Barobo™

Mobot and Linkbot User's Guide

Version 2.0.0

How to Contact Barobo

Mail	Barobo, Inc. 221 G Street, Suite 204 Davis, CA 95616
Phone	+ (530) 231-5178
Web	http://www.barobo.com
Email	info@barobo.com

Contents

1 Introduction

The Mobot and Linkbot are breakthrough modular robots. A single module is a fully functional robot capable of performing many possible motions. The robots can also be used as a building block to create robots with different geometric configurations.

As shown in Figure 2, The Linkbot-I and Linkbot-L are architecturally similar and they will may be jointly referred to as the "Linkbot-I/L" in the remainder of this document. "Mobot" will be used to refer specifically to the robot shown in Figure 1. The term "robot" will refer all forms of Mobots and Linkbots, including the Mobot, -I, and -L. This documentation introduces the basic computer setup required for controlling the Mobot, as well as several demo programs and a complete reference for all API function provided with the `CMobot, CLinkbotI, CLinkbotL, CMobotGroup, CLinkbotIGroup`, and `CLinkbotLGroup` classes.

The `CMobot` and `CLinkbot` libraries are collections of functions geared towards controlling the motors and reading sensor values of a robotic modules. The functions are designed to be intuitive and easy to use. Various functions are provided to control or obtain the speed, direction, and position of the motors.

1.1 Introducing the Mobot

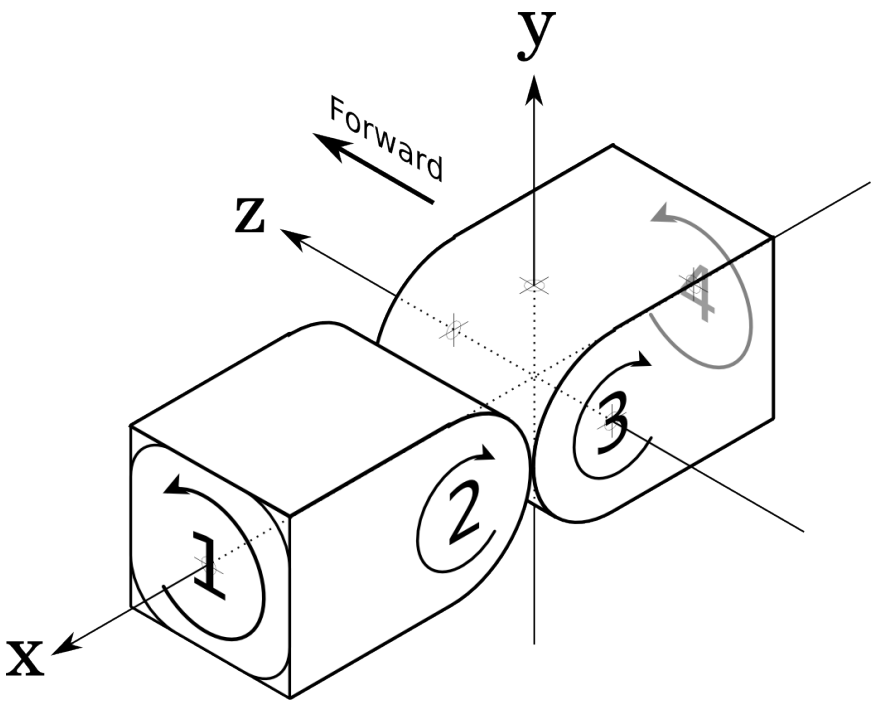


Figure 1: A schematic diagram of a Mobot module.

Figure 1 shows a schematic diagram displaying the locations and positive directions of the four joints of a Mobot module. The joints 1 and 4 shown in the figure are fully rotational and have no joint limits. Joints 2 and 3, however, can only move in the range -90 to +90 degrees.

1.2 Introducing the Linkbot-I/L

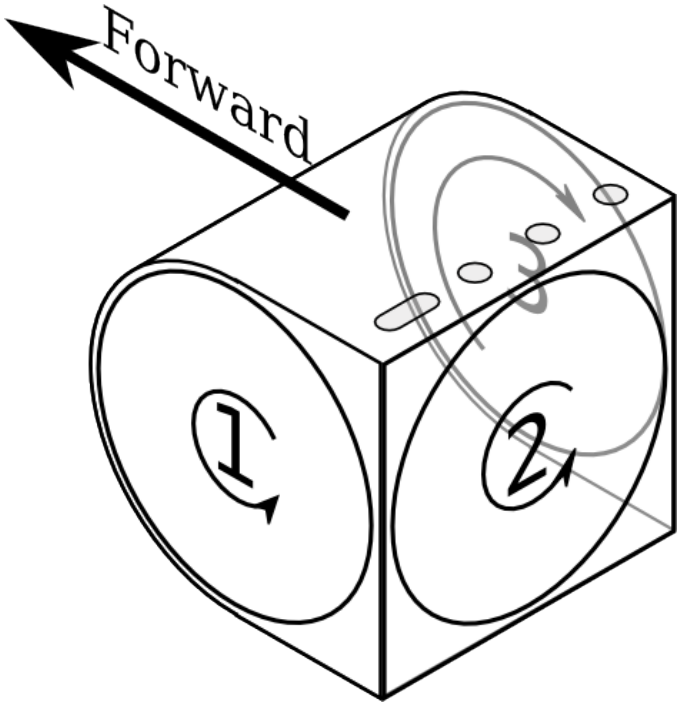


Figure 2: A schematic diagram of a Linkbot-I/L module.

Figure 2 shows a schematic diagram displaying the locations and positive directions of the four joints of a Linkbot-I/L module. Note that while Figure 2 displays three joints, only joints 1 and 3 are able to move on the Linkbot-I, and only joints 1 and 2 are able to move on the Mobot-L.

2 Configuring the Mobot for Remote Control

Mobot modules should be configured the first time they are used with a new computer. The process informs the computer which Mobots it is allowed to connect to. The is also necessary for certain functions in the `CMobot` API, such as `connect()`, to determine which Mobots to connect to.

The configuration is performed through the Barobo BaroboLink program. The remainder of the section contains step-by-step instructions and screenshots showing how to configure your Mobots.

2.1 Bluetooth Pairing for Mac OS X

For the Mac OS X operating system, an additional initial step needs to be performed. Windows users may skip these steps directly to section 2.2 on page 17.

To begin the pairing process, click on the Bluetooth icon on the applet tray, typically located on the top right portion of the screen. Figure 3 shows the Bluetooth icon on a typical Mac OS X system. Click on the icon to drop down a menu.

Figure 3: The Bluetooth Icon on a Mac OS X System.

The menu that is dropped down should appear similar to the one shown in Figure 4. Next, click on the menu item labeled "Set Up Bluetooth Device...".

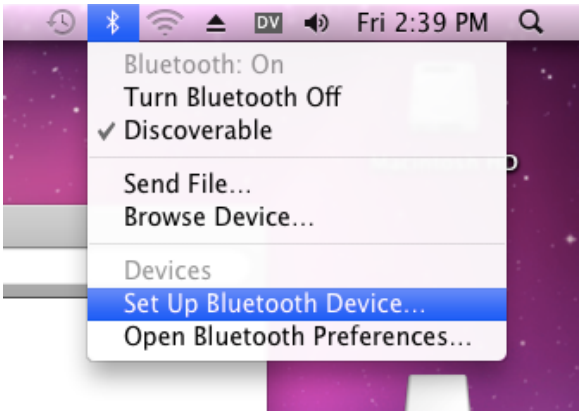


Figure 4: The Bluetooth Icon Dialog.

The previous step should start a dialog used to find and pair with Bluetooth devices. Make sure the Mobot is turned on, and it should appear as a device in the dialog, as shown in Figure 5. If there are multiple Mobot devices, as shown in the figure, you may pinpoint the specific Mobot to pair with by looking inside the battery compartment on the side of the Mobot with the power switch. Inside, there is a sticker with the Mobot's unique ID number. The last four digits of the ID number will coincide with the Mobot's Bluetooth ID number. Once the correct Mobot is selected, click on the "Continue" button.

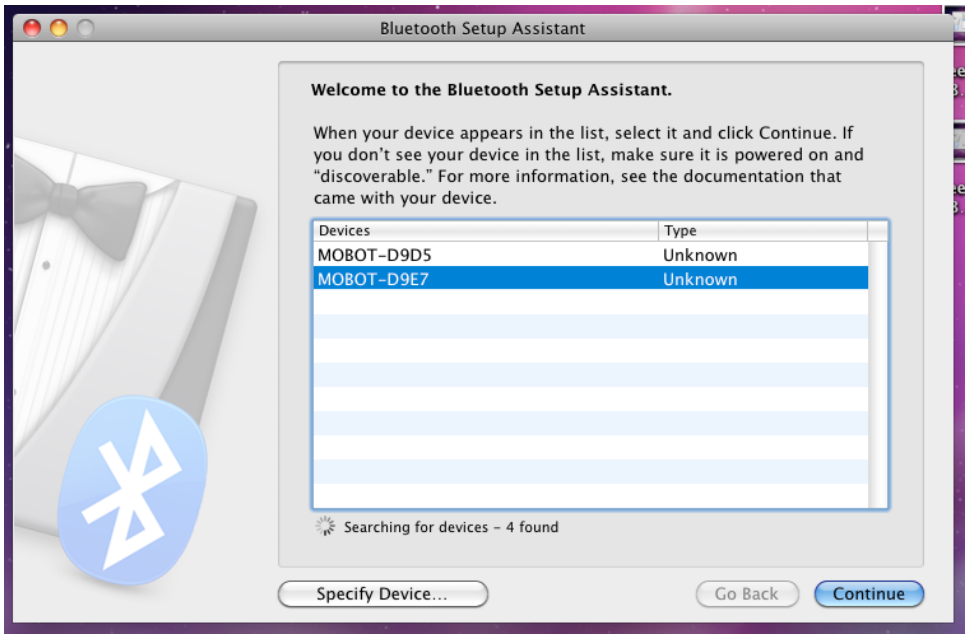


Figure 5: The Bluetooth Pairing Dialog.

At this point, the pairing dialog will attempt to pair with the Mobot using a default pairing key, "0000", as shown in Figure 6. This initial pairing attempt will fail because the Mobot pairing key is hard-coded to a value of "1234". Click on the button labeled "Passcode Options..." in order to modify the passcode options. This should bring up the dialog shown in Figure 7.

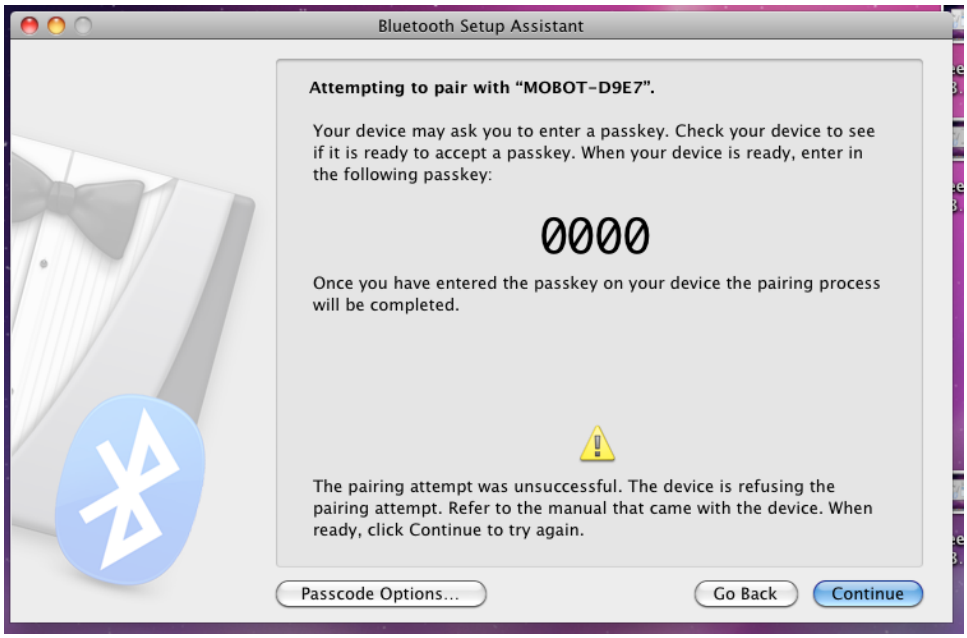


Figure 6: The Bluetooth Pairing Dialog.

