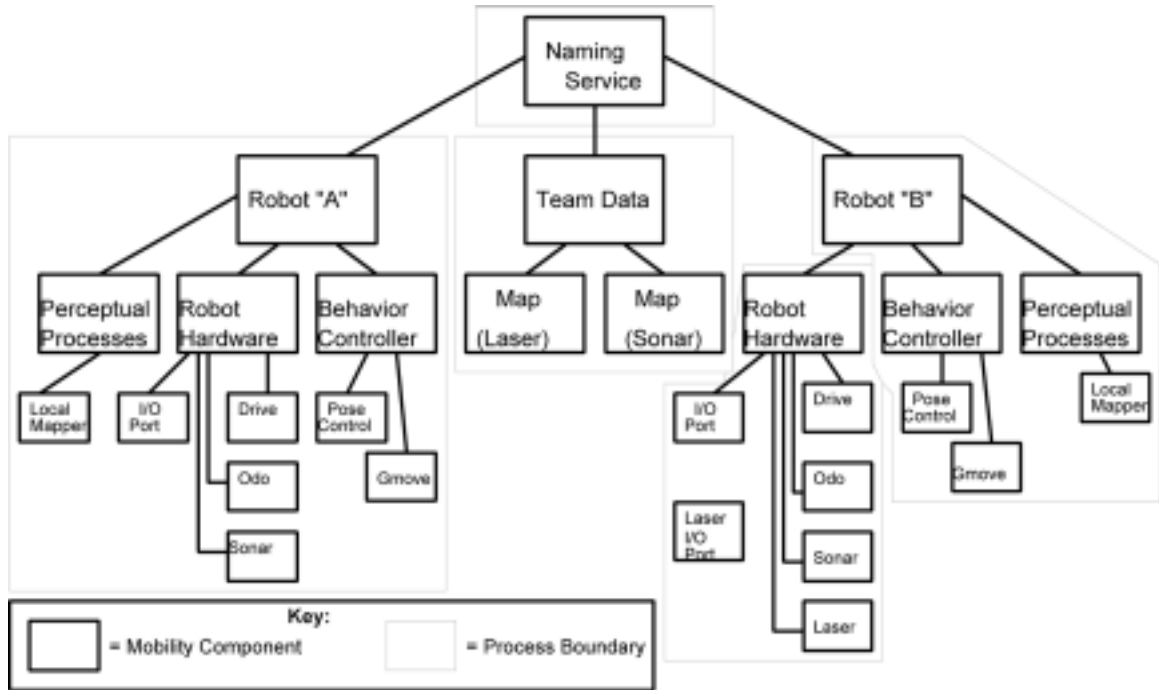


FIGURE 6 - 1. An Example Mobility Robot Software Setup

An Example Mobility Robot Control System

Mobility robot integration software includes an implementation of the modules and components needed to access all iRobot robot hardware from other Mobility modules. Many robot accessories offered by iRobot are also supported by components. These basic robot components provide abstractions of the robot hardware that support a wide variety of robot control algorithms, both high and low-level.

All basic components are written using the same Mobility Robot Object Model and the Mobility C++ Class Framework you'll use whenever you write C++ modules for Mobility robot integration software. These components manage all details of serial packet protocols, talking over the access bus to sonar sensors and dealing with robot geometry information so that you can focus on developing higher-level control software with some degree of cross-robot portability.

Other components provide desirable behaviors for robots, such as “avoid collision”, “wander”, “follow walls” and “go to point.” These can be combined to form a basic reactive controller for any iRobot robot.

The Mobility Robot Object Model (MROM) is defined using the CORBA Interface Definition Language (IDL). IDL allows sets of data structures and interfaces to be defined without regard to platform or programming language. This type of abstraction completely separates the interfaces of objects from the implementations of those objects.

IDL also permits the definition of compound interfaces (interfaces that are the union of several other interfaces). Therefore, it's possible for a particular implementation of an object to support multiple, different interfaces at the same time. The structure of these interfaces and compound interfaces is analogous to the structure of classes and derived classes in object-oriented programming languages.

Figure 6 - 2, “Mobility Class Diagram: Core and Components,” on page 6 - 5 shows part of the class/interface structure of Mobility. Studying this diagram will help you understand the next several sections, which define the interfaces supported by the objects used to build robot control software in Mobility.

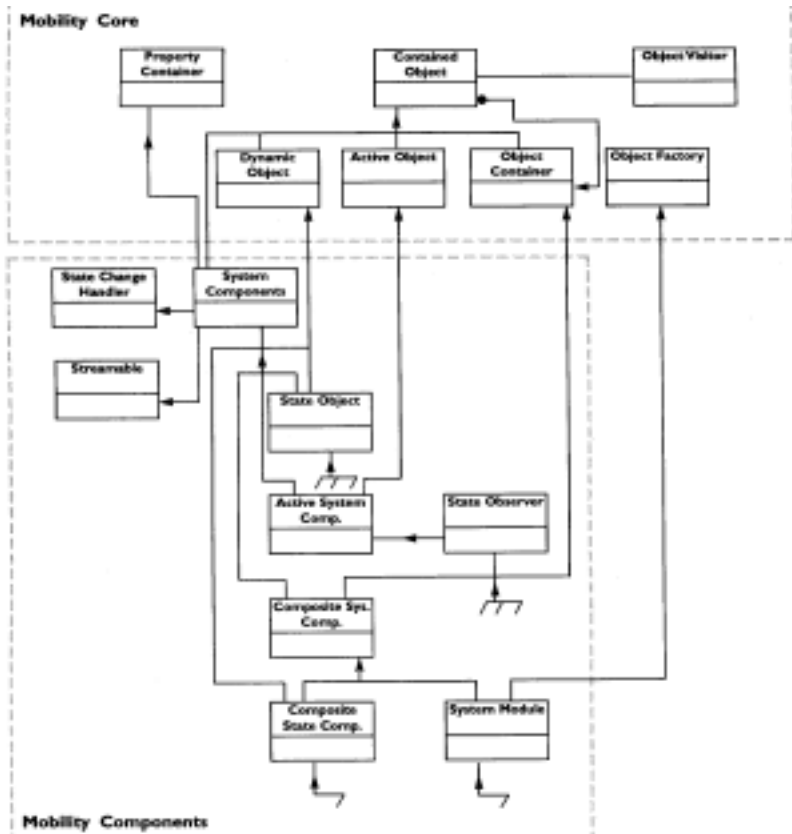


FIGURE 6 - 2. Mobility Class Diagram: Core and Components

The Mobility Core Interfaces

MobilityCore Interfaces are the backbone of the Mobility software system. These interfaces provide for

- location of components;
- organization of components into hierarchies;
- property management; and
- active objects.

Most of the elements of the Mobility software system (including components you write yourself) are `MobilityComponents`. `MobilityComponents` are objects that support a specific combination of the `MobilityCore` interfaces and will be described later. The following descriptions describe each of the `Mobility Core Interfaces`.

Contained Objects Interface

The `ContainedObject` Interface is supported by all `ContainedObjects`, objects that

1. have explicit identity;
2. can be located in containers; and
3. know that they are part of a container structure.

Other objects can be put into containers, but they will not be able to locate their own container.

Each `ContainedObject` has a self-describing data structure called the object descriptor. The object descriptor structure describes each object contained within a `Mobility` container. The object descriptor identifies

- the class of the object;
- any special service parameters;
- the instance name of the object;
- a list of `Mobility` interfaces supported by the object;
- a documentation string, a simple description of the object; and
- the object's `ObjectStatus`.

The `ObjectStatus` describes the basic states for all `Mobility` objects.

1. An object is created in the `Uninitialized`. state
2. An object's creator must initialize an object before use, causing the object to enter the `Initialized` state.
3. An initialized object may become an `Active` object, that is, may receive its own thread control.
4. When an `Active` object is stopped or waiting for synchronization, it is in the `ActiveWaiting` state.
5. When an `Active` object has been alerted of a particular condition, but has not yet synchronized itself, it is in the `ActiveAlerted` state.

These states, and the relationships among them, are shown in.

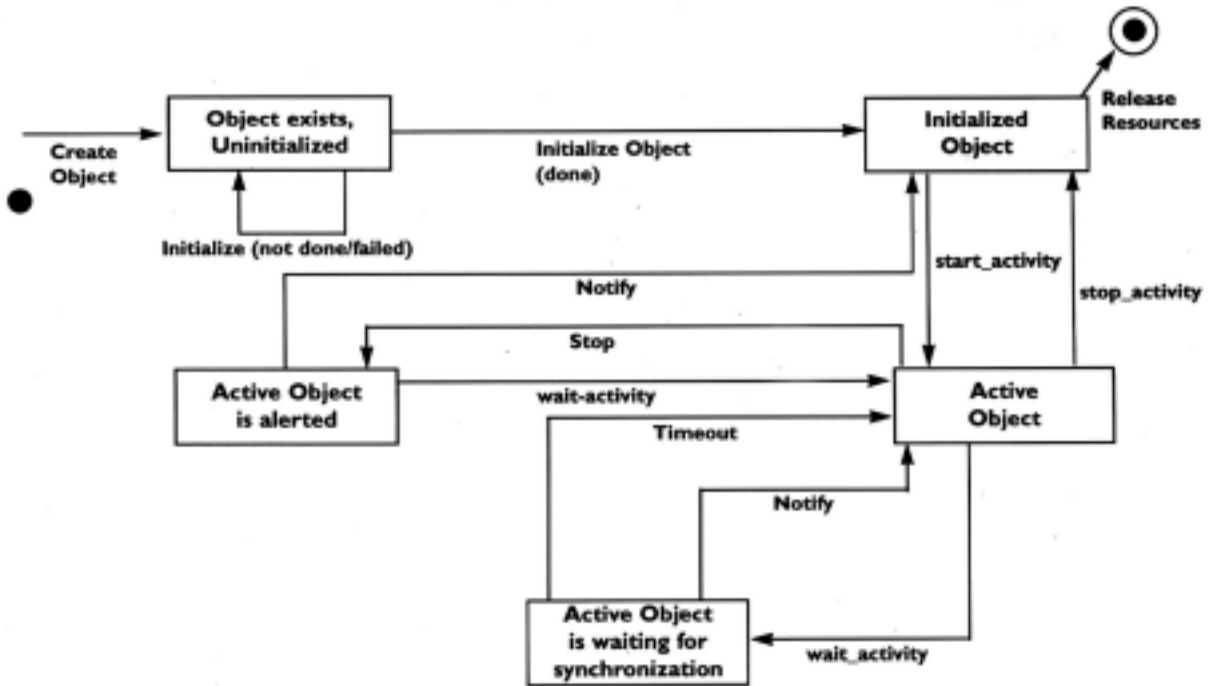


FIGURE 6 - 3. Mobility Object States

Object Container Interface

An Object Container is a component that also supports the interface needed for containing other Mobility components. This additional interface is called the Object-Container interface. The ObjectContainer Interface allows objects to be inserted or removed from a container at runtime, and supports dynamic location of objects within the system.