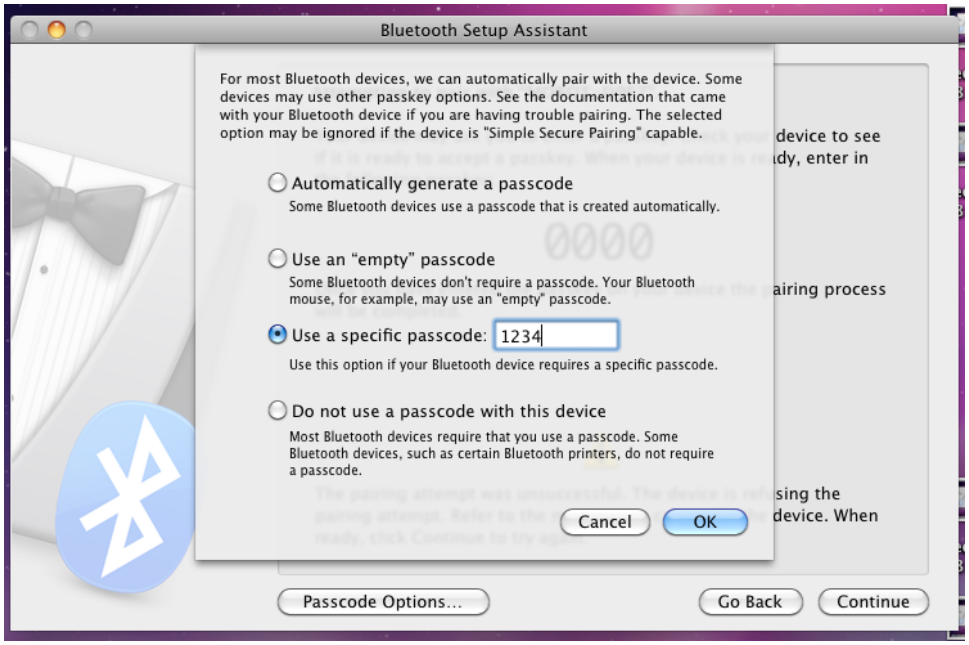


Figure 7: The Bluetooth Pairing Dialog.

In the dialog shown in Figure 7, select the option to "use a specific passcode", and enter the value "1234". Next, click on the "OK" button.

You should be greeted with the dialog shown in Figure 8, which indicates that the Mobot has successfully paired. At this point, you may continue on with the rest of the pairing process, or repeat this step for all Mobots that need to be paired before continuing.

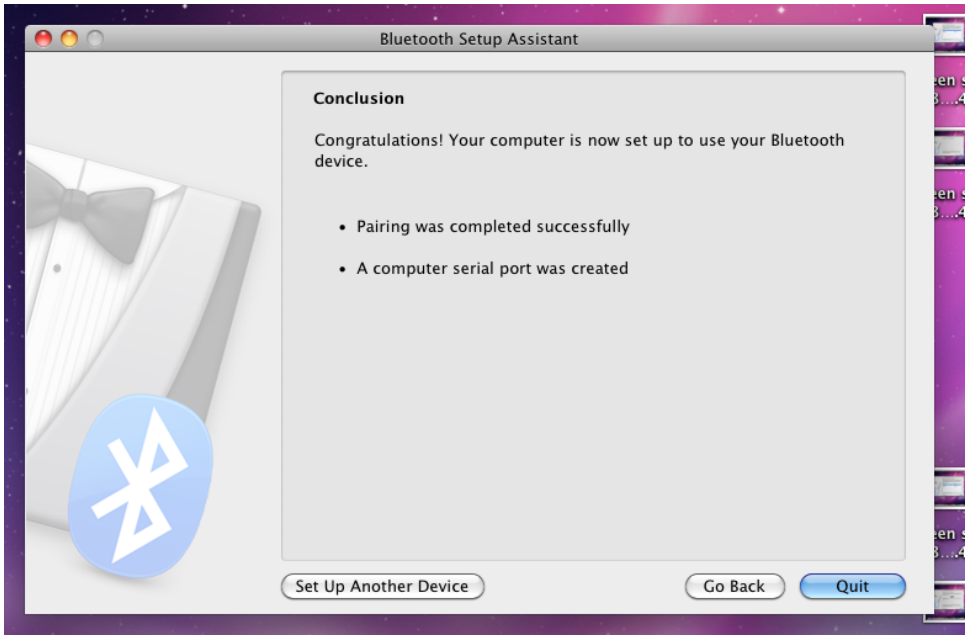


Figure 8: The Bluetooth Pairing Success Dialog.

2.2 Adding Bluetooth Addresses of Robots in BaroboLink

First, start the provided Barobo BaroboLink application. In Windows, start the provided Barobo Robot Control Program by clicking on the icon labeled "BaroboLink" on your desktop, as shown in 9. On Mac OS X systems, the BaroboLink application is located inside the "Applications" folder in Finder. The control dialog as shown in Figure 10 should pop up.

Figure 9: The icon for Barobo BaroboLink.

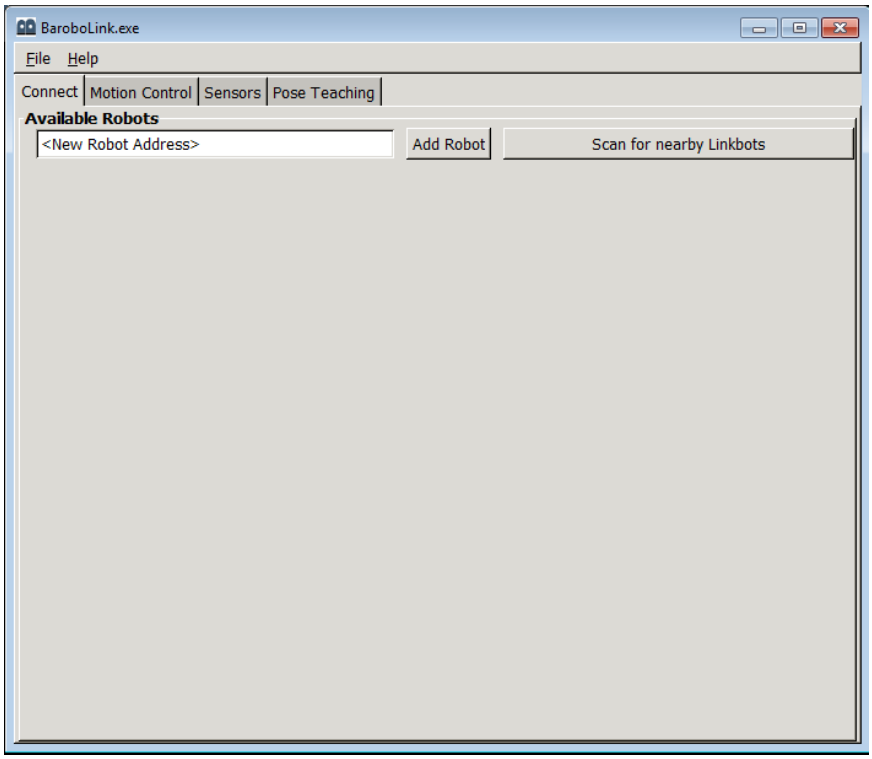


Figure 10: The opening screen for BaroboLink.

The BaroboLink application, as seen in Figure 10, is organized into several different sections, each denoted by a tab. Upon startup, the default tab that is selected is the "Connect" tab. This section of the application allows you to manage, connect, and disconnect from Mobots.

First, add your Mobot address(es) into BaroboLink by typing in the address and clicking on the "Add Robot" button. The dialog should now something like the figure shown in Figure 11.

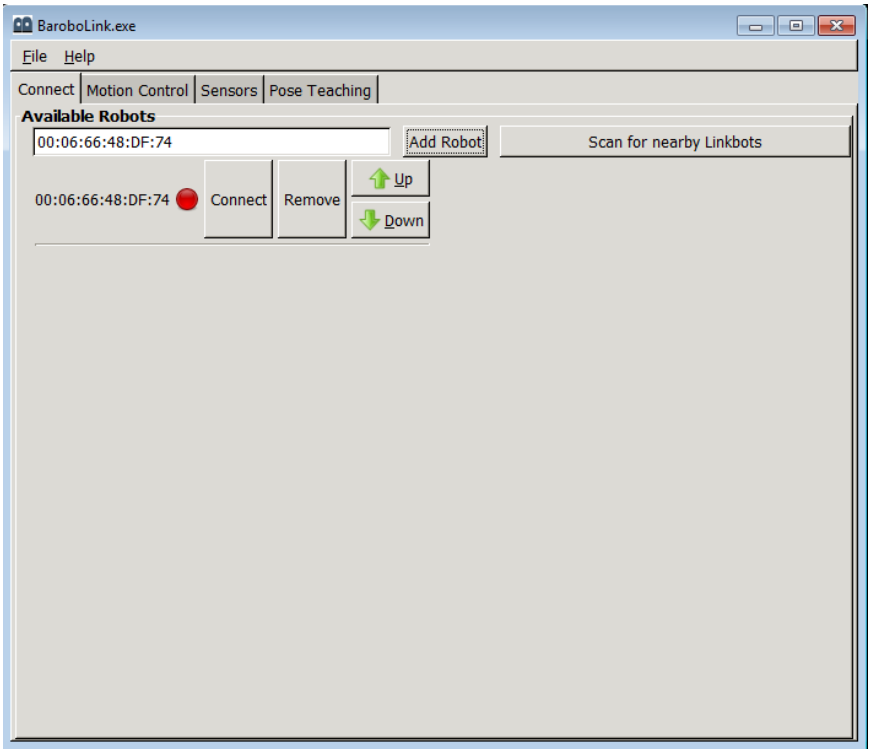


Figure 11: Configuring robot Bluetooth connection.

Each additional Mobot added occupies its own row in the dialog. After the addition of another robot, the dialog appears as shown in Figure 12.

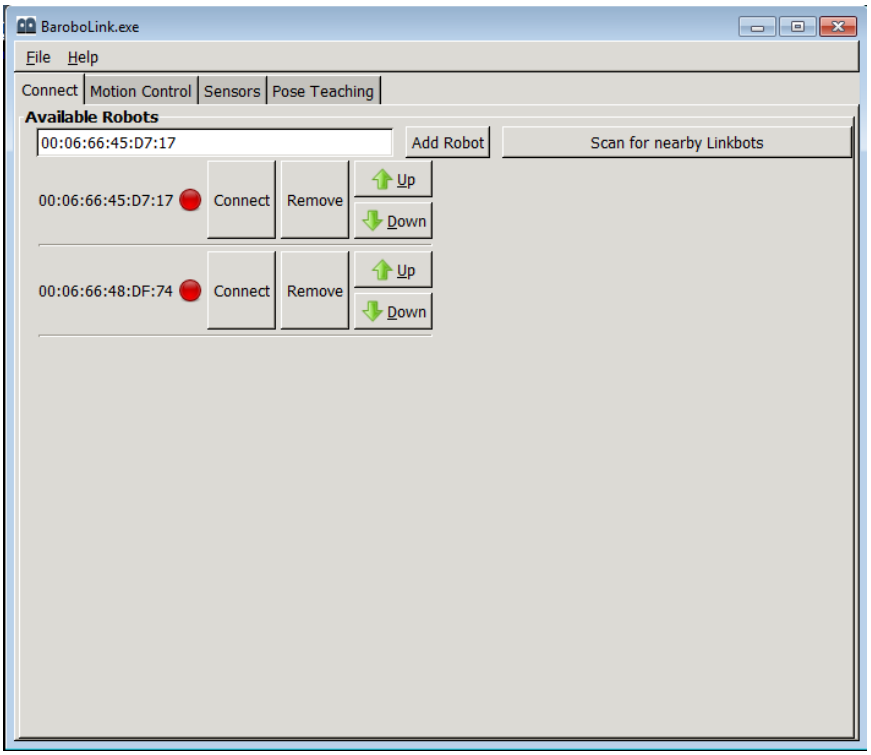


Figure 12: The Connection dialog two Mobot entries.

Once new addresses are added, BaroboLink will remember the addresses in the future. The red dot next to the added addresses indicate the current connection status of the robot. A red dot indicates that the Mobot is not currently connected, and a green dot indicates that the Mobot is properly connected. To connect to a robot, make sure the robot is on by moving the power switch to the "RUN" position, and click on the button labeled "connect" inside BaroboLink.

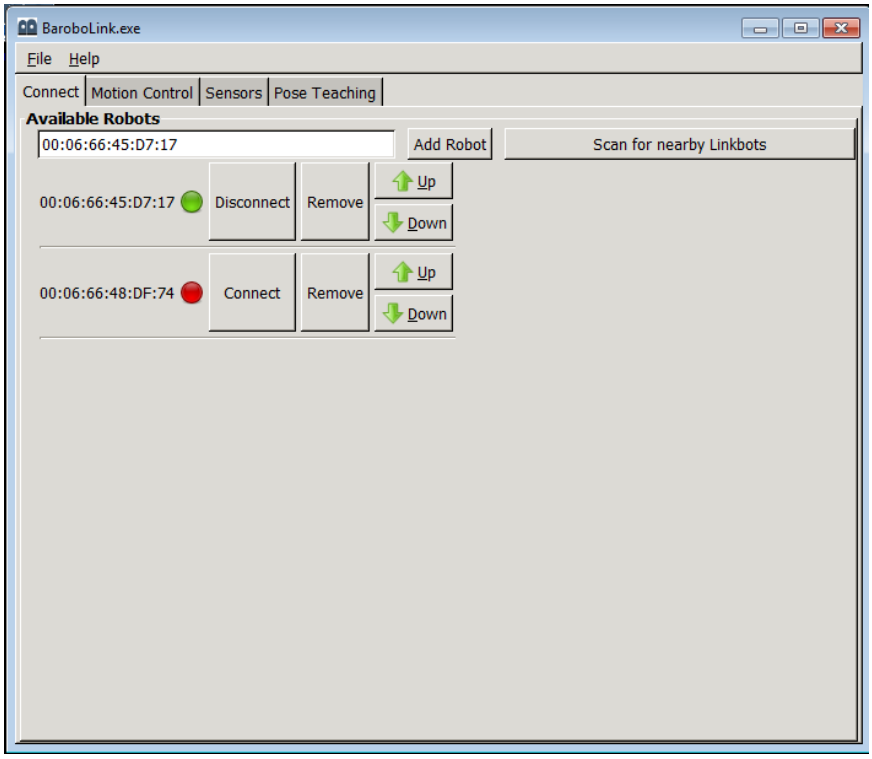


Figure 13: The Connection dialog showing a connected Mobot.

If the connection succeeds, the red dot should turn green to indicate that a connection has been established, as shown in Figure 13. Should the connection fail, please check that the address has been entered correctly, the Mobot is on, your computer has Bluetooth capability, and that the Mobot is within range of your computer, and try again. Note that in Figure 13, an additional Mobot had been added to the list of known Mobots to illustrate how the dialog behaves as more Mobots are added. Each additional Mobot appears as its own row in the dialog.

Once the connection succeeds, the "Connect" button for that Mobot turns into a "Disconnect" button. Clicking on the "Disconnect" button disconnects the Mobot.

Furthermore, please note that Bluetooth devices have a maximum limit of connected devices. The maximum limit is 7 devices connected simultaneously. This means that a maximum of 7 Mobots may be connected to a computer simultaneously, if no other devices are connected. If, for instance, two other devices are currently connected, such as a Bluetooth enabled phone and a Bluetooth mouse, then the maximum number of connected Mobots decreases to 5, for a total of 7 connected devices.

2.3 Upgrading the Mobot's Firmware

Upon connecting to Mobots, BaroboLink is able to determine if the firmware currently residing on the Mobot is up to date. If the firmware is not up to date, BaroboLink will draw an "Upgrade Firmware" button, as shown in Figure 14.

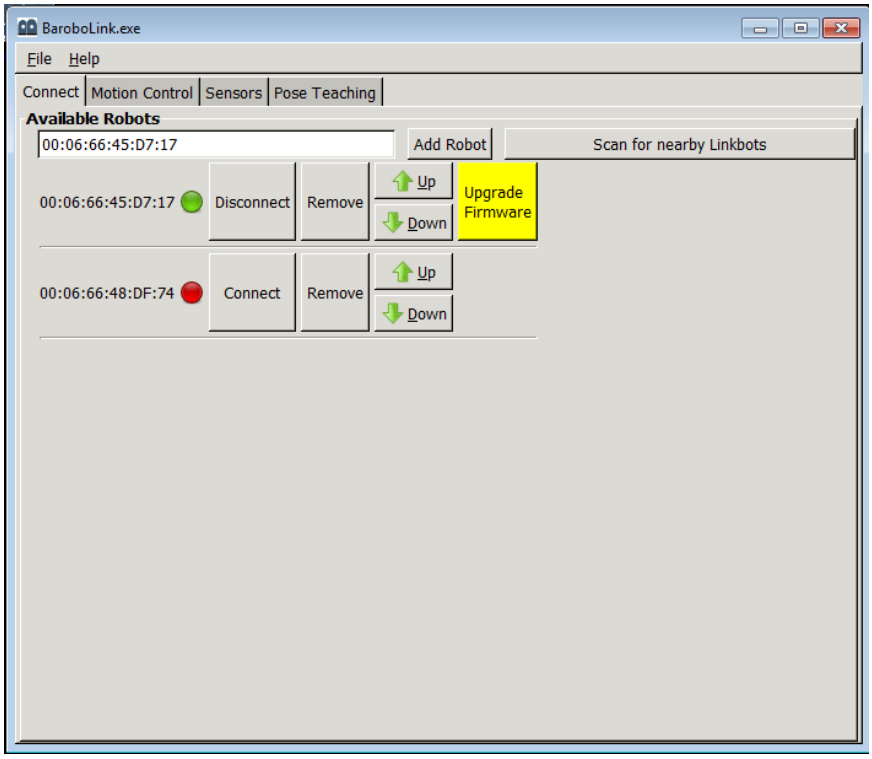


Figure 14: The BaroboLink connect dialog showing a connected Mobot with out-of-date firmware.

Upgrading the firmware of a Mobot is optional, but Mobots with out-of-date firmware may have missing functionality. It is recommended to update the firmware of out-of-date Mobots as soon as possible. The update process is started by clicking on the button labeled "Upgrade Firmware".

After clicking the "Upgrade Firmware" button, the dialog shown in Figure 15 is shown. This dialog instructs the user to press the "A" button on the Mobot and waits until the button-press is detected. The entire upgrade process can be canceled by clicking on the "Cancel" button. Once the "A" button is pressed, the upgrade process proceeds to the next screen.

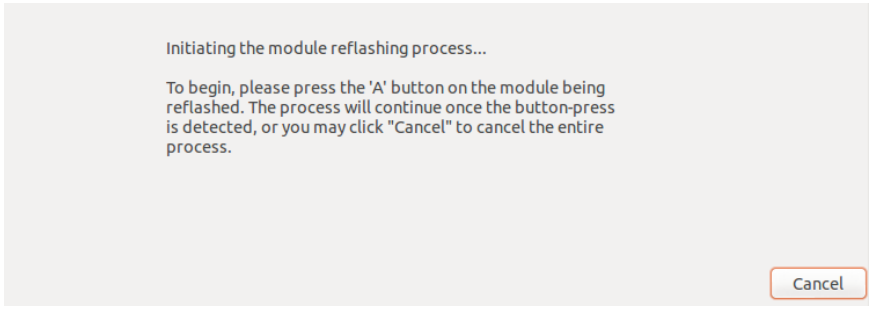


Figure 15: The first step in the upgrade firmware process.

The next dialog in the process, shown in Figure 16, instructs the user move the switch to the middle position, marked "PROG". After moving the switch into this position, click on the "Continue" button to continue.

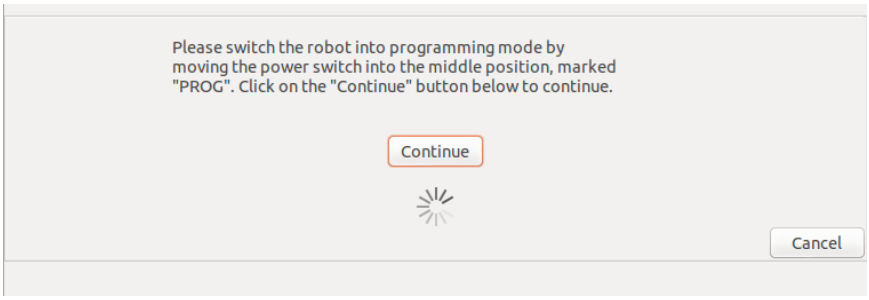


Figure 16: The second step in the upgrade firmware process.

The final upgrade dialog as shown in Figure 17 is shown after clicking on "Continue". At this point, the robot should be turned on by moving the switch to the "RUN" position. Once the robot is on, the update process begins immediately. The progress bar shown in Figure 17 should begin moving until completion. The entire process may take several minutes, and the Mobot should not be turned off or moved away from the computer during this time, or the firmware may become corrupted.

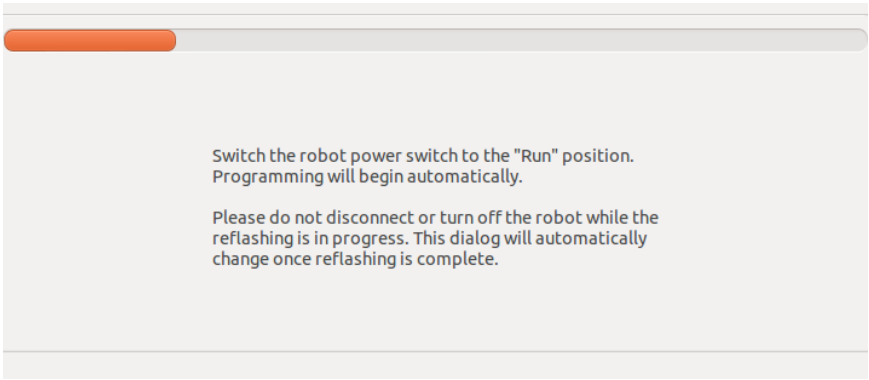


Figure 17: The third step in the upgrade firmware process.

2.4 Forcing a Mobot's Firmware to Reset

In the rare event that a Mobot's firmware gets corrupted, it is possible to reset the firmware to a "safe" version. Symptoms of a Mobot whose firmware may have been corrupted include the following:

- The blue light on the Mobot does not turn on.
- The Mobot resets randomly.
- The default button actions do not work.

To begin the reset process, click on "File → Reset Mobot Firmware" in the application menu. This will bring up the dialog shown in 18. To begin the process, type in the address of the Mobot to be re-flashed and click on the button labeled "begin".

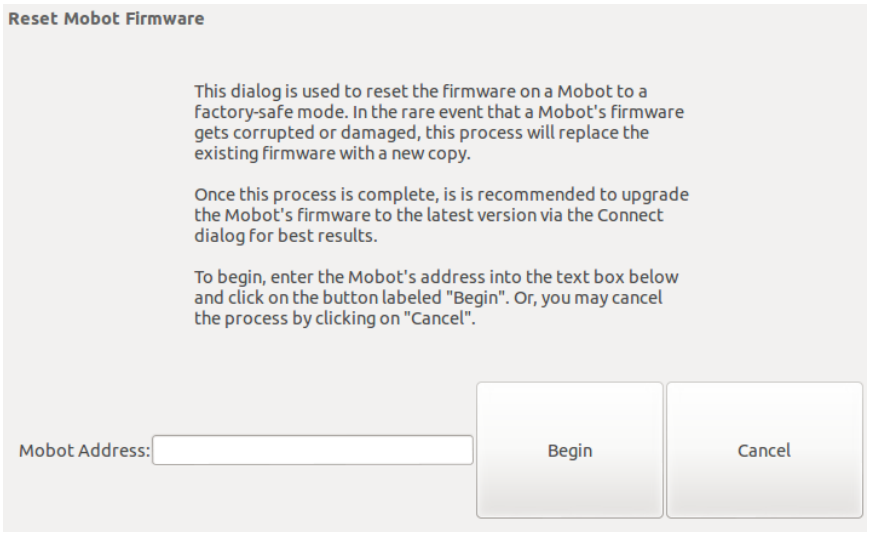


Figure 18: The opening dialog for force-reflashing a Mobot.

The rest of the process is similar to the normal upgrading process of the Mobot. Once the "safe" version of the firmware has been flashed onto the Mobot, it is recommended to connect to the Mobot using BaroboLink and upgrade the firmware back to the most up-to-date version.

3 Controlling Linkbots Without a Computer

If you have access to more than one Linkbot, they can be used to control each other in several ways. To begin, the modules must be paired together using a process called BumpConnect.

3.1 BumpConnect

BumpConnect is a process that is used to pair modules together. The process can also be used to add additional slave robots to an existing group. All BumpConnected groups of Mobots contain a single master which can control one or more slaves.