Mobility objects are organized hierarchically. For example, from the top down, robot configurations contain services. Services, in turn, contain object interfaces. Objects and other system elements may contain subordinate elements. Any of these Mobility ContainedObjects can be located via a pathname consisting of a sequence of name element strings.

ObjectContainer objects are frequently notified of changes or updates to their child objects and are used to update a group of child objects, based on the update of a single child object. The Object Container Interface is also supported by containers for other Mobility system elements.

Property Container Interface

The Property Container Interface allows other system elements (including user interface programs) to examine and modify the properties of an object at runtime. This interface is supported by an object (like a service) that handles properties used for its configuration and management.

Many types of objects and modules support dynamic and static properties that are used to manage configuration. The property descriptor structure identifies and names properties within Mobility. Some examples of properties are

- the I/O port used by a hardware server
- the conversion factor between device units and physical units
- the “gain parameters” of a control algorithm

ActiveObject Interface

The ActiveObject Interface allows clients to create and manage ActiveObjects within Mobility.

An ActiveObject encapsulates an activity that is an independent thread of control within the system. ActiveObjects respond based on timeouts or other triggers, and represent a prioritized path of execution. An active object periodically synchronizes with other active objects and can wait for other active objects. Once the active object is stopped, it can be restarted. Many of the components that handle I/O ports in Mobility, for example, are ActiveObjects.

Object Factory Interface

Object factories permit the dynamic creation of Mobility objects at runtime. Factories allow clients to dynamically create new objects within running servers. Mobility’s ability to reload a configuration of objects from persistent storage derives partly from the services provided by factory objects. The SystemModuleComponent is one Mobility object that supports the Object Factory Interface.

Mobility Externalization Interfaces

Externalization Interfaces support persistent configurations of Mobility components. A group of software modules, along with their properties and interconnections can be stored in a configuration on disk and restored later.

NOTE: The externalization functionality is not implemented for the Mobility 1.1 release.

Two primary abstractions are used in the externalization Interface:

1. The I/O Stream, an abstraction of a serial medium that supports read and write operations like files
2. The externalizable object, which can write its persistent state to an IOStream when requested

Each externalizable object has a “key” that is stored with it. Through this key the Mobility system can locate the factory to dynamically create the object when the object is to be restored from persistent storage. The structure of the externalization system allows for the centralized management of configuration, while still permitting fully distributed processing and decision making within a multi-robot system Externalization Interface

MobilityComponents Module

This module defines combinations of MobilityCore and MobilityExternalization interfaces that are implemented by the components of a Mobility software system. Software that you write for Mobility will generally be some type of MobilityComponent and will be derived from framework base classes that provide much of the implementation for MobilityComponent interfaces for you.

StateChangeHandler Interface

This is the core notification mechanism between dynamic state and computation in Mobility. This interface is implemented by Mobility objects that want to register for state change notifications with a portion of dynamic state within the system. More complex notification schemes are based on this simple notification, further examination of state, and other factors. StateChangeHandlers are usually registered by containers that enforce constraints between contained objects.

SystemComponent

The SystemComponent Interface defines an object that is a property container and is streamable to persistent storage. System Component objects represent robot hardware and software “modules” that respond to configuration through properties and can be registered as StateChangeHandlers. These components form the backbone of most modules. The system component classes also have several control states that allow system components to process updates and requests differently.

SystemComponentStatus

The SystemComponentsStatus enumeration defines the possible processing states for SystemComponent classes:

1. Disabled state: a component may simply do nothing in response to certain updates
2. Reactive state: a component will respond fully to updates
3. Active state: a component will take actions based on internal threads of control

CompositeSystemComponent

The CompositeSystemComponent can contain other Mobility objects in a hierarchy of object instances. CompositeSystemComponents can be used to aggregate related state and computational objects and treat them as a group. A special CompositeSystemComponent, called the SystemModuleComponent becomes the “root” of a collection of objects within a single process.

ActiveSystemComponent

The ActiveSystemComponent has its own thread of control. Objects that sample hardware, communicate between processes across networks, or coordinate long term, open ended processing are frequently ActiveSystemComponents.

SystemModuleComponent

A Mobility SystemModuleComponent is a CompositeSystemComponent. That is, it contains all the other components in the module. It provides the generic “factory interface” that allows clients or managers to create objects in the module.

The SystemModuleComponent also supports the creation of new objects dynamically at runtime. It is essentially a use of the “factory design pattern” to allow mod-

ules to support dynamic creation of services under the control of a configuration client.

StateObserver

The StateObserver object is an explicit subscription object created by clients of state objects. The StateObserver gets configured for various state change events and uses its internal thread to “push” or send notifications back to a client.

Combining the notification mechanism of data objects with a StateObserver object allows you to build very flexible update mechanisms. A StateObserver acts as a StateChangeHandler and attaches a number of client interfaces to a number of state objects.

When the State Observer is triggered, it updates the client objects with the appropriate state information. Typically, a client object will configure the rest of the triggering mechanism through instance properties for maximum flexibility. One way to think of the StateObserver is as a programmable “caller-backer” object.

MobilityData Module

This module defines combinations of the MobilityCore and MobilityComponents interfaces as well as some new interfaces. The components defined in the Mobility-Data module serve as state storage components for the Mobility system. There are both generic StateComponents and typed StateComponents. The current framework provides implementations for these components that are ready to use in your own software.

DynamicObject Interface

This interface captures objects that generate change events. The primary state mechanism is a subclass of this generic object. DynamicObjects can be modified externally and are designed to tell any registered parties when they are modified, through a notification interface. Objects that support these interfaces work with objects that support the StateChangeHandler interface to support event-based notification and data push.

Mobility StateComponents

Mobility StateComponents are representations of state that are updated dynamically and support multiple observers of this update. Some Mobility StateCompo-

nents are also container objects and contain many sub-representations of state (that is, different views or levels of detail.) Mobility StateComponents form a “perceptual buffer space.” Change is communicated by propagating notification calls from states to registered clients.

Think of StateComponents as type-safe mini-blackboards.

Mobility’s standardized container interfaces allow data objects and representations to be extended over time. Thus, new elements can be added to meet the need of the robotics research and development community.

The StateComponents that represent the lowest level of the robot hardware are updated dynamically and asynchronously. Multiple active objects in the base server interact with robot I/O ports and update the lowest level StateComponents. The change notification from these updates initiate the processing and transforms that provide higher level-abstractions.

All StateComponents have similar interfaces. StateComponents of various types exist for various purposes in the Mobility system. The set of StateComponents will grow over time as needed. The basic StateComponents that are supported at this time are:

- BvectorState — Byte vectors are useful for sparse raw sensor data.
- FvectorState — Floating point vector state is good for converted sensor data in meters.
- IvectorState — Integer vector state is good for communication of dense raw sensor data.
- TransformState — Represents changing geometric transformations such as odometry.
- PointState — Set of 3D points in space (like sonar target points).
- SegmentState — Set of 3D line segments in space (like sonar beam centers).
- ActuatorState — Represents the status of an actuator or an actuator command value.
- StringState — Set of strings is a useful state for blackboards and debugging information.
- ImageState — Integer image data from most cameras.
- FimageState — Floating point image data from things like probability images.

- `GenericState` — Used as a catch-all for passing various state samples around, less efficient than typed interfaces.

This type of object represents raw actuator data for a set of actuators, presenting generalized notions of status and command for a set of actuators. For example, a 6-axis manipulator might take a 6-element force command, or a mobile robot base might take a translate force and a rotate “torque,” that is, a generalized force.

Managing System Configuration

Mobility’s `ObjectContainers` let you easily manage the configuration of a multi-robot system. The `ObjectContainer` model allows a user, or other system objects, to “browse” through the system and examine its structure and components dynamically at runtime.

Three other sets of interfaces deal primarily with configuration of the system:

- `Properties`
- `Factories`
- `Externalization`

These interfaces work together to provide the ability to configure a multi-robot system from a single program, as well as to save that configuration to permanent storage and load that same configuration from storage back into a running system.

Property Container Interface

All Mobility components are property containers. Properties allow users or even other objects in the system to modify or “tune” the operation of an object. Properties store hardware configuration data like port names and robot parameters as well as behavioral parameters like a robot’s safety distance for avoiding obstacles. Objects respond to some of their properties only when they are initialized. Other properties are dynamic. Dynamic properties, for example, the gain of a feedback controller or the keep-off distance of an avoidance routine, affect the operation of the object immediately and can be changed at runtime.

Putting the Components Together

Let’s look at an example of all these classes and interfaces working together to build a robot hardware abstraction from a set of reusable, configurable Mobility objects. Figure 6 - 2, “Mobility Class Diagram: Core and Components,” on

page 6 - 5 shows a SystemModule that contains several different MobilityComponents, each of which represents a different part of the robot system and has a unique pathname within the hierarchy of instances. shows two different types of object interactions.

- Solid lines and arrows show the system’s containment hierarchy, the organizational plan that forms the backbone of the Mobility system. Through this hierarchy, objects locate one another and discover available services. This hierarchy also allows for managing properties and saving objects to persistent storage.
- Dashed lines and arrows show the dynamic flow of sensor and actuator data through the system. These interactions include updates of DynamicObjects, notifications between DynamicObjects and their container objects through the StateChangeHandler interfaces and other calls that update and modify the state objects.

To see how data might flow through a robot system, let’s look at sonar data. The ActiveSystemComponent called “Hardware I/O” is a subclass of ActiveSystemComponent that knows how to talk to a particular set of robot hardware, in this case, the sonar sensors. The “Hardware I/O” component is configured to connect to several StateObjects that represent relatively raw sonar data going to and from the robot hardware. When the “Hardware I/O” component updates the raw sonar data, contained within a class of StateObject called FvectorState, the owner of the “raw” data is notified through the StateChangeHandler interface.

The “Sonar” object, a subclass of the CompositeStateObject, processes this notification and uses its knowledge of sonar geometry to update its abstract representations of the robot state called “targets” and “rays,” geometric representations of the raw sonar data. These abstract representations are more useful than raw information because the robot-specific geometry is handled within the “Sonar” component and users of the data don’t depend as much on specific robot hardware. When the “targets” are updated, they trigger conditions to push this update into a remote object somewhere else in the system.

This combination of hierarchical structure and notification-based data flows are the keys to a extensible, flexible software development environment. As your research requirements evolve and expand, your Mobility robot system will remain fully integrated even as it grows more capable.

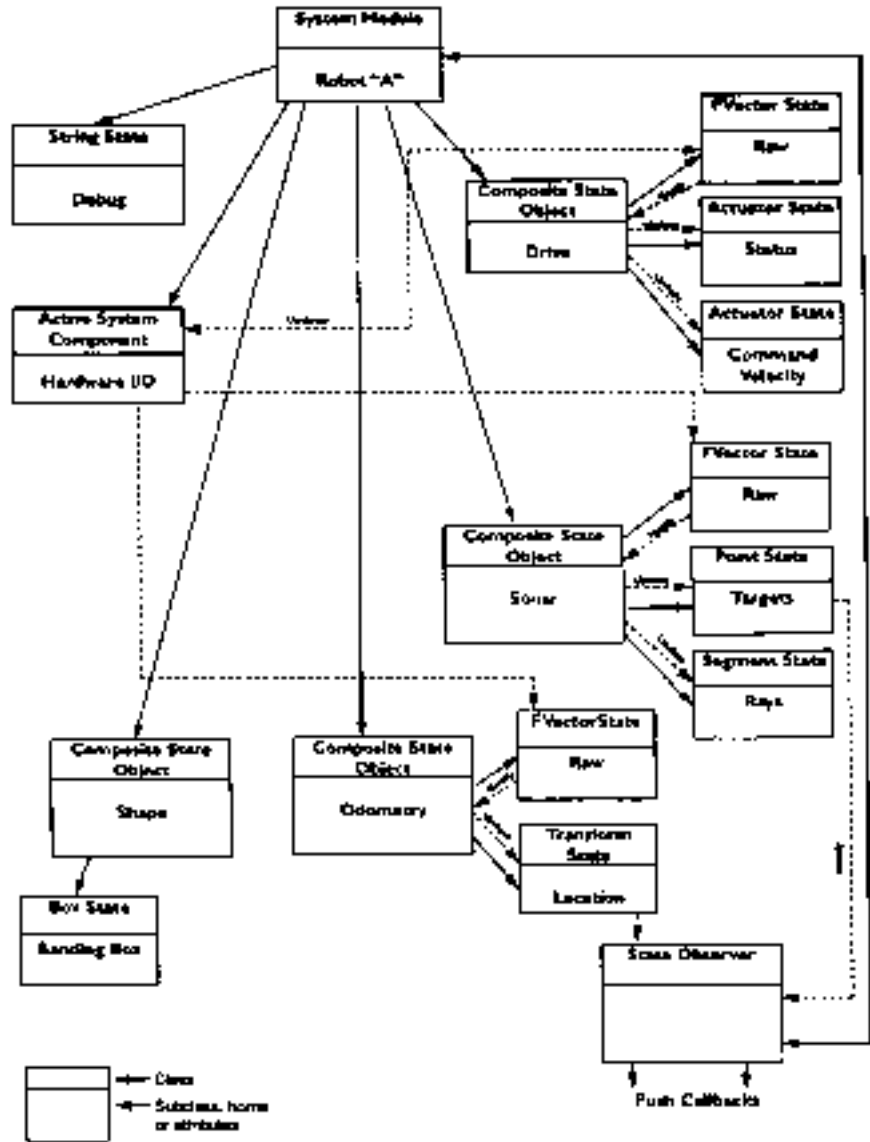


FIGURE 6 - 4. Anatomy of a Mobility Base Server

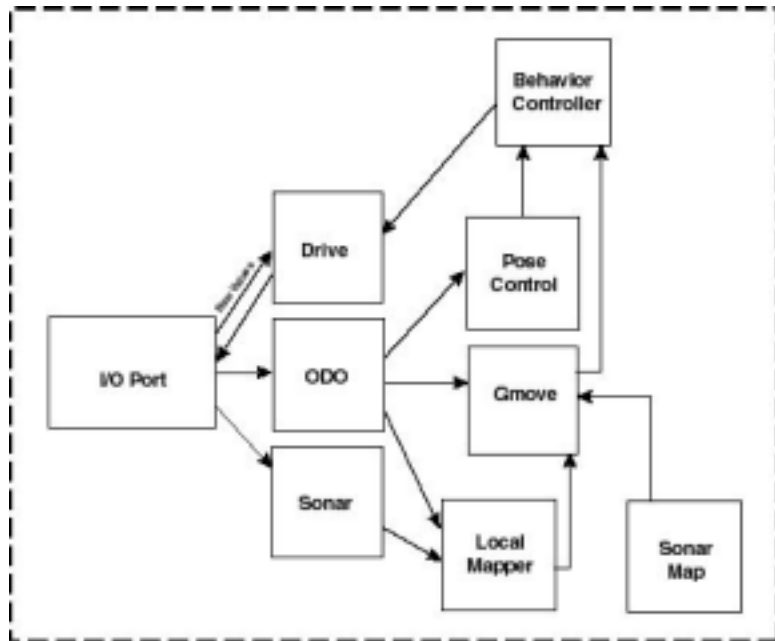


FIGURE 6 - 5. Data Flow in a Mobility Base Server

Mobility Building Blocks: Basic Robot Components and Interfaces

The Robot as a Hierarchy

Mobility's basic robot components provide low-level abstractions of robot sensors, actuators and physical properties in the form of Mobility System Components. Previous robot software packages utilized a kind of "base server" program that managed some of the functionality provided by Mobility's basic robot components. But Mobility's basic robot components provide a much richer and extensible abstraction of the physical robot.

In Mobility, a robot is a hierarchically-arranged collection of elements. The top level of any robot is the `SystemModuleComponent`, a `Mobility SystemComponent` that contains the separate subsystems of an individual robot. Each of these subsystems is itself a `CompositeSystemComponent`. The subsystems contain dynamic `StateObjects`, updated based on the state of robot sensors and actuators, and properties that allow client programs to discover and adapt to the properties and resources of an individual robot at runtime.

Through the interfaces of these objects, client programs can obtain

- state information about the robot
- location of robot sensors
- properties of the robot sensors

- general shape of the robot body
- other basic geometric information determined by the robot hardware itself

Clients also command the robot to move by updating the state of objects contained in the drive object.

While differences among robot hardware characteristics are never completely hidden, Mobility objects let your programs easily and flexibly identify and use a wide variety of robot hardware. State Objects provide baseline abstraction for various robot hardware platforms and ways for clients to discover and adapt to differences among robots.

Basic robot objects, especially sensors, provide different views of their state information: raw sensor numbers and geometric information like robot-relative target points or contact-points.

Robot Abstractions, Objects and Interfaces

The Mobility Robot Object Model defines some common abstractions that are useful for building robot control software. These abstractions include representations for

1. Sensor state
2. Actuator state
3. Physical robot shape

In addition to these fundamental abstractions, there are some basic interfaces for building robot behaviors or control layers for your robot control system.

The concept of state is captured by StateComponents. These objects provide simple state buffers that are accessible from anywhere in the Mobility system. This allows StateComponents to communicate state changes, and provide access points where the internal operation of the software can be “viewed” remotely for debugging of algorithms.

Sensor Systems

The sensor systems of a robot are each represented as a CompositeStateComponent whose properties describe the relevant features of the sensor system. For example, the divergence angle of sonar sensors is stored as a property in the Sonar CompositeStateComponent. There is usually a set of StateComponents included as child

objects underneath a sensor system component that represent various “views” of the sensor state. For sonar sensors these views include raw range data (in a `FvectorState` component) as well as points (`PointState` component) and line segments (`SegmentState` component). The combination of various state views and sensor system properties allows displays and behaviors to adapt to the geometric variations among robot systems more easily.

Sonar Sensing

The sonar sensor object is a `CompositeSystemComponent` that contains several `StateObjects` that represent the state of the robot sonar sensors. The sonar sensor object contains an `FVectorState` object that represents the raw sonar sensor reading for the robot. This information is provided primarily for debugging and testing use, because the sonar sensors also provide sonar readings in the form of a `PointState` object. The `PointState` object provides target points for each sonar sensor on the robot in robot relative coordinates.

A `SegmentState` object provides line segments for each sonar sensor reading that include the origin and target point for each sonar reading. The range view tool program uses this view of the sonar system state to provide a visualization of the robot’s sonar sensors.