To BumpConnect two unpaired Linkbots:

1. Press and hold the 'B' button on the Linkbot you want to be the master.
2. Press and hold the 'B' button on a second Linkbot.
3. Gently bump the two modules together.
4. Release the 'B' buttons.
5. After a second or so, the LED colors on both robots should change. The Master robot will blink, cycling between Blue and a randomly chosen group color. The Slave should display the randomly chosen group color.
6. If nothing happens after 5 seconds, try again starting from step 1.

To BumpConnect a paired Linkbot with an unpaired Linkbot:

1. Press and hold the 'B' button on both robots.
2. While holding the 'B' buttons, gently bump the two robots together.

3. The unpaired robot should take on the color of the paired robot. If this does not happen after 5 seconds, try the process again starting from step 1.

3.2 Pose Teaching Mode

After robots have been paired with each other, they enter the default "Pose Teaching" Mode. Initially, there are zero recorded poses. In this mode, the buttons have the following functions:

- The Power or "X" button: If the robots are not playing poses, this button deletes all recorded poses.
- The 'A' button: If the robots are not playing poses, this button records a new pose.
- The 'B' button: If there are no recorded poses, the 'B' button makes the robot switch into "TiltDrive" mode. If there are recorded poses, the 'B' button begins playing the poses. If the robot is currently playing poses, the 'B' button stops playing poses.

3.3 TiltDrive Mode

While the group is in TiltDrive mode, the Master robot will flash its LED colors between green and the group color. In TiltDrive mode, the master module can be tilted forward, backward, and side-to-side to drive the slave modules.

In this mode, the buttons have the following functions:

- The 'B' button: Changes the current mode from "TiltDrive" mode to "CopyCat" mode.

3.4 CopyCat Mode

While the group is in CopyCat mode, the Master robot will flash its LED colors between light-blue and the group color. In CopyCat mode, all of the slave modules will move their joints to match the position of the joints on the Master module. If there are intermixed -I and -L Linkbot modules, joint 2 will match the angle on the Master's joint 3, and vice versa.

In this mode, the buttons have the following functions:

- The 'B' button: Changes the current mode from "CopyCat" mode to "Pose Teaching" mode.

4 Configuring the Linkbot-I/L for Remote Control

In order to control Linkbot-I/L modules, at least one Linkbot-I/L module must be directly connected to the computer using a Micro-USB cable. The connected module acts as a coordinator, relaying messages between the computer and other modules connected wirelessly.

When connecting a module to the computer for the first time, Windows may need to install special drivers in order to communicate with the module. The drivers are located in the "C:/Ch" directory.

The first time BaroboLink is started, it will try and search for a Linkbot-I/L dongle that is connected using a USB cable.

There are a couple ways to associate Linkbot-I/L modules with BaroboLink. First, BaroboLink is capable of scanning for Linkbot-I/L modules which are powered on, within range, and not currently connected to any other computer. Simply click on the button labelled "Scan for nearby Linkbots" to start scanning. Otherwise, Linkbot-I/L's may be manually added to BaroboLink by typing the Mobot's serial ID directly into the "Add Robot" text box. The Mobot's serial ID is a string of four letters and numbers located on the black sticker directly above buttons A and B on the module. Once the robots are added, they may be connected by clicking on the "Connect" button for each module. Unlike the Mobot, there is no maximum limit to the number of simultaneously connected modules.

5 The "Motion Control" Panel

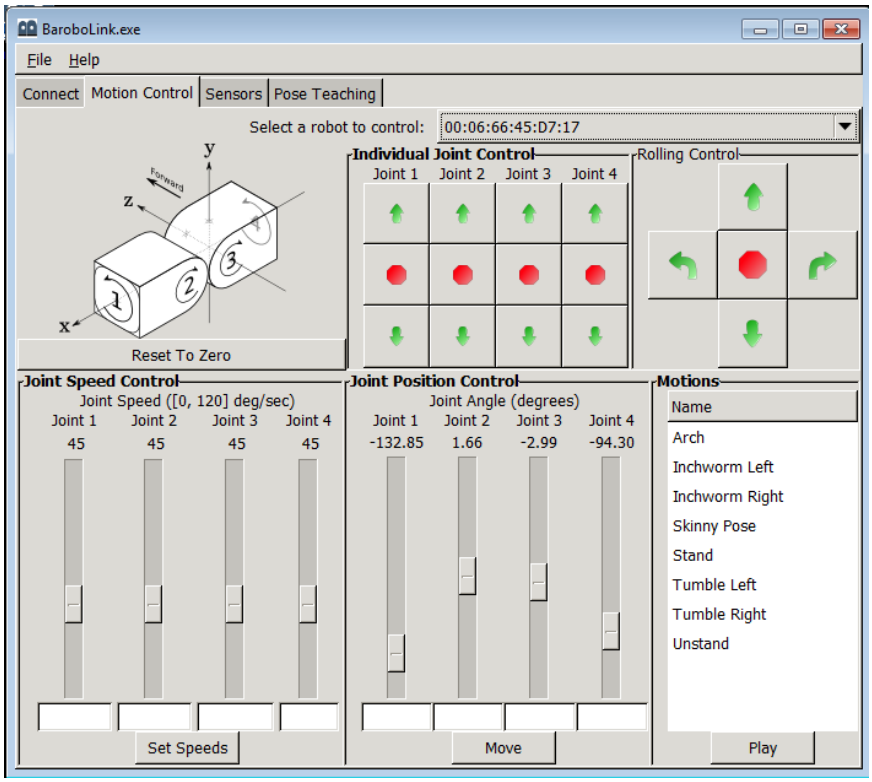


Figure 19: The graphical display of BaroboLink while connected to a Mobot.

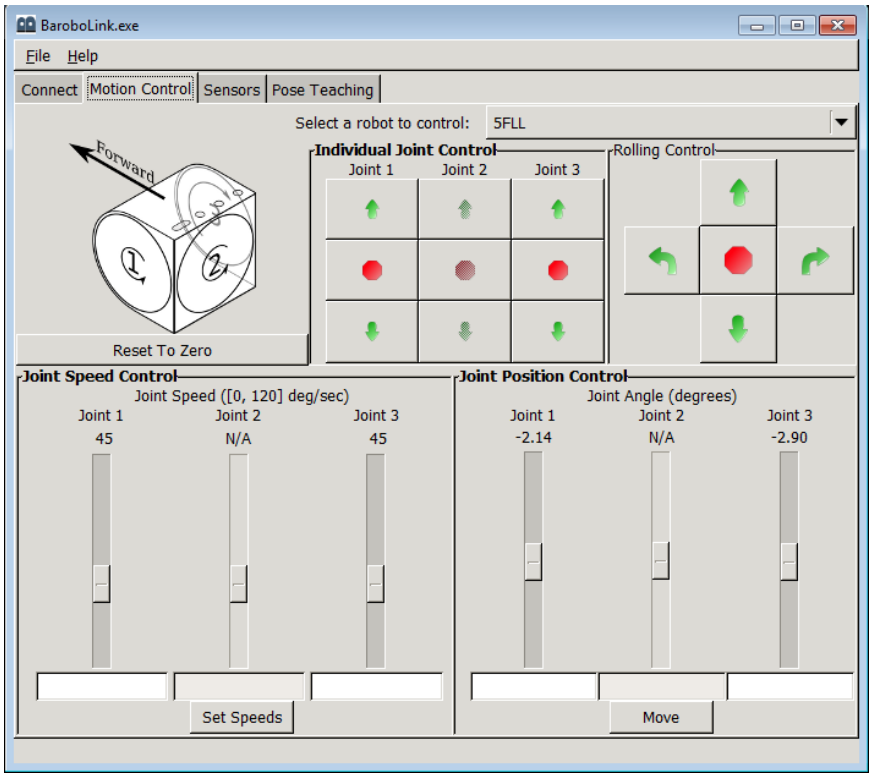


Figure 20: The graphical display of BaroboLink while connected to a Linkbot-I or -L.

Once a robot is connected to BaroboLink, the joint angles and speeds of the robot are displayed as shown in Figure 19. This dialog is located under the third tab of BaroboLink, labeled "Motion Control". The "Motion Control" tab can be used to display information about a Mobot's joint positions, and also control the speeds and positions of the Mobot's joints. The interface is divided up into six sections; three on the top half of the interface, and three on the bottom half.

5.1 The Robot Diagram and "Reset To Zero" Button

The first section of the GUI located on the top left of the interface displays a schematic diagram of the robot, displaying motor positions. Underneath the diagram, there is a large button with the text "Reset To Zero". When clicked, this button will command the connected Mobot to rotate all of its joints to a flat "Zero" position.

It is also possible to move the robot into zero position using the on-board buttons. Please refer to Section 6.3 on page 28 for more details.

5.2 Individual Joint Control

The second section, located at the top-middle section of the interface, is the "Individual Joint Control" section. These buttons command the Mobot to move individual joints. When the up or down arrows are clicked, the robot begins to move the corresponding joint in either the positive, or negative direction. The joint will continue to move until the stop button, located between the up and down arrows, is clicked.

If the joint encounters any obstacle that prevents it from moving, the joint will automatically disengage power to the joint. This may happen, example, if a body joint attempts to rotate beyond its limits, or if it collides with the other corresponding body joint.

5.3 Rolling Control (Mobot and Linkbot-I only)

This section contains buttons for controlling the Mobot as a two wheeled mobile robot. The up and down buttons cause the robot to roll forward or backward. The left and right buttons cause the robot to rotate towards the left, or towards the right. The stop button in the middle causes the robot to stop where it is. Note that this functionality does not exist for the Linkbot-L.

5.4 Joint Speeds

The "Joint Speeds" section, located at the bottom left of the interface, displays and controls the current joint speeds of the robot. The joint speeds are in units of degrees per second. To set a specific desired joint speed for a particular joint, the joint speed may be typed directly into the edit boxes below the sliders, and the "Set" button should be clicked.

5.5 Joint Positions

This section, located in the bottom-middle of the interface, is used to display and control the positions of each of the four joints of a robot. The joint positions are displayed in the numerical text located above each vertical slider. The displayed joint positions are in units of degrees.

The method of controlling the joints is by using the vertical sliders. Each vertical slider's position represents a joint's angle. The angles for the two body joints vary from -90 to 90 degrees. When the position of the slider is moved, the robot will move its joints to match the sliders.

Underneath the sliders, there are four text entry boxes. The text boxes accept specific angles for each joint which the user may type in. If the "Move" button is clicked, each joint will move to their respective by the amount specified in the text boxes. If any text entry is left blank, the corresponding joint will not move.

5.5.1 Joint Limits

Joints 1 and 4 are fully rotational and have no joint limits. Joints 2 and 3, however, are limited to a range of -90 to +90 degrees.

5.6 Motions (Mobot only)

This section, located on the bottom right of the interface, contains a set of preprogrammed motions for the Mobot. To execute a preprogrammed motion, simply click on the name of the motion you wish to execute, and then click the button labeled "Play". Note that this functionality only works for the Mobot. This section of the dialog will not appear if controlling a Linkbot-I or -L.

6 Using the On-Board Buttons

6.1 Running the On-Board Joint Test program

Each robot comes with a demo test program hard-coded into the on-board computer. The test program is designed to test the robot's joints range of motion and make sure everything is working properly. The test program will move the robot continuously until the robot is shut off.

To run the test program, simply hold the "A" button for 3 seconds. After 3 seconds have passed, the blue LED should blink 3 times quickly, and the demo motion will begin.

6.2 Recalibrating the robot's Zero Position

A user may recalibrate the robot's zero positions if necessary. To recalibrate the robot, perform the following steps:

- Position the robot into its zero position. If the Mobot is currently actuating any of its joints, or if any joints feel "stiff", you may power the robot off and on again, or you may press both the "A" and "B" buttons to relax the joints.
- While the robot is powered on, press and hold both the "A" and "B" buttons simultaneously. After three seconds, the LED should flash quickly 3 times. The Mobot has now been recalibrated and you may release the buttons.

6.3 Testing the robot's Zero Position

Buttons "A" and "B" pressed together may be used to toggle the robot between holding its zero position or relaxing all of its joints. If the Mobot is currently relaxed, the Mobot will move to zero position and hold its position. If the "A" and "B" buttons are pressed again, the Mobot will relax its motors, allowing the user to reposition the Mobot.

This function is typically used to check the robot's zero position to make sure it is properly calibrated. The button may also be used to relax a robot that is currently holding any of its joints at a position.