How the Sonar Sensors Work

SONAR, an acronym for `SOund Navigation And Ranging`, models the contours of an environment based on how it catches and throws back sound waves. The sender generates a sonic, or sound, wave that travels outwards in an expanding cone, and listens for an echo. The characteristics of that echo can help the listener locate objects. The sonar sensors on an iRobot robot provide a useful map of the robot’s surroundings, as long as their inherent limitations are realized

iRobot’s B21 robot, for example, reads its sonar sensors about three times per second. (Other robots update at different rates; check the *User’s Guide* for your own robot.) For each reading, the total time between the generation of the ping and the receipt of the echo, coupled with the speed of sound in the robot’s environment, generates an estimate of the distance to the object that bounced back the echo.

As the robot’s sonar sensors fire off pings and receive echoes, they continuously update a data structure. Each sonar sensor detects obstacles in a cone-shaped range that starts out, close in to the robot, with a half-angle of about 15 degrees, and spreads outwards. An obstacle’s surface characteristics (smooth or textured, for

example), as well as the angle at which an obstacle is placed relative to the robot, significantly affect how and even whether that obstacle will be detected. Rather than assuming that sonar sensor data is infallible, look at multiple readings and do appropriate cross checking. The sonar sensors can be fooled for any of these reasons:

- The sonar sensor has no way of knowing exactly where, in its fifteen-degree and wider cone of attention, an obstacle actually is.
- The sonar sensor has no way of knowing the relative angle of an obstacle. Obstacles at steep angles might bounce their echoes off in a completely different direction, leaving the sonar sensor ignorant of their existence, as it never receives an echo
- The sonar sensor can be fooled if its ping bounces off an obliquely-angled object onto another object in the environment, which then, in turn, returns an echo to the sonar sensor. This effect, called specular reflection, can cause errors; the sonar sensors overestimate the distance between the robot and the nearest obstacle.
- Extremely smooth walls presented at steep angles, and glass walls, can seriously mislead the sonar sensors.

But each robot has multiple sonar sensors, providing redundancy and enabling cross checking. Sonar sensors almost never underestimate the distance to an obstacle. Therefore, it's a good idea to examine the distances returned by a group of sonar sensors and use only the lowest values.

Or, record multiple readings as the robot moves about, and use the data from each to build up an occupancy grid. If several readings, from several angles and several sonar sensors, keep detecting an obstacle in more or less the same place, it's probably safe to mark that spot as occupied.

How The Sonar Sensors Can be Fooled

When the sonar beam strikes a surface with a large angle of incidence, the edge of the wave front is reflected back to the sensor instead of the centerline. This effect, called radial error, often results in errors greater than one foot.

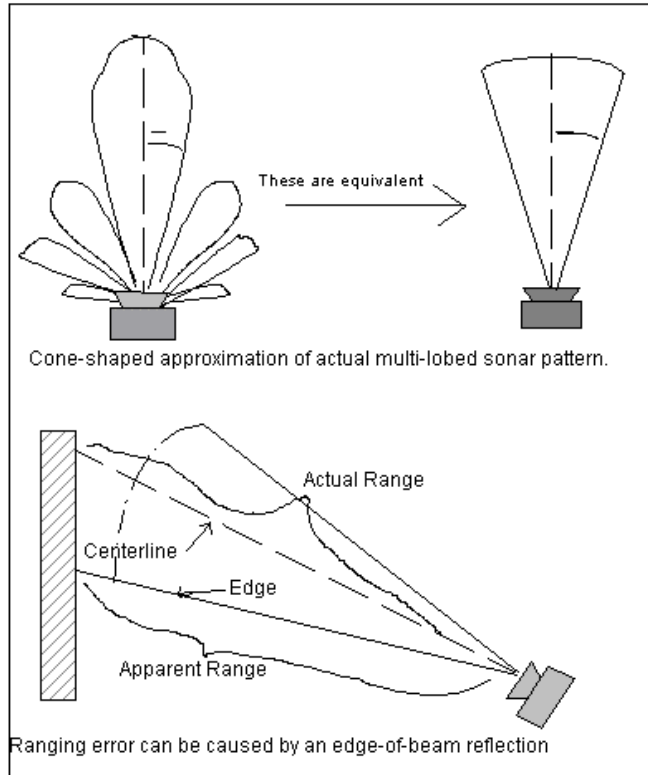


FIGURE 7 - 1. How the Sonar Sensor Can Be Fooled: Ranging Errors

In addition, because of the sonar sensor's relatively large beam (its angle is about 15 degrees), it tends to produce a rather blurred image of its surroundings. This can

lead to angular error, which affects the robot's impressions of its surroundings in ways similar to radial error.

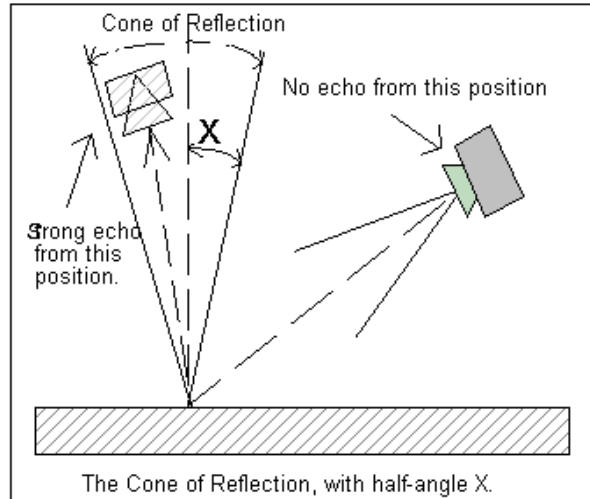


FIGURE 7 - 2. How the Sonar Sensor Can Be Fooled: Angular Errors

After striking a surface at a large angle of incidence, the echo may bounce into oblivion rather than reflecting a strong echo back to the sonar receiver. This type of false reflection occurs when the incidence angle of the beam is greater than a critical angle, denoted in Figure 7 - 2, “How the Sonar Sensor Can Be Fooled: Angular Errors,” on page 7 - 6 ,as X , which defines the cone of reflection (CR) for the surface.

A sonar beam striking a wall from outside the CR will be reflected away from the sensor, producing an unrealistically long sonar ray. The sonar beam apparently penetrates the wall.

Every surface material has its own CR half-angle, ranging from 7 or 8 degrees (for glass) to nearly 90 degrees (for rough surfaces.)

Infrared Sensing

Your robot may be equipped with infrared sensors. These sensors provide an indication of the proximity of an obstacle by emitting light and measuring the intensity of the reflection bounced back off the obstacle's surface. Infrared light, the part of the electromagnetic spectrum of radiation above 0.75 millimeters in wavelength, is

not visible to humans. (We perceive it only as heat.) Infrared sensors provide the robot information it can use to help build up a picture of its surroundings.

Robotic Tactile Sensing

A robot's tactile sensors are represented by a CompositeSystemComponent that contains several StateObjects that represent the states of each of the robot's tactile sensors. The tactile sensor object contains a BVectorState object that represents the raw tactile sensor readings. This information is provided primarily for debugging and testing use. The tactile Sensor object also provides a PointState view of the contact points of the robot.

Odometry and Position Control: the RobotDrive Object

The robot drive object is a Composite SystemComponent. It contains several StateObjects that represent the state of the robot drive system and the current drive command. An ActuatorState object represents the state of the robot current drive system. Another ActuatorState object represents the state of the current drive command.

The robot odometry object is a CompositeSystemComponent. It contains several StateObjects that represent the state of robot odometry. A TransformState object represents the current location of the robot. An FVectorState object represents the robot's current velocity.

The robot's mobile base is equipped with wheel encoders that keep track of the revolutions of the wheels as the robot travels about its environment. The robot's motion controller integrates these measurements to attempt to estimate the robot's current position at any time with respect to its original position; that is, where it was when it started rolling.

While this measurement is highly accurate for short distances, error can and does accumulate as the robot travels further afield. By itself, the robot's motion controller hardware has no way to detect wheel skid or errors in wheel tracking, routine hazards in real-world research and operational environments plagued with slippery floors, carpeting, doorjams and the like. Carpet fibers present special challenges to robots operating on carpeted surfaces.

Mobility provides odometry information computed from robot wheel rotation. You can get the robot position in X,Y, and theta relative to the startup location. You can

also get the velocity of the robot in X,Y and theta. The odometry output is in meters and radians and the velocity output is meters/second and radians/second.

With Mobility, you need not concern yourself with processing raw encoder data. Mobility updates the odometry quite rapidly (> 10 Hz). Therefore, Mobility does not provide direct encoder feedback at this time

How Mobility Processes Encoder Data

Your robot's rFlex controller takes in the 4 raw encoders and does motion control based on the encoder feedback. The rFlex controller combines the encoders into two virtual axes:

1. Translate axis
2. Rotate axis

These “virtual encoders” are sent to Mobility. Mobility sends rotate and translate velocity commands to these virtual axes. The Mobility base server also converts these integers into actual units (meters and radians). The velocity of the axes is also reported and converted to meters/second and radians/second. A Mobility- based program can get updates from the virtual axes at about 10-15Hz.

The combination of encoders to virtual axes accommodates and corrects for the slippage of the wheels in turning and when covering rough terrain to provide the best estimate of the translation and rotation movements (position and velocity).

CAUTION: iRobot does not support modification or access to the encoders, because such access and/or modification can compromise the safety or our robot control system.

NOTE: In future releases, Mobility will fuse the data from odometry and a compass, improving both speed and accuracy of odometry readings.

Actuator System Abstractions

The actuator systems of a robot are each represented as a CompositeStateComponent whose properties describe the relevant features of the actuator system. For example, the maximum acceleration of a robot base is captured as a property. The actuator system will also contain several child components that represent various views of the actuator system, including a child that represents the status of the actu-

ator system and a child that is the current command input to the actuator system.