7 The "Pose Teaching" Panel

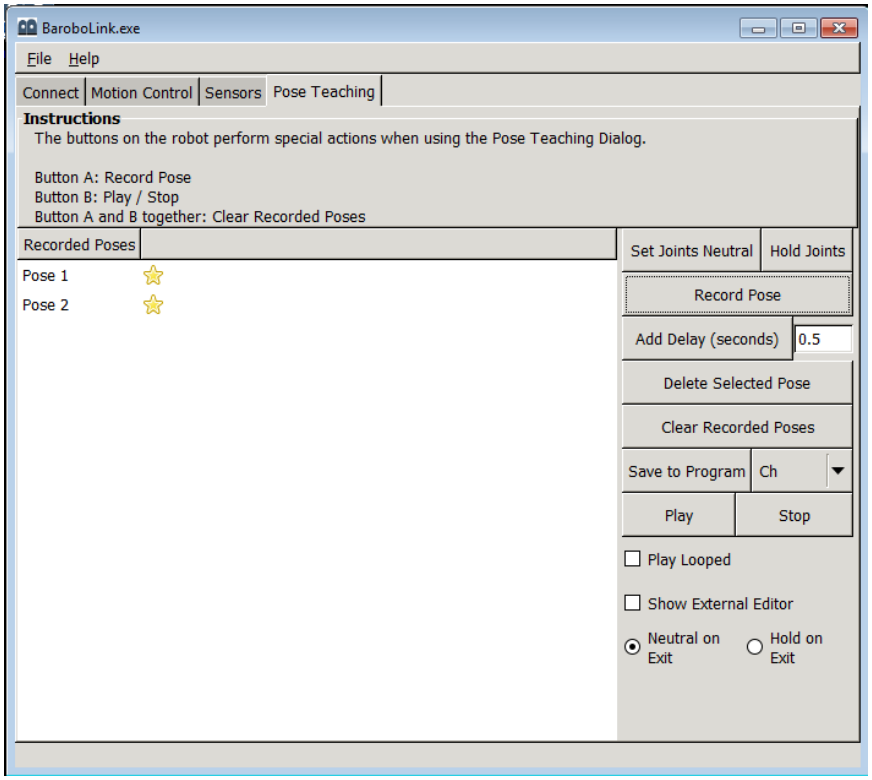


Figure 21: The pose training dialog.

"Pose Teaching" is the process of training a robot to perform motions by physically moving the robot into desired positions. To pose teach a robot or multiple robots, begin by connecting to all of the robots that will be trained by pose teaching in the "Connect" tab. Once all of the desired Mobots are connected, switch to the "Pose Teaching" tab. This will enable the pose training dialog, as shown in Figure 21. Once you have

switched tabs, the function of the Mobot's on-board buttons will change to Pose Teaching mode. In Pose Teaching mode, the Mobot buttons have the following functions:

- Button A: Record pose. This takes the position readings of all of the connected Mobots and saves them as a single recorded pose.
- Button B: Play the current recorded motions in order, or stop if motions are currently being played.
- Buttons A and B together: Clear all of the recorded poses.

Note that when the Mobots are being used in this mode, all of the Mobot buttons are connected together and do the same functions. In other words, if both Mobot1 and Mobot2 are connected, pressing the "A" button on Mobot1 is identical to pressing the "A" button on Mobot2; They will both result in a new recorded pose of both Mobots being created in BaroboLink.

The joints of the Mobots can be set to either "Neutral" mode or "Hold" mode by clicking on either the "Set Joints Neutral" button or the "Hold Joints" button, respectively.

Each time a pose is recorded, an indicator for that pose will appear in the dialog. As the poses are played, the pose icon changes for each pose to indicate the current pose being played.

Delays may be added between poses by using the "Add Delay" button. The length of the delay (in seconds) should be entered in the text box next to the "Add Delay" button.

It is also possible to save the recorded motion as a program by clicking on the "Save to Program" button. The generated program is a standard Ch Mobot program which may be executed in ChIDE.

8 The "Sensors" Panel

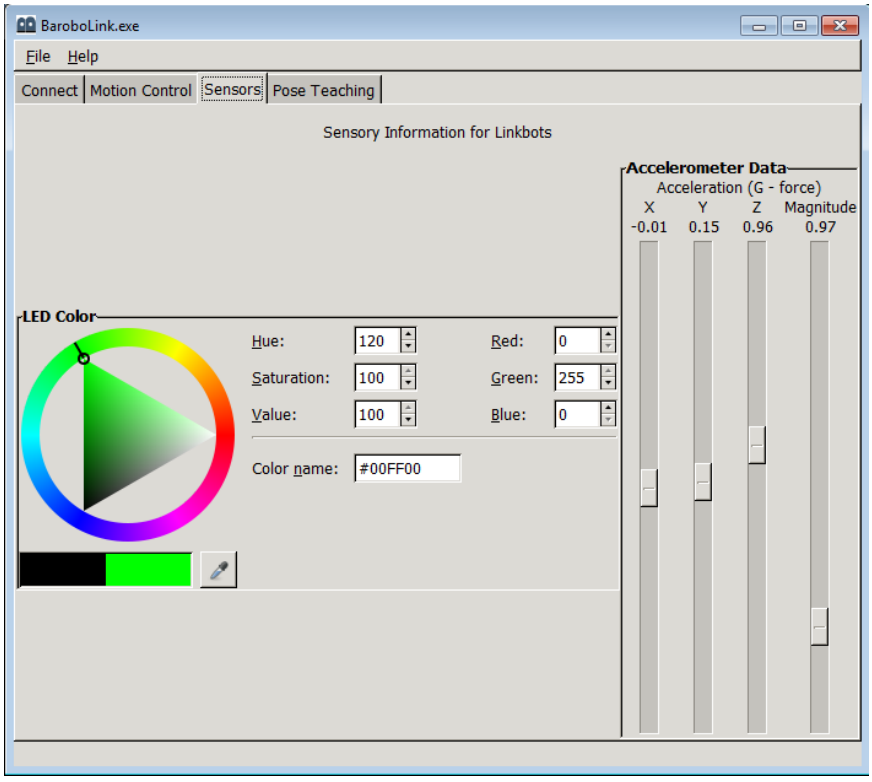


Figure 22: The sensors dialog.

The "Sensors" panel displays relevant information regarding any sensors the Mobot has access to. The Linkbot-I and -L robots have an internal accelerometer and LED color control, which are displayed in the dialog. The Mobot has no sensors so this dialog has no functionality with Mobots.

9 Getting Started with Programming the Mobot

BaroboLink can be used to control the Mobot for simple tasks and applications. For more complicated applications, computer programs are better suited for controlling the Mobot. Mobot can be controlled using a C/C++ program through Ch, a C/C++ interpreter. Ch Professional Edition, Ch Student Edition, or Ch Standard Edition is required to run the demo programs for controlling Mobot. Ch Student Edition is free for students. Ch Standard Edition is free for both academic and commercial uses. Ch is available from SoftIntegration, Inc. at `http://www.softintegration.com`

Before the Ch program is executed, the Bluetooth addresses of the robots need to be added using BaroboLink as described in Section 2.

To help the user become acquainted with the Mobot control programs, sample programs will be presented in this section to illustrate the basics and minimum requirements of a Mobot control program. The sample programs are located at `CHHOME/package/chbarobo/demos`, where `CHHOME` is the Ch home directory, such as `C:\Ch` for Windows. For Windows, it is located at `C:\Ch\package\chbarobo\demos` by default.

The first demo presents a minimal program which connects to a Mobot and moves joints 1 and 4. Although the programs are presented in Ch, the core concepts and member functions are applicable to C++ and other language bindings.

9.1 `start.ch`, A Basic Ch Mobot Program

9.1.1 `start.ch` Source Code

```
/* Filename: start.ch
 * Move the mobot faceplates. */
#include <mobot.h>

CMobot robot;

/* Connect to the paired Mobot */
robot.connect();

/* Set the robot to "home" position, where all joint angles are 0 degrees. */
printf("Reset to zero...\n");
robot.resetToZero();

/* Rotate each of the faceplates by 360 degrees */
printf("move...\n");
robot.move(360, 0, 360, 360);
printf("done.\n");
```

9.1.2 Demo Code for `start.ch` Explained

The beginning of every robot control program will include header files. Each header file imports functions used for a number of tasks, such as printing data onto the screen or controlling the Mobot. The `mobot.h` header file must be included in order to use the `CMobot` class and related robotic control functions.

```
#include <mobot.h> // Required for Mobot control functions
```

Next, we must initialize the C++ class used to control the Mobot.

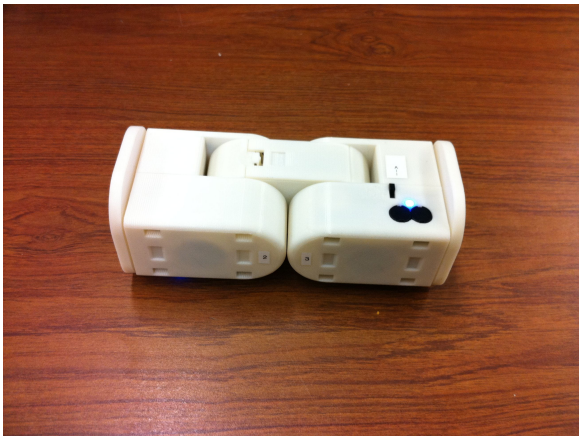


Figure 23: The mobot in zero position

```
CMobot robot;
```

This line initializes a new variable named `robot` which represents the remote Mobot module which we wish to control. This special variable is actually an instance of the `CMobot` class, which contains its own set of functions called "methods", "member functions", or simply "functions".

The next line,

```
robot.connect();
```

will connect our new variable, `robot`, to a Mobot that has been previously configured with the computer in the process described in Section 2.

Note that there are two common methods to connect to a remote Mobot. The most common method, demonstrated in the previous line of code, is used to connect to a Mobot that is already paired to the computer. It is also possible to connect to Mobots which are not paired with the computer. This method is necessary for connecting to multiple Mobots simultaneously, as only a single Mobot may be paired with the computer at a time. The second method uses the function `connectWithBluetoothAddress()`, and its default usage is as such:

```
string_t address = "11:22:33:44:55:66";
int defaultChannel = 1;
robot.connectWithBluetoothAddress(address, defaultChannel);
```

The string `"11:22:33:44:55:66"` represents the Bluetooth address of the Mobot, which must be known in advance. The channel number `1` represents the Bluetooth channel to connect to. Channel `1` is the default channel Mobots listen on for incoming connections, but may be set to other values depending on the type of robot. Detailed documentation for each of the Mobot functions, such as `connect()` and `connectAddress()`, are presented in Appendix A.4.

The next line,

```
robot.resetToZero();
```

uses the `resetToZero()` member function. The `resetToZero` function causes the Mobot to move all of its motors to the zero position, as shown in Figure 23.

The next line of code command joints 1 and 4 to rotate 360 degrees.

```
robot.move(360, 0, 0, 360);
```

Note that the member function `move()` expects input angles in degrees, so the angles in radians must be first be converted to degrees using the `radian2degree()` function. The `radian2degree()` function takes an angle in radians as its argument and returns the angle in degrees. The function is implemented in Ch with the code

```
#include <math.h> /* For M_PI */
double radian2degree(double radians)
{
    double degrees;
    degrees = radians * 180.0 / M_PI;
    return degrees;
}
```

If desired, values in radians may also be converted to degrees using the counterpart function, `degree2radian()`. Joints 1 and 4 are the faceplates of the Mobot which are sometimes used to act as "wheels".

9.2 `returnval.ch`, A Basic Ch Mobot Program Which Checks Return Values

9.2.1 Source Code

```
/* Filename: returnval.ch
 * Rotate the faceplates by 90 degrees */
#include <mobot.h>
CMobot robot;
double angle1, angle4;

/* Connect to the paired Mobot */
if(robot.connect())
{
    printf("Failed to connect to the robot.\n");
    exit(-1);
}
/* Set the robot to "home" position, where all joint angles are 0 degrees. */
robot.resetToZero();

/* Rotate each of the faceplates by 360 degrees */
angle1 = 360;
angle4 = 360;
robot.move(angle1, 0, 0, angle4);
/* Move the motors back to where they were */
angle1 = -360;
angle4 = -360;
robot.move(angle1, 0, 0, angle4);
```

9.2.2 `returnval.ch` Explained

The first portion of the code, the lines

```
#include <mobot.h>
CMobot robot;
```

set up our program for controlling mobots as seen in previous demos. The next line,

```
double angle1, angle4;
```

declares two variables that will be used to hold angle values later in the program.

The next lines, which connect to the robot, appear as such:

```
if(robot.connect())
{
    printf("Failed to connect to the robot.\n");
    exit(-1);
}
```

This section connects to the remote robot as in previous examples, but also does some error checking. The majority of the `CMobot` member functions return an integer value indicating whether or not the function succeeded. The `CMobot` member functions return 0 if they succeed, and -1 if any type of error has occurred. Errors may occur for any number of reasons, including lost connections, mechanical failure, and electrical interference. The demos up to this point have ignored the return values of the `CMobot` member functions.

Next following lines,

```
/* Set the robot to "home" position, where all joint angles are 0 degrees. */
robot.resetToZero();

/* Rotate each of the faceplates by 360 degrees */
angle1 = 360;
angle4 = 360;
robot.move(angle1, 0, 0, angle4);
/* Move the motors back to where they were */
angle1 = -360;
angle4 = -360;
robot.move(angle1, 0, 0, angle4);
```

move the robot into its zero position, rotates its end plates by one full rotation, and then rotates the end plates back to their original position. Unlike the previous demo, this demo uses variables to store the joint angle values. The variables are assigned values, and then used as the function arguments for the `move()` function.

9.3 `getJointAngle.ch`, A Basic Ch Mobot Program Which Retrieves a Joint Angle

9.3.1 Source Code

```
/* Filename: getJointAngle.ch
 * Find the current joint angle of a joint. */
#include <mobot.h>
CMobot robot;

/* Connect to a robot */
robot.connect();

/* Get the joint angle of the first joint */
double angle;
robot.getJointAngle(ROBOT_JOINT1, angle);

/* Print out the joint angle */
printf("The current joint angle for joint 1 is %lf degrees.\n", angle);
```

9.3.2 `getJointAngle.ch` Explained

The first portion of the program,

```
#include <mobot.h>
CMobot robot;

/* Connect to a robot */
robot.connect();
```

initialize the robot variable and connect to the remote robot, as shown in the previous demo. Next, the line

```
double angle;
```

initializes a new variable called `angle`, which will be used to store the current angle of one of the mobotic joints. The next line,

```
robot.getJointAngle(ROBOT_JOINT1, angle);
```

retrieves the current angle of joint 1, which is one of the faceplates of the robot. `ROBOT_JOINT1` is an enumerated value defined in the header file `mobot.h`. Detailed information for all enumerated values defined in `mobot.h` can be found in Appendix A.1.

Finally, the last line of the program,

```
printf("The current joint angle for joint 1 is %lf degrees.\n", angle);
```

prints the value of the variable onto the screen.

10 Controlling the Speed of Mobot Joints

10.1 `setspeed.ch` Source Code

```
/* Filename: setspeed.ch
  Move the two wheeled mobot with different speed. */
#include <mobot.h>
#include <math.h>
CMobot robot;

/* Connect to the paired Mobot */
robot.connect();

/* Set the robot to "home" position, where all joint angles are 0 degrees. */
robot.resetToZero();

double speed, radius;
robot.getJointMaxSpeed(ROBOT_JOINT1, speed);
printf("The maximum speed is %lf degrees/s\n", speed);

robot.setJointSpeed(ROBOT_JOINT1, 60);
robot.setJointSpeed(ROBOT_JOINT4, 60);

//robot.setJointSpeedRatio(ROBOT_JOINT1, 0.5);
//robot.setJointSpeedRatio(ROBOT_JOINT4, 0.5);

//robot.setJointSpeeds(60, 0, 0, 60);
```

```
printf("Roll forward 360 degrees.\n");
robot.motionRollForward(360);

speed = 1.83; // = (3.5/2) * M_PI * 60/180 (inch/s)
radius = 3.5/2;             // radius is 1.75
robot.setTwoWheelRobotSpeed(speed, radius);

printf("Move 360 degrees.\n");
robot.move(360, 0, 0, 360);

/* move at 1.83inch/sec with the radius 3.5 inches for 3 seconds */
printf("Move continuously for 3 seconds.\n");
robot.setMovementStateTime(ROBOT_FORWARD, ROBOT_HOLD, ROBOT_HOLD, ROBOT_FORWARD, 3);
```

10.2 `setspeed.ch` Source Code Explanation

The first several lines,

```
#include <mobot.h>
#include <math.h>

CMobot robot;

/* Connect to the paired Mobot */
robot.connect();

/* Set the robot to "home" position, where all joint angles are 0 degrees. */
robot.resetToZero();
```

initialize the program, connect to the robot, and move the robot into its zero position, similar to the previous demos. The next three lines,

```
double speed, radius;
robot.getJointMaxSpeed(ROBOT_JOINT1, speed);
printf("The maximum speed is %lf degrees/s\n", speed);
```

initializes the variables `speed` and `radius`, retrieves the maximum joint speed for the first joint using the member function, `getJointMaxSpeed` and stores it in the variables named `speed`. The value of the maximum speed is then printed onto the screen using the `printf()` function.

The next two lines,

```
robot.setJointSpeed(ROBOT_JOINT1, 60);
robot.setJointSpeed(ROBOT_JOINT4, 60);
```

set the joint speed settings for the two faceplate joints to 90 degrees per second.

The next lines,

```
//robot.setJointSpeedRatio(ROBOT_JOINT1, 0.5);
//robot.setJointSpeedRatio(ROBOT_JOINT4, 0.5);

//robot.setJointSpeeds(60, 0, 0, 60);
```

are two alternate ways of setting the joint speeds of the faceplate joints. The member function `setJointSpeedRatio()` sets the joint speeds as a ratio of the maximum speed. The function `setJointSpeeds()` is used to set

all four joint speeds simultaneously. Note though, that there is a slight difference between using the `setJointSpeeds()` function as shown in this example compared to the other methods. The other methods do not alter the joint speeds for joints 2 and 3, while the `setJointSpeeds()` function used as shown in the example explicitly sets the joint speeds of joints 2 and 3 to zero.

The next two lines,

```
printf("Roll forward 360 degrees.\n");
robot.moveForward(360);
```

print a message to the screen and rolls the robot forward by rotating the faceplates 360 degrees.

The next lines,

```
speed = 1.83; // = (3.5/2) * M_PI * 60/180 (inch/s)
radius = 3.5/2;             // radius is 1.75
robot.setTwoWheelRobotSpeed(speed, radius);
```

use the `setTwoWheelRobotSpeed()` function to set the faceplate joint speeds for a robot acting as a two wheeled car. The `setTwoWheelRobotSpeed()` function takes a desired speed and the radius of the wheels as arguments and calculates the necessary rotational speed of the faceplate wheels to achieve the desired speed. Note that the units for the speed must match the units for the radius. For instance, if the radius is provided in inches, the desired speed must be provided in inches per second. If the radius is provided in centimeters, the speed must be provided in centimeters per second, and so on. In this example, specifying the speed of the two wheeled robot at 1.83 inches per second is equivalent to setting the speeds of the two wheels at 60 degrees per second.

The following two lines,

```
printf("Move 360 degrees.\n");
robot.move(360, 0, 0, 360);
```

rotate the faceplates forward at the necessary rate to achieve a forward speed of 1.83 inches per second.

Finally, the last two lines,

```
printf("Move continuously for 3 seconds.\n");
robot.setMovementStateTime(ROBOT_FORWARD, ROBOT_HOLD, ROBOT_HOLD, ROBOT_FORWARD, 3);
```

roll the robot forward for three seconds. The enumerated values `ROBOT_FORWARD` and `ROBOT_BACKWARD` indicate the forward and backward directions for each motor to turn, respectively. `ROBOT_NEUTRAL` indicates that the motor should not turn, but should remain flexible and back-drivable. `ROBOT_HOLD` indicates that the joint will not turn, and that the joint will be forcefully held in place at its current position. More information regarding these macros may be found in Section A.3. In the previous line of code, joints 1 and 4 move forward while joints 2 and 3 hold their current positions. The last argument of the function `setMovementStateTime()` specifies the duration of time to move or hold the motors in seconds.

11 Preprogrammed Motions

The robot API contains functions for executing preprogrammed motions. The preprogrammed motions are motions which are commonly used for robot locomotion. Following is a list of available functions and a brief description about their effect on the robot.

- `motionArch()`: This function causes the robot to arch up for better clearance.
- `motionInchwormLeft()`: This function causes the robot to perform the inchworm gait once, moving the robot towards its left.
- `motionInchwormRight()`: This function causes the robot to perform the inchworm gait once, moving the robot towards its right.

- `motionSkinny()`: This function makes the robot assume a skinnier rolling profile.
- `motionStand()`: This function causes the robot to stand up onto a faceplate, assuming the camera platform position.
- `motionTumbleRight()`: This function causes the robot to perform the tumbling motion, flipping end over end.
- `motionTumbleLeft()`: This function causes the robot to perform the tumbling motion, flipping end over end.
- `motionUnstand()`: This function causes the robot to drop down from a standing position.

Note that all of the functions listed above are "blocking" functions, meaning they will not return until the motion has completed. These functions also have non-blocking equivalents which are discussed in Section 13.

11.1 `inchworm.ch`: A Demo using the `motionInchwormLeft()` Preprogrammed Motion

11.1.1 `inchworm.ch` Source Code

```
/* File: inchworm.ch
 * Perform the "inchworm" motion four times */
#include <mobot.h>
CMobot robot;

/* Connect to the paired Mobot */
robot.connect();

/* Set robot motors to speed of 0.50 */
robot.setJointSpeedRatio(ROBOT_JOINT2, 0.50);
robot.setJointSpeedRatio(ROBOT_JOINT3, 0.50);

/* Set the robot to "home" position, where all joint angles are 0 degrees. */
robot.resetToZero();

/* Do the inchworm motion four times */
robot.motionInchwormLeft(4);
```

11.1.2 `inchworm.ch` Explained

First, the header file `mobot.h` is included. This header file is required before usage of the `CMobot` class and its associated member functions can be used. Next, we create a variable to represent our robot and connect to the robot with the following lines.

```
CMobot robot;

/* Connect to the paired Mobot */
robot.connect();
```

Next, we set the motor speeds to 50% speed with the following lines.

```
robot.setJointSpeedRatio(ROBOT_JOINT2, 0.50);
robot.setJointSpeedRatio(ROBOT_JOINT3, 0.50);
```

We then move the robot to its zero position in preparation for the inchworm gait.

```
robot.resetToZero();
```

Finally, we perform the inchworm gait four times. The argument for the function `motionInchwormLeft()` represents the number of times the gait should be performed.

```
robot.motionInchwormLeft(4);
```

11.2 `stand.ch`: A Demo Using the `motionStand()` Preprogrammed Motion

This demo is a simple demonstration of the `motionStand()` member function.

11.2.1 `stand.ch` Source Code

```
/* Filename: stand.ch
 * Make a Mobot stand up on a faceplate */
#include <mobot.h>
CMobot robot;

/* Connect to the Mobot */
robot.connect();
/* Run the built-in motionStand function */
robot.motionStand();
delay(3); // Stand still for three seconds
/* Spin the robot around two revolutions while spinning the top faceplate*/
robot.move(2*360, 0, 0, 2*360);
/* Lay the robot back down */
robot.motionUnstand();
```

11.2.2 `stand.ch` Explained

After the initialization and connection as seen in the previous demo, it executes the following line of code:

```
robot.motionStand();
```

This line of code causes the Mobot to perform a sequence of motions causing it to stand up on a faceplate. After the robot has stood up, the next line of code,

```
delay(3); // Stand still for three seconds
```

pauses the program for three seconds, causing the robot to remain still for three seconds.

After the pause is over, the line

```
robot.move(2*360, 0, 0, 2*360);
```

turns both faceplates of the robot two full rotations, making the robot spin in place while standing. Finally, the line

```
robot.motionUnstand();
```

causes the robot to drop back down into a flat position.

11.3 `tumble.ch`: A Demo Using the `motionTumbleLeft()` Preprogrammed Motion

11.3.1 `tumble.ch` Source Code

```
/* Filename: tumble.ch
 * Tumbling mobot */
#include <mobot.h>
CMobot robot;

/* Connect to the paired Mobot */
robot.connect();
/* Set the robot to "home" position, where all joint angles are 0 degrees. */
robot.resetToZero();

/* Tumble two times */
robot.motionTumbleLeft(2);
```

11.3.2 `tumble.ch` Explained

The first portion of the program,

```
/* Filename: tumble.ch
 * Tumbling robot */
#include <mobot.h>
CMobot robot;

/* Connect to the paired Mobot */
robot.connect()
/* Set the robot to "home" position, where all joint angles are 0 degrees. */
robot.resetToZero();
```

initialize the proper variables, connect to the remote robot, and make it move to a flat zero position, similar to previous demos.

Next, we make the robot perform the tumbling motion with the following line:

```
robot.motionTumbleLeft(2);
```

The argument, "2", indicates that the tumbling motion should be performed two times.