Robot Shape Abstractions

The robot shape abstractions provide geometric descriptions of the physical shape of the robot at various levels of detail. Shape abstractions include things like an overall “bounding box” for the robot, as well as more detailed descriptions of robot shape.

NOTE: The robot shape abstractions are not supported by Mobility 1.1 servers.

Behavioral Abstractions

Active objects, along with the ability to communicate dynamic state updates between components, allow Mobility to support the development of closed-loop robot control behaviors. These components combine input from one or more sensor systems and generate commands for one or more actuator systems within a robot.

Parallel Behaviors

Mobility 1.1 does not include the necessary command arbitration facilities for building parallel behaviors.

Layers of Control

The simplest form of behavior combination is to provide a “layer” between high-level control and lower level control. This “layer” combines some sensor inputs with commands from a higher level component to produce the commands for one or more actuators. For example, the guarded motion combines drive commands and sonar sensors from base server to provide reactive obstacle avoidance.

Mobility Building Blocks for Extensibility

As iRobot continues to develop new features for Mobility robot integration software, we will package these features as reusable building-blocks for your robotics application development needs. Some of the packages will be included with the environment and others will be offered as additional packages, based on Mobility.

The following sections provide brief descriptions of components that will be available for Mobility in future releases.

Keeping Track of Obstacles: Local Map

The Local Map components provide multiple views of a “robot-centric” space. This space is like short-term memory for the robot system. It provides a place where the results of processing various sensors may be stored and used by behaviors or other control algorithms to generate motion commands. The Local Map components can be part of the reactive layer for a robot control system. Everything in the Local Map decays with time so that spurious sensor results are eventually erased after several seconds.

The GUI Tools

Along with the Local Map components there are additional viewers provided with MOM that will allow on-line display of the state of the Local Map for debugging the operation of the robot.

The Programming Interface

Local Map components provide additional interfaces and properties that you can use to write your own perceptual routines or behaviors, based on the shared Local Map data base.

Playing Nice: Guarded Motion

Guarded motion is a kind of collision avoidance behavior that combines an input command with feedback from the robot sensors. The resulting motion command is one that uses the robot’s sensors to avoid collision with obstacles in the environment.

The Programming Interface

Guarded Motion adds new components, but uses the same interfaces as an actuator system. To programs that use Guarded Motion, it is a “virtual actuator” that provides built in collision avoidance.

Getting to the Point: Pose Control

Many robot systems are designed to take “point commands.” Mobility, on the other hand, is based on a more reactive, “velocity command” approach. However, Mobility’s Pose Control components provide a “position command” interface to the robot. Mobility’s Pose Control components use the odometry feedback from the robot, along with input commands, to generate a command that will push the robot into a specific position and orientation, or “pose”.

Sim: The Mobility Simulator

The Mobility Simulator

Mobility robot integration software supports full multi-robot simulation. The Mobility Simulator, called Sim, provides a simple hardware simulation for debugging your algorithms before you try them out on a robot. Sim is a kinematic/geometric simulator and does not simulate the mass of objects, inertia, and similar physical characteristics. Sim uses the models of sensors drawn from experience and empirical measurement of sensor performance on real robot platforms.

Sim is a collection of Mobility components that, in concert, serve as a replacement for the hardware components that interact with the lower level of a robot. Sim's core is a centralized database for keeping track of multiple robot clients and the simulated world state.

Hardware simulator modules connect to this world simulator core, and provide to the rest of your Mobility system the simulation of a particular robot type. From the point of view of software interfaces, these hardware simulator modules are identical to the base server components of real robot hardware. Thus, your Mobility robot control components run identically whether you are running Sim or driving a real, physical robot. Sim lets you test your code on different robot hardware platforms, thereby discovering any untoward assumptions about robot geometry that may be lurking in your code.

Sim's World Simulator Core

Sim's world simulator core is the central database that describes the simulated world in which you are testing your robot system. The simulator core is a Mobility-SystemComponent (just like base servers and your robot control programs) that contains a dynamic database of objects and robot locations. The world simulator core is fully multithreaded; with more processing power available, it will scale up to simulation of larger numbers of interacting robots.

The GUI Tools

The Programming Interfaces

Sim's Robot Hardware Simulator Modules

Sim's hardware simulator modules connect to the world simulator core and provide the same software abstractions as modules that connect to real, physical robot hardware. Of course, Sim's hardware simulators do not exactly mimic in quality and fidelity real robot hardware in a real environment, but Sim is nonetheless a useful tool for debugging the first few passes of a new control algorithm. After using Sim to sanity-check your robot programs, you won't have to waste time on such trivialities as accidental code bugs when you finally get to fire up a real robot.

The Robot Simulator Visualization Interface

Web-Based Visualization Interface

RML 2.0 Interface

Running Other Objects with Sim

Advanced Issues And Common Questions

How Do I...?

Work with a multi-robot team?

Mobility lets you seamlessly integrate a multi-robot team and access all the elements of a team from the Mobility System Manager.

Write modules that handle multiple robots?

All objects in a Mobility system are accessible through the hierarchy of objects. You can connect your module to any robot by using pathnames for different robots.

Deal with multiple threads in my modules?

Mobility uses multithreaded modules to provide programming flexibility. However some libraries are not compatible with the threading libraries under Linux.

Make my own interfaces and extend the robot object model?

Mobility lets you seamlessly integrate a multi-robot team and access all the elements of a team from the Mobility System Manager.

Use my old BeeSoft programs with Mobility?

Mobility has a much larger scope than BeeSoft. Mobility does not include support for running BeeSoft programs unmodified. In most cases, though, it should be possible to port your BeeSoft algorithms to Mobility. You might want to get together with other BeeSoft users to plan your BeeSoft-to-Mobility migration strategy.

Use my old Saphira programs with Mobility?

Mobility is also much larger in scope than Saphira. Mobility does not include support for running Saphira programs unmodified.

Program Mobility from my LISP system?

Because Mobility uses an open standard to define the interfaces to all elements of the system, you can use all the tools, GUIs and basic robot objects of Mobility with programs written in LISP. You'll need a CORBA 2.X-compliant implementation for your version of LISP, and you'll need to compile the Mobility interface definition files (IDL) for LISP. You won't have the benefit of the class framework, but you will be able to directly access all Mobility components.

Change a Mobility-defined interface?

iRobot respects and appreciates the diverse talents, projects and ideas of Mobility users. If you come up with a change or enhancement that you think would benefit Mobility's core interfaces, please let us know. (support@rwii.com or docs@rwii.com). iRobot engineers might have a suggestion to accomplish what you need to do without changing interfaces. Or, we may incorporate your suggestion into a future release of Mobility robot integration software.

Why Did You...?

Use CORBA 2.x as an interface standard?

The second version of the CORBA standard included a number of significant improvements. Plus, several commercial and non-commercial implementations were available. iRobot has noticed a tendency for developers to build robot control software based upon custom, proprietary or non-standard communications protocols for managing the many different elements of a multi-robot system. This prac-

tice doesn't encourage the kind of cooperation and code re-use iRobot would like to see and support in the robotics research community

Change from BeeSoft?

BeeSoft's simple API was just not adequate to support multiple robot systems, nor were there any facilities for building an infrastructure for a system that would be extensible over time. We decided that our customers would be better served by a more complete, powerful and more fully integrated robot software package.

Support only C++ and Java?

These languages offer the most commonly available object-oriented tools used by a large community of commercial and non-commercial developers. High-quality Java and C++ development tools are available for a wide range of operating systems and the tools.

Installing Mobility

These instructions assume that you have some experience with Linux system administration. If you are new to Linux system administration you'll need a bit of help from someone more experienced.

(At the time of this writing, the iRobot engineers are working to turn the Mobility software into a set of RPM's. RPM, the Redhat Package Manager, is a tremendously useful, easy-to-use, and powerful tool for installing, uninstalling, and upgrading software packages on Linux. When that is complete, management of Mobility software will be significantly simplified.)

There are many scenarios in which you might need to install Mobility software. The following scenarios are covered in this appendix.

You need to download and install a new release of Mobility on your robot.

- You want to install Mobility on another PC in your laboratory for software development.
- You have a new Magellan or similar robot with no on-board PC, and you need to download, install, and configure Mobility to control your robot over a radio RS-232 link.
- You want to install just MOM on a desktop computer in order to get good user interface performance.

- You want to do software development off the robot, and copy only the necessary files onto the robot to run.
- You want to install Mobility on a new computer in your robot, so that it's just the same as we install it at the factory.

NOTE: Your robot comes from iRobot with Linux and Mobility pre-installed. You do not need to do these steps with a robot "out of the box".

The scenarios above are highly overlapping. Note, however, that The sections that follow are organized to minimize repetition. They are presented in the order in which you're most likely to need to do them:

- Install Linux on the robot's on-board PC and prepare it for Mobility.
- Set up a mobility account.
- Download Mobility software.
- Install Mobility.
- Configure an off-board PC for radio RS-232 link to a robot such as the Magellan.
- Install MOM only on a desktop PC.
- Install base server only on a robot PC.

Install Linux on the Robot's On-board PC and Prepare it for Mobility.

Follow these instructions to (re)install the operating system on the robot's on-board PC and make other preparations for the Mobility software. The PC will be set up for self-hosted development. That is, the same computer may be used for the software development as well as actually running the software.

Consult your Red Hat documentation to install Red Hat Linux 5.2 (or 5.1). iRobot recommends that you connect the COM1 serial port of the robot's PC to a serial port on another computer, and run a terminal emulator program such as Minicom to access the PC console. Connect the robot's PC to your local ethernet and use the CDROM drive of another computer "over the net" to avoid having to connect a CDROM to your robot PC.

Do a reasonably full installation of Linux. Be sure to include networking, C++ software development tools, httpd, and system administration tools. If you want a graphical front-end to the multithreaded debugger, download the following RPM's from the iRobot ftp server (see below) or elsewhere on the net.

```
ddd-doc-3.0-5.i386.rpm  
ddd-static-3.0-5.i386.rpm
```

As root, install them with the commands

```
rpm -U ddd-doc-3.0-5.i386.rpm  
rpm -U ddd-static-3.0-5.i386.rpm
```

For Red Hat Linux 5.1 only:

You need to install the required version of egcs (the C++ compiler) and some software for multithreaded debugging.

Obtain two tar files containing egcs from the iRobot ftp server (see below) or elsewhere on the net, and five files containing RPM's for multithreaded debugging.

```
egcs-core-1.0.3a.tar.gz  
egcs-g++-1.0.3a.tar.gz  
glibc-2.0.7-20.i386.rpm  
glibc-devel-2.0.7-20.i386.rpm  
glibc-debug-2.0.7-20.i386.rpm  
glibc-profile-2.0.7-20.i386.rpm  
gdb-4.17-5.i386.rpm
```

As root, build egcs and install it in the default location (/usr/local).

```
tar xzf egcs-core-1.0.3a.tar.gz  
tar xzf egcs-g++-1.0.3a.tar.gz  
cd egcs-1.0.3a/  
./configure  
make  
make install
```

As root, install the multithreading libraries.

```
rpm -U glibc-2.0.7-20.i386.rpm  
rpm -U glibc-devel-2.0.7-20.i386.rpm  
rpm -U glibc-debug-2.0.7-20.i386.rpm  
rpm -U glibc-profile-2.0.7-20.i386.rpm  
rpm -U gdb-4.17-5.i386.rpm
```

You are now finished preparing a robot computer for Mobility.

Set Up a mobility Account.

At the factory, all iRobot robot on-board computers are set up with a user account for the Mobility software. The account's user name is `mobility`. This account is configured to have all the environment variables and details set up automatically upon login to minimize the effort of getting started with Mobility. Each software developer will normally have his or her own individual account, sharing the Mobility libraries and programs that are in the `mobility` account.

When setting up an off-board computer for Mobility software development, or to control a robot that has no on-board computer, it is usually most convenient to set up a mobility account in nearly the same way. By convention, the Naming Service places its IOR in a file in `~mobility/public.html`.

Before setting up a mobility account your computer, your computer must have the Linux operating system installed.

As root, create the `mobility` user account. Give the account sudo privileges (optional) and a password.

```
useradd mobility
visudo
```

At the end of the file, add the line:

```
mobility ALL=ALL
passwd mobility
```

Log in as `mobility` and edit the file `~/.bashrc`. Determine which robot base server program is appropriate for your robot using the table in Chapter 5. In the text below, replace `***robotserver***` with the correct server program name. Be sure to use the actual version number of your software release.

At the end of `~mobility/.bashrc`:

```
export MOBILITY_ROOT=~/.mobility-b-1.1.0
export MOBILITY_NS=http://`hostname`-s`/~mobility/NamingService
alias base=***robotserver***
if [ -f $MOBILITY_ROOT/etc/setup ]; then
```

```
source $MOBILITY_ROOT/etc/setup
fi
```

Create the directory `~mobility/public_html` with permissions `2777` to hold the file containing the IOR of the Naming Service and to hold a link to the Mobility on-line documentation.

```
cd
chmod a+rx .
mkdir public_html
chmod 2777 public_html
```

Create a symbolic link from `~mobility/public_html/docs` to the online documentation.

```
cd ~/public_html
ln -s ~mobility/mobility-b-1.1.0/docs .
```

Enable the web server to serve the docs:

```
cd /etc/httpd/conf
sudo vi access.conf
```

After the line "`<Directory />`" change the line:

```
Options None
```

to be

```
Options Indexes Includes FollowSymLinks
```

Now restart the web server:

```
sudo /etc/rc.d/init.d/httpd stop
sudo /etc/rc.d/init.d/httpd start
```

Download Mobility software.

Obtain the latest release of the Mobility binary software from the Real World Interface FTP server. There may be several versions of the software and Mobility User's Guide available. iRobot recommends that you obtain the latest one (that is, the one with the highest version numbers). Other files are also available such as the enhancements necessary for Red Hat Linux 5.1 to support Mobility and a graphical front end to the multithreaded debugger.

You may download these files from iRobot whenever you need them during your installation process. There is no need to be root when you download them.

Use FTP and anonymous login to connect to ns.rwii.com.. Go to the Mobility 1.1 software package directory. The directory itself is invisible, and the case of the directory name is critical. Once you are in the directory, its contents are visible.

```
ftp ns.rwii.com
```

(Use anonymous login)

```
cd private/moBility-1.1  
bin  
hash
```

Get the software packages that you want, for example, the Mobility software itself:

```
get mobility-b-1.1.0.tgz  
bye
```

Here is a list of the files that you may find there, and their contents.

TABLE A - 1. Mobility Files and Their Contents

Files	Contents
<code>mobility-b-1.1.0.tgz</code>	Mobility software (binary release), on-line documentation, servers, and tools.
<code>Mobility-1-1.zip</code>	The Mobility User's Guide, in Microsoft Word format (zipped).
<code>ddd-doc-3.0-5.i386.rpm</code> <code>ddd-static-3.0-5.i386.rpm</code>	Optional graphical front-end for the multithreaded debugger.
<code>egcs-core-1.0.3a.tar.gz</code> <code>egcs-g++-1.0.3a.tar.gz</code>	Enhancements for Red Hat 5.1 only, required for Mobility to run.
<code>glibc-2.0.7-20.i386.rpm</code> <code>glibc-devel-2.0.7-20.i386.rpm</code> <code>glibc-debug-2.0.7-20.i386.rpm</code> <code>glibc-profile-2.0.7-20.i386.rpm</code> <code>gdb-4.17-5.i386.rpm</code>	

Install Mobility

Installing the Mobility software itself consists of installing a directory tree of files and making one program setuid root. In addition, you must edit one configuration file for Red Hat Linux 5.1 systems.

Before installing the Mobility software, the computer must have Red Hat Linux 5.2 installed, or Red Hat Linux 5.1 plus the enhancements mentioned above in the section "Install Linux on the robot's on-board computer and prepare it for Mobility". If the computer is not on board a robot, the same operating system requirements still apply.

Untar the Mobility software, on-line documentation, etc. Make sure the `killOmniNames` program is setuid root.

```
tar xzf mobility-b-1.1.0.tgz
```

This will produce the subdirectory `mobility-b-1.1.0`.

```
cd mobility-b-1.1.0/etc
sudo chown root killOmniNames
sudo chmod u+s killOmniNames
```

For Red Hat Linux 5.1 only:

You must edit the file `scripts/config.mf` and change all three references to `gcc` or `g++` from `/usr/bin` to `/usr/local/bin` in order to get the `egcs` compiler required by Mobility.

```
vi scripts/config.mf
```

Configure an Off-board PC for Radio RS-232 Link

Magellan.

Some robots such as the Magellan can be configured to have no on-board radio. Instead, there is a radio RS-232 link from the low-level dedicated controller in the robot to an off-board PC. Do the same install of Linux, Mobility software, and the mobility user account as outlined above. Then make the following changes:

1. Determine the Linux device of the off-board PC to which the radio RS-232 link is attached. (It will be something like `/dev/cua1` or `/dev/cur3`.)
2. Edit the file `~mobility/.bashrc`. Change the 'base' alias in `~mobility/.bashrc` so that it specifies the radio device with the command line argument `-deviceport /dev/whatever`. For example, change

```
base=magellanserver
to
base='magellanserver -deviceport /dev/cua1'
```

The simplest way to test the setup is to try running the base server. The most likely cause of failure is something wrong with the serial link. Possible causes are:

- wrong Linux device name
- wrong cabling (null modem or lack thereof) between the PC and the radio
- radio set for DTE in stead of DCE or vice versa
- other radio problem

Install MOM Only on a Desktop PC

The Mobility Object Manager is a portable Java application that has been tested on Linux, Windows 95, and Windows NT 4.0. It is experimental software; it tickles occasional bugs in the underlying Java implementations. We have found it to be most reliable on Linux (in fact, it can crash Windows NT), but it has better performance on Windows platforms.

At this time, the process of installing just MOM on any computer is manual. Fortunately, it is not terribly difficult. Unfortunately, in our testing it has not worked to put the compiled Java code into one (or several) JAR files.

The general procedure is to copy five sets of files to the target machine and then create a script file that sets up the environment and launches MOM. The five sets of files are:

- the directory tree rooted at `MOBILITY_ROOT/java`,
- the directory tree rooted at `MOBILITY_ROOT/tools/jacorb_dev`,
- the file `MOBILITY_ROOT/tools/swing.jar`,
- the directory tree rooted at `MOBILITY_ROOT/icons`, and
- the Java development kit or runtime environment.

Because of the way Sun Microsystems licenses Java, if you wish to use the Java development kit, you must obtain it yourself. MOM is written for Java 1.1, and the source code must be modified for Java 1.2 (aka Java 2).

MOM-only on Linux

1. Select an installation directory. In these instructions we will call it `MOBILITY_ROOT`.
2. Copy the four file sets so as to mirror the Java portion of a full Mobility installation.
3. Obtain the Java development kit or runtime environment and install it.
4. Copy over `MOBILITY_ROOT/etc/mom` to be the basis for your launch script. Either by editing the mom script or by setting environment variables, arrange for the following settings:
 - `MOBILITY_ROOT` is the root directory of the MOM-only Mobility software installation.
 - `JAVA_HOME` is the root directory of the Java JDK or JRE installation.

- CLASSPATH is set to \$MOBILITY_ROOT/java:\$MOBILITY_ROOT/tools/jacorb_dev:\$MOBILITY_ROOT/tools/swing.jar (it need not have java-Classes in it).
- MOBILITY_NS optionally may be set to contain your site's default URL of the file containing the Naming Service's IOR.