11.4 `motion.ch`: A Demo Using Multiple Preprogrammed Motions

11.4.1 `motion.ch` Source Code

```
/* Filename: motion.ch
 * Move the two wheeled mobot. */
#include <mobot.h>
CMobot robot;

/* Connect to the paired Mobot */
robot.connect();

/* Set the robot to "home" position, where all joint angles are 0 degrees. */
robot.resetToZero();

robot.moveTo(90, 0, 0, 90);
```

```
robot.moveTo(180, 0, 0, 180);

robot.moveTo(0, 0, 0, 0);

/* test all pre-programmed motions */
robot.motionArch(90);
robot.motionInchwormLeft(4);
robot.motionInchwormRight(4);
robot.moveBackward(360);
robot.moveForward(360);
robot.turnLeft(360);
robot.turnRight(360);
robot.motionStand();
robot.move(360, 0, 0, 360);
robot.motionUnstand();
robot.motionTumbleLeft(2);
```

11.4.2 `motion.ch` Explained

The first portion of the program initializes and connects to the remote robot similar to the previous demos.

The next portion of code executes a number of different programmed motions to demonstrate the motion capabilities of the robot. First, the "Arch" motion is demonstrated by the following line of code:

```
robot.motionArch(15);
```

The parameter given to the function, "15" in this case, is the angle in degrees that the body joints should form in relation to each other.

The next couple lines of code,

```
robot.motionInchwormLeft(4);
robot.motionInchwormRight(4);
```

make the robot inchworm to the left four times, and then inchworm to the right four times.

Next, the lines

```
robot.moveBackward(360);
robot.moveForward(360);
robot.turnLeft(360);
robot.turnRight(360);
```

make the robot roll backward, forward, and then turn left, and turn right sequentially. For each of these functions, the parameter is the angle in degrees to turn the faceplates. For instance, the line rolls the robot backward using its faceplates as wheels by rotating the faceplates 360 degrees.

Next, the robot stands up by executing the line

```
robot.motionStand();
```

While the robot is standing, the line

```
robot.move(360, 0, 0, 360);
```

rotates both faceplates one complete rotation. This causes the robot to spin around in a circle, since it is currently standing on one of its faceplates.

Next, we lay the robot back down into a prone position with the following line of code:

```
robot.motionUnstand();
```

Finally, we perform the tumbling motion.

```
robot.motionTumbleLeft(2);
```

The tumbling motion is a movement in which the robot stands up and then flips, end over end. The argument provided to the function, "2" in this case, is the number of times to perform the motion.

12 Detailed Examples of Preprogrammed Motions and Writing Customized Motions

In the previous sections, preprogrammed motions have been demonstrated. In this section, the inner workings of the preprogrammed motions will be discussed, as well as various methods of designing and creating custom motions. Some more complex motions, such as "motionTumbleLeft()" and "motionUnstand()" will be discussed in Section 13.

12.1 Inchworm Gait Demo

The next demo will illustrate how a simple gait known as the "Inchworm" gait can be implemented.

12.1.1 `inchworm2.ch` Source Code

```
/* File: inchworm2.ch
 * Perform the "inchworm" motion four times */
#include <mobot.h>
CMobot robot;

/* Connect to the paired Mobot */
robot.connect();

/* Set robot motors to speed of 0.50 */
robot.setJointSpeedRatio(ROBOT_JOINT2, 0.50);
robot.setJointSpeedRatio(ROBOT_JOINT3, 0.50);

/* Set the robot to "home" position, where all joint angles are 0 degrees. */
robot.resetToZero();

/* Do the inchworm motion four times */
int i, num = 4;
double angle2 = -45;
double angle3 = 45;
for(i = 0; i < num; i++) {
    robot.moveJointTo(ROBOT_JOINT2, angle2); /* Move joint 2 */
    robot.moveJointTo(ROBOT_JOINT3, angle3); /* Move joint 3 */
    robot.moveJointTo(ROBOT_JOINT2, 0);      /* Move joint 2 */
    robot.moveJointTo(ROBOT_JOINT3, 0);      /* Move joint 3 back to zero position */
}
```

12.1.2 Demo Code for `inchworm2.ch` Explained

The first portion of the code is identical to the previous demo, and performs the same function of declaring a Mobot variable and connecting to a paired Mobot.

(a) Zero Position (b) Move joint 2 (c) Move joint 3 (d) Move joint 2 (e) Move joint 3 back to zero position

Figure 24: The inchworm motion

```
#include <mobot.h>
CMobot robot;

/* Connect to the paired Mobot */
robot.connect();
```

The next lines of code set the joint speeds for the two body joints, joints 2 and 3, to 50% speed. They are set to fifty percent speed in order to slow the motion down in order to minimize slippage.

```
/* Set robot motors to speed of 0.50 */
robot.setJointSpeedRatio(ROBOT_JOINT2, 0.50);
robot.setJointSpeedRatio(ROBOT_JOINT3, 0.50);
```

Next, we move the robot into a flat "zero" position, as shown in Figure 24a.

```
/* Set the robot to "home" position, where all joint angles are 0 degrees. */
robot.resetToZero();
```

Finally, we perform the actual inchworm motion. The inchworm motion is a gait defined by a sequence of motions performed by the body joints. The motions are as such:

1. The first body joint, referred to as joint A, rotates towards the ground. This drags the Mobot towards the direction of joint A. (Figure 24b)
2. The other body joint, joint B, rotates towards the ground. Since the center of gravity is currently positioned over joint A, this causes the trailing body joint to slide toward joint A. (Figure 24c)
3. Joint A moves back to a flat position. (Figure 24d)
4. Joint B moves back to a flat position. (Figure 24e)
5. Repeat, if desired.

The direction of travel depends on the selection of the initial body joint. In the following code example, joint 2 is chosen as the initial body joint to move. In this case, the Mobot will traverse towards joint 2. The entire motion is encapsulated in a "for" loop which executes the entire motion four times.

```
/* Do the inchworm gait four times */
int i, num = 4;
double angle2 = -45;
double angle3 = 45;
for(i = 0; i < num; i++) {
    robot.moveJointTo(ROBOT_JOINT2, angle2); /* Move joint 2 */
    robot.moveJointTo(ROBOT_JOINT3, angle3); /* Move joint 3 */
    robot.moveJointTo(ROBOT_JOINT2, 0);      /* Move joint 2 */
    robot.moveJointTo(ROBOT_JOINT3, 0);      /* Move joint 3 back to zero position*/
}
```

The values of the variables `angle2` and `angle3` may also be modified to produce different variations of the inchworm gait to accommodate different terrain textures and ground surfaces.

12.2 Standing Demo

12.2.1 `stand2.ch` Source Code

```
/* Filename: stand2.ch
 * Make a Mobot stand up on a faceplate */
#include <mobot.h>
CMobot robot;

/* Connect to the paired Mobot */
robot.connect();

/* Set robot motors to speed of 90 degrees per second */
robot.setJointSpeed(ROBOT_JOINT2, 90);
robot.setJointSpeed(ROBOT_JOINT3, 90);
/* Set the robot to "home" position, where all joint angles are 0 degrees. */
robot.resetToZero();

/* Move the robot into a fetal position */
robot.moveJointTo(ROBOT_JOINT2, -85);
robot.moveJointTo(ROBOT_JOINT3, 70);

/* Wait a second for the robot to settle down */
delay(1);

/* Rotate the bottom faceplate by 45 degrees */
robot.moveJointTo(ROBOT_JOINT1, 45);

/* Lift the body up */
robot.moveJointTo(ROBOT_JOINT2, 20);

/* Pan the robot around for 3 seconds at 45 degrees per second*/
robot.setJointSpeed(ROBOT_JOINT1, 45);
robot.setMovementStateTime(ROBOT_FORWARD, ROBOT_HOLD, ROBOT_HOLD, ROBOT_HOLD, 3);
```

12.2.2 `stand2.ch` Explained

The first portion of the program performs the necessary setup and connecting, similar to the previous demos. Similar to the previous inchworm demo, the motor speeds are set to a speed of 90 degrees per second, and the function `resetToZero()` is called to put the robot into a flat position. Next, the following lines are executed:

```
robot.moveJointTo(ROBOT_JOINT2, -85);
robot.moveJointTo(ROBOT_JOINT3, 70);
```

These movement commands cause the Mobot to curl up into a fetal position with both of its faceplates facing toward the ground. The next line,

```
delay(1);
```

causes the program to pause for one second before continuing. This allows the robot to settle down, in case it was still in motion from the last movement.

Next, the Mobot rotates one of the faceplates by 45 degrees.

```
robot.moveJointTo(ROBOT_JOINT1, 45);
```

This endplate will eventually become the "foot" of the standing Mobot. Next, the Mobot lifts itself into a standing position, balancing on its endplate.

```
robot.moveJointTo(ROBOT_JOINT2, 20);
```

Note that the previous joint angle for Joint 2, a body joint, was -85 degrees. This motion causes joint 2 to rotate all the way to a 20 degree position, which lift up the body of the Mobot such that the Mobot is balancing on faceplate joint 1.

Finally, we rotate joint 1, the foot joint, for three seconds which causes the entire Mobot to rotate in place. The speed is first set to 45 degrees per second to make the rotation a slow rotation. Next, the `setMovementStateTime` member function is used to continuously rotate a joint for a desired amount of time.

```
robot.setJointSpeed(ROBOT_JOINT1, 45);
robot.setMovementStateTime(ROBOT_FORWARD, ROBOT_HOLD, ROBOT_HOLD, ROBOT_HOLD, 3);
```

13 Blocking and Non-Blocking Functions

All of the Mobot movement functions may be designated as either "blocking" functions or "non-blocking" functions. A blocking function is a function which does not return while operations are being performed. All standard C functions, such as `printf()`, are blocking functions. The `moveWait()` function is a blocking function. When called, the function will hang, or "block", until all the joints have stopped moving. After all joints have stopped moving, the `moveWait()` function will return, and the rest of the program will execute.

Furthermore, some functions have both a blocking version and a non-blocking version. For these functions, the suffix `NB` denotes that the function is non-blocking. For instance, the function `motionStand()` is blocking, meaning the function will not return until the motion is completed, whereas the function `motionStandNB()` is non-blocking, meaning the function returns immediately and the robot performs the "standing" motion asynchronously.

The function `moveNB()` is an example of a non-blocking function. When the `moveNB()` function is called, the function immediately returns as the joints begin moving. Any lines of code following the call to `moveNB()` will be executed even if the current motion is still in progress.

Demos for the non-blocking functions are located in the next section of this document.

13.1 List of Blocking Movement Functions

- `driveJointTo()`
- `driveTo()`
- `move()`
- `moveBackward()`
- `moveDistance()`
- `moveForward()`
- `moveJoint()`
- `moveJointTo()`
- `moveJointWait()`

- moveTo()
- moveToZero()
- moveWait()
- motionArch()
- motionInchwormLeft()
- motionInchwormRight()
- motionSkinny()
- motionStand()
- motionTumbleLeft()
- motionTumbleRight()
- motionUnstand()
- resetToZero()
- setJointMovementStateTime()
- setMovementStateTime()
- turnLeft()
- turnRight()

13.2 List of Non-Blocking Movement Functions

- driveJointToNB()
- driveToNB()
- moveNB()
- moveBackwardNB()
- moveDistanceNB()
- moveForwardNB()
- moveJointNB()
- moveJointToNB()
- moveToNB()
- moveToZeroNB()
- motionArchNB()
- motionInchwormLeftNB()
- motionInchwormRightNB()
- motionSkinnyNB()
- motionStandNB()

- motionTumbleLeftNB()
- motionTumbleRightNB()
- motionUnstandNB()
- resetToZeroNB()
- setJointMovementStateNB()
- setJointMovementStateTimeNB()
- setMovementStateNB()
- setMovementStateTimeNB()
- turnLeftNB()
- turnRightNB()

13.3 Blocking and Non-Blocking Demo Programs

13.3.1 nonblock.ch Source Code

```
/* File: nonblock.ch
   use the non-blocking functoin moveNB() . */
#include <mobot.h>
CMobot robot;

/* Connect to the paired Mobot */
robot.connect();

robot.resetToZero();

/* Rotate each of the faceplates by 720 degrees */
//robot.move(720, 0, 0, 720); // Blocking version
robot.moveNB(720, 0, 0, 720); // Non-Blocking version
while(robot.isMoving()) {
    printf("robot is moving ...\n");
}
printf("move finished!\n");
```

13.3.2 nonblock.ch Source Code Explanation

This demo gives an example of how non blocking functions operate. After the initial setup and initialization similar to the previous demos, the following line is executed:

```
mobot.moveNB(720, 0, 0, 720); // Non-Blocking version
```

The function `moveNB()` is a non-blocking function, which means that the program will continue executing even before the movement has completed. The next lines of code appear as such:

```
while(mobot.isMoving()) {
    printf("mobot is moving ...\n");
}
```

The previous lines of code basically loops as long as the `mobot.isMoving()` function is returning true. In other words, in plain English, as long as the mobot is moving, the program will print the message "mobot is moving...". As soon as the mobot completes its motion, the loop will break and the message "move finished!" is printed.