MOM-only on Windows

1. Select an installation directory. In these instructions we will call it MOBILITY_ROOT.
2. Copy the four file sets so as to mirror the Java portion of a full Mobility installation.
3. Obtain the Java development kit or runtime environment and install it.
4. Write a batch file to set the necessary environment variables and launch MOM. Something like the following should work on your system.

```
setpMOBILITY_NS="http://your_default_host/~mobility/NamingService"
if "%1"==" " goto no_arg
    set MOBILITY_NS=%1
:no_arg
set MOBILITY_ROOT="c:\wherever\you\installed\the\software"
setpCLPATH=%MOBILITY_ROOT%\java;%MOBILITY_ROOT%\tools\jacorb_dev;%MOBILITY_ROOT%\tools\swing.jar;%CLASSPATH%
javap-class-
path:%CLPATH%p-Dcom.isr.mby.root=%MOBILITY_ROOT%p-Dcom.isr.mby.ns=%MOBILITY_NS%pcom.isr.mby.mom.Mom
```

You may prefer to set environment variables in `autoexec.bat` (Windows 95) or in “My Computer” → Properties → Environment (Windows NT) and just refer to them in the MOM launch script. We expect to find the CLASSPATH referred to in the script defined in the environment by the Java installation.

If you prefer to start MOM without specifying where to find the Naming Service (and then provide it with the Browse → Connect menu item), leave out the `-Dcom.isr.mby.mom.ns=...` command line argument.

Install Base Server Only on a Robot PC.

It's sometimes most convenient to run only the base server on the robot's on-board PC and to run all the other robot software on your desktop or lab computer. That

can be the case if you find it easier to debug software running locally, for example. If for some reason you also find it undesirable to put a full installation of Mobility on the robot's on-board computer, then you can copy only the files necessary for running the base server to the robot's on-board computer.

The on-board PC must have Linux installed and prepared for Mobility as described above. It need not have a `mobility` user account.

There is a script `MOBILITY_ROOT/etc/makeBaseOnly` that extracts from a full Mobility installation those files required for a given base server. The script has built-in knowledge about which base servers require which libraries. It collects the base server program, all the libraries it requires, the Naming Service program, and scripts to make launching them simpler into a single tar file.

The following command creates a tar file containing the base-only setup for a B14 robot.

```
makeBaseOnly b14
```

Transfer the resulting file `b14server.tgz` to the robot's on-board computer. Select a directory in which to install the file. For example, `/home/joe/robot`. Install the files in that directory. Two subdirectories will be created, `base` and `etc`. The `base` subdirectory contains the programs and libraries, and the `etc` subdirectory contains the scripts. You must edit `etc/name` and `etc/base` and enter the installation directory into each.

```
cd /home/joe/robot
tar xzf b14server.tgz
vi etc/name
```

Change the line

```
$basedir = '/home/mobility/base'
```

to

```
$basedir = '/home/joe/robot'
vi etc/base
```

Change the line:

```
dir=~mobility/base
```

to

```
dir=~joe/robot
```

DO NOT set `MOBILITY_ROOT` in your environment when you are running base-only setup. If the base and name scripts note the presence of `MOBILITY_ROOT` in the environment, they will assume a full Mobility installation.

CAUTION: In spite of our best efforts to the contrary, it is often the case that new software versions are not reflected in the `makeBaseOnly` script. You may find your base-only setup complains of a missing library or two that `makeBaseOnly` missed. If the base server fails to run and complains of missing libraries, check to see if the missing libraries are in the subdirectory `base`. if so, then there is a problem with your environment or the `etc/base` script. If not, then `makeBaseOnly` left out the library and you should manually copy it from `$(MOBILITY_ROOT)/lib` of a full Mobility installation.

External Copyright Information

JaccORB and OmniORB2

GNU Library General Public License

Version 2, June 1991

Copyright (C) 1991 Free Software Foundation, Inc.

675 Mass Ave, Cambridge, MA 02139, USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed. [This is the first released version of the library GPL. It is numbered 2 because it goes with version 2 of the ordinary GPL.]

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public Licenses are intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users.

This license, the Library General Public License, applies to some specially designated Free Software Foundation software, and to any other libraries whose authors decide to use it. You can use it for your libraries, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if

you distribute copies of the library, or if you modify it.

For example, if you distribute copies of the library, whether gratis or for a fee, you must give the recipients all the rights that we gave you. You must make sure that they, too, receive or can get the source code. If you link a program with the library, you must provide complete object files to the recipients so that they can relink them with the library, after making changes to the library and recompiling it. And you must show them these terms so they know their rights.

Our method of protecting your rights has two steps: (1) copyright the library, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the library.

Also, for each distributor's protection, we want to make certain that everyone understands that there is no warranty for this free library. If the library is modified by someone else and passed on, we

want its recipients to know that what they have is not the original version, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that companies distributing free software will individually obtain patent licenses, thus in effect

transforming the program into proprietary software. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License, which was designed for utility programs. This license, the

GNU Library General Public License, applies to certain designated libraries. This license is quite different from the ordinary

one; be sure to read it in full, and don't assume that anything in it is the same as in the ordinary license.

The reason we have a separate public license for some libraries is that they blur the distinction we usually make between modifying or adding to a program and simply using it. Linking a program with a library, without changing the library, is in some sense simply using the library, and is

analogous to running a utility program or application program. However, in a textual and legal sense, the linked executable is a combined work, a derivative of the original library, and the ordinary General Public License treats it as such.

Because of this blurred distinction, using the ordinary General Public License for libraries did not effectively promote software sharing, because most developers did not use the libraries. We

concluded that weaker conditions might promote sharing better.

However, unrestricted linking of non-free programs would deprive the users of those programs of all benefit from the free status of the libraries themselves. This Library General Public License is intended to permit developers of non-free programs to use free libraries, while

preserving your freedom as a user of such programs to change the free libraries that are incorporated in them. (We have not seen how to achieve this as regards changes in header files, but we have achieved it as regards changes in the actual functions of the Library.) The hope is that this will lead to faster development of free libraries.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a “work based on the library” and a “work that uses the library”. The former contains code derived from the library, while the latter only works together with the library.

Note that it is possible for a library to be covered by the ordinary General Public License rather than by this special one.

Terms and Conditions for Copying, Distribution, and Modification

Section 1. This License Agreement applies to any software library which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Library

General Public License (also called “this License”). Each licensee is addressed as “you”.

A “library” means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

The “Library”, below, refers to any such software library or work which has been distributed under these terms. A “work based on the Library” means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term “modification”.)

“Source code” for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

Section 2. You may copy and distribute verbatim copies of the Library's complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the

Library.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

- Section 3. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Step Section 1. above, provided that you also meet all of these conditions:
- i. The modified work must itself be a software library.
 - ii. You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.
 - iii. You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License.
 - iv. If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful. (For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in

themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

Section 4. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

Section 5. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange. If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

Section 6. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a “work that uses the Library”. Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

However, linking a “work that uses the Library” with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a “work that uses the library”. The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

When a “work that uses the Library” uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not.

Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

Section 7. As an exception to the Sections above, you may also compile or link a “work that uses the Library” with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer's own use and reverse engineering for debugging such modifications.

You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

- i. Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable “work that uses the Library”, as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)
- ii. Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.
- iii. If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.

- iv. Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

For an executable, the required form of the “work that uses the Library” must include any data and utility programs needed for reproducing the executable from it. However, as a special exception,

the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on

which the executable runs, unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot *use both them and the Library together in an executable that you distribute*.

Section 8. You may place library facilities that are a work based on the Library side-by-side in a single library together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:

- i. Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.
- ii. Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

Section 9. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

Section 10. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance

of this License to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.

Section 11. Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

Section 12. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent license would not permit royalty-free redistribution of the Library by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Library.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

Section 13. If the distribution and/or use of the Library is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In

such case, this License incorporates the limitation as if written in the body of this License.

Section 14. The Free Software Foundation may publish revised and/or new versions of the Library General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library specifies a version number of this License which applies to it and “any later version”, you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Library does not specify a license version number, you may choose any version ever published by the Free Software Foundation.

Section 15. If you wish to incorporate parts of the Library into other free programs whose distribution conditions are incompatible with these, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

No Warranty

Section 16. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE LIBRARY “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

Section 17. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCI-

DENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Libraries

If you develop a new library, and you want it to be of the greatest possible use to the public, we recommend making it free software that everyone can redistribute and change. You can do so by permitting redistribution under these terms (or, alternatively, under the terms of the ordinary General Public License).

To apply these terms, attach the following notices to the library. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the “copyright” line and a pointer to where the full notice is found.

<one line to give the library's name and a brief idea of what it does.>

Copyright (C) <year> <name of author>

This library is free software; you can redistribute it and/or modify it under the terms of the GNU Library General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version. This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Library General Public License for more details.

You should have received a copy of the GNU Library General Public License along with this library; if not, write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

Also add information on how to contact you by electronic and paper mail.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a “copyright disclaimer” for the library, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the library `Frob' (a library for tweaking knobs) written by James Random Hacker.

<signature of Ty Coon>, 1 April 1990

Ty Coon, President of Vice

Glossary

Angular reflection (sonar). The angle at which the sonar beam bouncing off an obstacle is directed back at that sonar. This angle depends on the texture and other characteristics of the obstacle's surface, and its position relative to the sonar. **API:** Application program Interface.

BeeSoft. An iRobot robot application interface software package for mobile autonomous robots. As Mobility supplants BeeSoft, iRobot's official support for BeeSoft will be frozen.

ContainedObject. This Mobility defined interface is support by nearly every object that is part of a Mobility system.

Container. See Mobility Container.

CORBA (Common Object Request Broker Architecture). An OMG standard that defines the protocols, programming language mappings and programming-language-independent models for building distributed object-oriented software systems.

Dead reckoning. Using the latest wheel encoder readings to determine the robot's position at any given time relative to its position when it started moving.

Drift. The error that accumulates in odometry due to wheel skid and other tracking errors, as the robot continues to move along a heading.

DynamicObject. This Mobility-defined interface is supported by all objects that provide dynamic notification of updates.

Emergency stop button. A button or switch on your robot that immediately stops all the motors, thereby stopping the robot's motion. Thereafter, the robot will be in limp mode, that is, movable by hand.

Externalization Interface. Mobility defines these interfaces for future extensions of the basic Mobility system that support save and restore of Mobility robot controllers.

Externalization object. See Externalization Interface.

Factory Object Interface. Mobility defines this interface so Mobility Components can be dynamically created under program control. (NOTE: This feature will be supported in an upcoming release of Mobility.)

GNU. Software developed and licensed by the Free Software Foundation; necessary for proper installation of Linux.

Grid cell. A unit in a grid-based map, containing a value which indicates the presence or absence of an obstacle in the corresponding region of the physical environment

Guarded motion. A mode of robotic travel in which the robot automatically tries to avoid collisions with those obstacles that it can perceive with its sensors.

Heading. The direction in which the robot is moving at any given time.

IDL (Interface Definition Language). A programming-language-independent language for defining objects and interfaces standardized by the OMG within the CORBA 2.X standard.

Infrared sensor. A sensor that emits Infrared light C light in that part of the electromagnetic spectrum above 0.75 millimeters in wavelength. The robot estimates the approximate distance to an obstacle by measuring the intensity of the reflection bounced back from the obstacle's surface.

I/OStream. This interface is part of the externalization functionality supported by upcoming releases of Mobility. See Externalization Interface.

Interface Definition Language. See IDL.

JDK. JAVA development kit.

Joystick. A human-operated control device that allows simultaneous, tele-operated rotation and translation of an autonomous mobile robot. Each iRobot robot is shipped with a joystick which can be plugged in the robot's labeled joystick socket. (Check the documentation for your own robot.) Special Note: The joystick shipped with some iRobot robots has been specially modified to control your robot. It will not work with other equipment or games you might have.

JRE. JAVA run-time environment.

Kill switch. Another term for the emergency stop button on your robot.

Laser rangefinder. A low-intensity, eye-safe laser on board some iRobot robots that detects the proximity of an obstacle by “time of flight” C the time it takes for an invisible, concentrated beam of light to bounce off an obstacle and return. The laser rangefinder measures in the nanosecond (billionths of a second) range. The laser rangefinder itself does not provide angular resolution, but ranging resolution. A precise encoder provides the angular resolution. In sum, the laser rangefinder package provides a much higher angular resolution than the sonar sensors, rendering much more precise measurements, with an average accuracy of plus or minus five centimeters.

Linux. A completely free re-implementation of POSIX specifications, with SYSV and BSD extensions (meaning it looks like UNIX but does not come from the same source code base), available both in source code form and binary form. It is copyrighted by Linus B. Torvalds and other contributors, and is freely redistributable under the terms of the GNU public license. It is not in the public domain, nor is it shareware.

MCF. (See Mobility Class Framework.

MROM. See Mobility Robot Object Model.

Mobility Class Framework (MCF). The programming language-specific implementation of the Mobility Robot Object Model (MROM) interfaces you can use to speed your implementation of Mobility-based software systems. The MCF mirrors the Mobility Robot Object Model and handles much of the grunt work involved in programming robot software. By deriving a new class from the Mobility Class Framework you can easily add your own sensors, actuators, behaviors, perceptual processes and data classes to the system. See MCF.

Mobility Container. Any Mobility Component that also supports the interface needed for containing other Mobility Components. The additional interface is the ObjectContainer interface.

Mobility Component. Any Mobility-based object that supports the core set of interfaces to be utilized by other Mobility Components. This designation refers to an object that implements a basic set of interfaces: ContainedObject, PropertyContainer, and StateChangeHandler.

Mobility Robot Object Model (MROM). An object model defined using CORBA 2.X IDL files that defines the interfaces to each Mobility object in a programming language-independent manner.

Mobility robot integration software. A distributed, object-oriented toolkit for building control software for single and multi-robot systems. Mobility consists of the following components:

- a set of software tools
- an object model for robot software
- a set of basic robot control modules
- an object-oriented class framework to simplify code development

Mobility Object Manager (MOM). The Mobility Object Manager is a graphical user interface written in Java that allows you to launch programs, create objects, edit object properties, connect and configure objects and control which objects are active. MOM also lets you launch a variety of object viewers that provide visualization of your robot’s actuators, sensors, algorithm outputs and debugging information all from a central management point. MOM uses the core Mobility interfaces to provide access and management functionality for a system of Mobility Components; it serves as the “integrated view” of your robot system in Mobility

Mobility State Component. Any Mobility Component that supports the DynamicObject interface for registration of interested objects that implement the StateChangeHandler interface in addition to the normal Mobility Component interfaces.

MOM. See Mobility Object Manager.

Motion controller. The electronics that integrates the measurements of the wheel encoders to attempt to estimate the robot’s current position at any time with respect to its original position, that is, where it was when it started rolling. It also controls the motion of the motors that drive the robot’s wheels.

Name server. This facility allows your software to access the many elements of multi-robot software systems. The top of the Mobility Robot Object Model is the CORBA 2.x standard naming service. Mobility’s top level name server contains a directory of robot objects or shared support objects. (Sometimes called name service or naming service.)

Object. A separate unit of software with identity interfaces and state. In Mobility, objects represent abstractions of whole robots, sensors, actuators, behaviors, perceptual processes and data storage. Together, these objects provide a flexible model of a robot system that can be reconfigured as new hardware, new algorithms and new applications are developed for a robot system. A MobilityComponent is an object that supports a defined set of interfaces. See MobilityComponent.

Object Request Broker. A communication management library that allows transparent access to objects in different address spaces on the same or other computer systems.

Object Descriptor. A ContainedObject's self-describing data structure that identifies the class of the object, any special service parameters and the instance name of the object.

ObjectContainer. The ObjectContainer interface is supported by any MobilityComponent that can contain, locate and manage other Mobility Components.

ObjectStatus. An enumeration that describes the basic states for all mobility objects. An object can be Un-initialized, in ActiveWaiting state, or ActiveAlerted state.

Odometry. The robot's way of trying to keep track of where it is relative to where it was when it started moving, with the help of its wheel encoders in conjunction with its motion controller. While this measurement is highly accurate for short distances, error can and does accumulate as the robot travels further afield.

OMG (Object Management Group). A consortium of companies dedicated to development of open, standardized methods for distributed object computing. Their primary standard is CORBA and its extensions.

See <http://www.omg.org>

ORB. See Object Request Broker.

Orientation. The direction in which the robot is headed, measured with respect to some standard reference frame.

Property. A piece of data that affects the operation of a Mobility Component. These can be scaling factors, device port names or configuration or calibration data used by each Mobility Component to perform its task.

PropertyContainer. This interface is supported by all Mobility Components that can have properties. See property.

Property Descriptor Structure. Identifies and names properties within Mobility.

Robot. In Mobility, a collection of dynamically connected objects are used to represent the robot. More generally, an electro-mechanical entity equipped with sensory apparati and manipulative appurtenances, designed to perform routine or repetitive tasks, or to operate in environments lethal or dangerous to humans. The term robot was adapted from the Czech word robota, meaning forced labor, by Czech novelist and dramatist Karel Capek in his dystopian 1920 play, R.U.R. (Rossum's Universal Robots), a work exploring the dangers of technological progress.

Rotation. Leftward or rightward motion, centered on the robot's axis. Can be done either in place, or in conjunction with translation.

Sensor. An electro/mechanical, electrical, optical or other device that detects, measures and/or records information about phenomena in its surroundings and conveys that data in a usable form to a human, computer, or other entity that is prepared to use it. Mobility uses several kinds of sensors, such as sonar sensors and infrared sensors.

Sonar sensor. An ultrasonic transducer that generates a sonic, or sound, wave, called a ping, that travels outwards in an expanding cone, and listens for an echo. For each reading, the total time between the generation of the ping and the receipt of the echo, coupled with the speed of sound in the robot's environment, generates an estimate of the distance to the object that bounced back the echo.

Specular reflection. An anomaly that can occur when a sonar sensor's ping bounces off an obliquely-angled object onto another object in the environment, which then, in turn, returns an echo to the sonar. This effect can bedevil the sonars, causing them to overestimate the distance between the robot and the nearest obstacle.

State Change Handler. This interface is supported by Mobility Components and allows them to register for dynamic update notifications instead of polling for information they need.

System Components. Abstractions of robot hardware, software, behaviors, data stores and perceptual processes. Typical System Components provided for a robot would include odometry, tactile sensors, sonar sensors and an actuator component.

SystemModuleComponent. Provides a set of interfaces that allow organized access to all the components of each robot. Each SystemModuleComponent contains a set of SystemComponents. You can add, remove and discover services dynamically at runtime through the interfaces provided by the SystemModuleComponent. The SystemModule component is a special implementation of a Mobility Component that is designed to be the "root" of the objects within a single process.

Tactile sensors. Pressure-sensitive switches, mounted around the robot's perimeter.

Tele-operation. A hands-off direction of the robot's movements, via user program control.

Translation. Forward or rearward motion of the robot, along the heading to which it is currently directed.

Velocity. The speed at which the robot moves along its heading.

Wheel encoders. Devices in the robot base that keep track of the revolutions of the robot's wheels. Used for odometry.

Wheel skid. Errors in robot wheel tracking, caused by routine hazards in real-world research and operational environments such as slippery floors, carpeting, doorjamb, and the like. These errors are mitigated somewhat by the redundancy and cross-checking supplied by the robot's other sensors.