Alternatively, there is a commented line of code that appears in the program, which appears as

```
mobot.move(720, 0, 0, 720); // Non-Blocking version
```

If the `moveNB()` function is replaced with the `move()` function in this program, the message "mobot is moving..." is never printed to the screen. This is due to the fact that `move()` is a blocking function, and by the time the program continues past the `move()` statement, the mobot has already stopped moving.

13.3.3 `nonblock2.ch` Source Code

```
/* File: nonblock2.ch
   use the non-blocking functoin moveNB() . */
#include <mobot.h>
CMobot robot;

/* Connect to the paired Mobot */
robot.connect();

robot.resetToZero();

/* Rotate each of the faceplates by 360 degrees */
//robot.moveJoint(ROBOT_JOINT1, 360); // Blocking version
robot.moveJointNB(ROBOT_JOINT1, 360); // Non-Blocking version
robot.moveJoint(ROBOT_JOINT4, 360);
```

13.3.4 `nonblock2.ch` Source Code Explanation

The first block of the source code initialized and sets up the remote mobot similar to previous demos. The last two lines in the program appear like so:

```
mobot.moveJointNB(ROBOT_JOINT1, 360); // Non-Blocking version
mobot.moveJoint(ROBOT_JOINT4, 360);
```

The first line turns joint 1 for 360 degrees. However, since it is moved with a non-blocking function, the program immediately continues to the next line. The second line turns joint 4 for 360 degrees. Because computer programs execute so fast compared to the physical motion of the mobots, this program effectively begins rotating joints 1 and 4 simultaneously. Since the function `moveJoint()` is a blocking function, the program will not execute beyond that point until joint 4 has finished moving.

13.3.5 `nonblock3.ch` Source Code

```
/* File: nonblock3.ch
   Roll and arch simultaneously. */
#include <mobot.h>
CMobot robot;

/* Connect to the paired Mobot */
robot.connect();

robot.resetToZero();
```

```
printf("Rolling 360 degrees.\n");
robot.moveForward(360);
printf("Rolling 360 degrees while arching.\n");
robot.motionArchNB(15);
robot.moveForwardNB(360);
robot.moveWait();
```

13.3.6 `nonblock3.ch` Source Code Explanation

The first section of code initializes the necessary variables to control remote mobots as seen in previous demos. Next, we print a message on the screen and roll the mobot forward with the following two lines of code.

```
printf("Rolling 360 degrees.\n");
mobot.moveForward(360);
```

Next, we make two calls to non-blocking functions.

```
mobot.motionArchNB(15);
mobot.moveForwardNB(360);
```

The first call is to the function `motionArchNB()`, which arches the mobot up by moving joints 2 and 3. Since it is a non-blocking function, the program immediately continues on even before the arching motion as finished. The next call rolls the mobot forward by rotating joints 1 and 4. In effect, these two lines cause the mobot to roll forward and arch simultaneously. It is important to note that this compound motion works because the Arch motion only moves joints 2 and 3 while the rolling motions only move joints 1 and 4, so there are no conflicting motor commands.

In order to wait for all motions to finish, the last line of the program is

```
mobot.motionWait();
```

The `motionWait()` function will wait until all mobot motions are finished.

Note that if a program contains non-blocking functions, it is typically necessary to call a waiting function such as `moveWait()` or `motionWait()` before the program terminates. If a waiting function is not called, the program may terminate before the motion has been completed, which may halt the mobot in the middle of one of its motions.

13.4 Preprogrammed Motion Demos with Non-Blocking Functions

13.4.1 `unstand2.ch` Source Code

```
/* Filename: unstand.ch
 * Drop the mobot down while it is standing on faceplate 1. */
#include <mobot.h>
CMobot robot;

/* Connect to the paired Mobot */
robot.connect();
robot.resetToZero();

robot.moveJointToNB(ROBOT_JOINT2, -85);
robot.moveJointToNB(ROBOT_JOINT3, 45);
robot.moveWait();
robot.resetToZero();
```

13.4.2 `unstand2.ch` Source Code Explanation

The first block of code,

```
/* Filename: unstand.ch
 * Drop the mobot down from a standing position. */
#include <mobot.h>
CMobot mobot;

/* Connect to the paired Mobot */
mobot.connect();
```

initialize the program and connect to the remote mobot. Next, a series of non-blocking movements are performed:

```
mobot.moveJointToNB(ROBOT_JOINT2, -85);
mobot.moveJointToNB(ROBOT_JOINT3, 45);
```

Because both of these function calls are non-blocking, this function will effectively move joints 2 and 3 simultaneously. Since these are both non-blocking functions, a call to `moveWait()` is necessary to wait for the mobot to complete its motions, as done in the next line:

```
mobot.moveWait();
```

Finally, we move the mobot into a flat zero position with the following line:

```
mobot.resetToZero();
```

13.4.3 `tumble2.ch` Source Code

```
/* Filename: tumble2.ch
 * Tumbling mobot */
#include <mobot.h>
CMobot robot;

/* Connect to the paired Mobot */
robot.connect();

/* Set the robot to "home" position, where all joint angles are 0 degrees. */
robot.resetToZero();

/* Begin tumbling for "num" times */
int i, num = 2;
for(i = 0; i < num; i++) {
    /* First lift and tumble */
    robot.moveJointTo(ROBOT_JOINT2, -85);
    robot.moveJointTo(ROBOT_JOINT3, 80);
    robot.moveJointTo(ROBOT_JOINT2, 0);
    robot.moveJointTo(ROBOT_JOINT3, 0);
    robot.moveJointTo(ROBOT_JOINT2, 80);
    robot.moveJointTo(ROBOT_JOINT2, 45);
    /* Second lift and tumble */
    robot.moveJointTo(ROBOT_JOINT3, -85);
    robot.moveJointTo(ROBOT_JOINT2, 80);
    robot.moveJointTo(ROBOT_JOINT3, 0);
    robot.moveJointTo(ROBOT_JOINT2, 0);
```

```
    robot.moveJointTo(ROBOT_JOINT3, 80);
    if(i != (num-1)) { /* Do not perform this motion on the last tumble */
        robot.moveJointTo(ROBOT_JOINT3, 45);
    }
}

/* Unstand the robot */
robot.moveJointToNB(ROBOT_JOINT2, 0);
robot.moveJointToNB(ROBOT_JOINT3, 0);
robot.moveWait();
robot.resetToZero();
```

13.4.4 `tumble2.ch` Source Code Explanation

The first lines of the program,

```
#include <mobot.h>
CMobot mobot;

/* Connect to the paired Mobot */
mobot.connect();

/* Set the mobot to "home" position, where all joint angles are 0 degrees. */
mobot.resetToZero();
```

initialize the proper variables and connections as seen in previous demos.

Next, we begin the tumbling motion. We consider each tumbling motion to be the mobot flipping over twice. The reason the mobot flips twice per tumble is so that when the tumbling motion is done, the mobot ends right side up. A for loop is used to to tumble "num" times, as shown in the following code:

```
int i, num = 2;
for(i = 0; i < num; i++) {
```

These lines create two new variables, "i" and "num", which are the loop counter, and the number of times to tumble, respectively.

Inside the loop, two tumbling motions are performed. The first tumbling motion appears as such:

```
    /* First lift and tumble */
    mobot.moveJointTo(ROBOT_JOINT2, -85);
    mobot.moveJointTo(ROBOT_JOINT3, 80);
    mobot.moveJointTo(ROBOT_JOINT2, 0);
    mobot.moveJointTo(ROBOT_JOINT3, 0);
    mobot.moveJointTo(ROBOT_JOINT2, 80);
    mobot.moveJointTo(ROBOT_JOINT2, 45);
```

This movement is similar to the `motionStand()` motion, except that the mobot flips all the way over after standing up. After this motion is done, the mobot is balancing on joint 4. Next, we flip again so that the mobot is balancing on joint 1.

```
    /* Second lift and tumble */
    mobot.moveJointTo(ROBOT_JOINT3, -85);
    mobot.moveJointTo(ROBOT_JOINT2, 80);
    mobot.moveJointTo(ROBOT_JOINT3, 0);
    mobot.moveJointTo(ROBOT_JOINT2, 0);
    mobot.moveJointTo(ROBOT_JOINT3, 80);
```

```
    if(i != (num-1)) { /* Do not perform this motion on the last tumble */
        mobot.moveJointTo(ROBOT_JOINT3, 45);
    }
}
```

The `if` statement prevents the last motion from being executed when the mobot is on its last tumble. This is because the last motion in the loop is a preparatory motion for the next tumble, which is not needed if the mobot is currently performing its last tumble.

The entire motion, consisting of two flips, are performed "num" times. After the loop is completed, the mobot is made to fall back down into a prone positions with the following lines of code:

```
/* Unstand the mobot */
mobot.moveJointToNB(ROBOT_JOINT2, 0);
mobot.moveJointToNB(ROBOT_JOINT3, 0);
mobot.moveWait();
mobot.resetToZero();
```

14 Controlling Multiple Modules

The Mobot control software is designed to be able to control multiple modules simultaneously. There are some important differences in the program which enable the control of multiple modules. A small demo program which controls two modules simultaneously will first be presented, followed by a detailed explanation of the program elements.

14.1 `twoModules.ch` Source Code

```
/* Filename: twoModules.ch
 * Control two modules and make them stand and inchworm simultaneously. */
#include <mobot.h>
CMobot robot1;
CMobot robot2;

/* Connect robot variables to the robot modules. */
robot1.connect();
robot2.connect();

/* Set the robot to "home" position, where all joint angles are 0 degrees. */
robot1.resetToZeroNB();
robot2.resetToZeroNB();

robot1.moveWait();
robot2.moveWait();

/* Instruct the first robot to stand and the second robot to inchworm left four
 * times simultaneously. As soon as the first robot stands up, rotate its joints
 * 1 and 4 360 degrees. */
robot1.motionStandNB();
robot2.motionInchwormLeftNB(4);
robot1.motionWait();
robot1.moveNB(360, 0, 0, 360);
robot1.moveWait();
robot2.motionWait();
```

```
/* Instruct the first robot unstand and the second robot inchworm right four
 * times simultaneously. */
robot1.motionUnstandNB();
robot2.motionInchwormRightNB(4);
robot1.motionWait();
robot1.motionTumbleLeft(1);
robot2.motionWait();
```

14.2 Demo Explanation

The first two lines of interest appear as such:

```
CMobot mobot1;
CMobot mobot2;
```

These two lines declare two separate variables which will represent the two separate Mobot modules. Next, we need to connect each variable to a physically separate Mobot. This is done with the following lines.

```
mobot1.connect();
mobot2.connect();
```

These two lines connect the mobots to the first two addresses of the known mobot addresses. The list of the computer's known mobot addresses may be configured in the process detailed in Section 2 on page 13. For each separate control program, the first call to the `connect()` member function will connect to the first mobot listed in the configuration file. Each successive call to the `connect()` function will connect to successive mobots listed in the configuration file. The order in which they are connected may be modified using the "Configure Mobot Bluetooth" dialog, as discussed in Section 2.

```
mobot1.resetToZeroNB();
mobot2.resetToZeroNB();
```

These two lines command the two mobots to move to their zero positions. Note that these functions are non-blocking. This means that the `resetToZeroNB()` function will return immediately, and will not wait for the first mobot to finish completing the motion before commanding the second mobot to begin. In a normal program, this effectively causes both mobots to move to their zero positions simultaneously.

```
mobot1.moveWait();
mobot2.moveWait();
```

Since the `resetToZeroNB()` functions are non-blocking, we would like the program to wait until the motions are complete before continuing. By calling `moveWait()` on both of the mobots, we can be assured that the mobots have finished moving before the program continues.

```
mobot1.motionStandNB();
mobot2.motionInchwormLeftNB(4);
```

Similar to the calls to `resetToZeroNB()`, this block of code instructs the first Mobot to stand and the second Mobot to perform the inchworm motion four times. Note that we call the non-blocking versions of the functions, `motionStandNB()` and `motionInchwormLeftNB()`. Since these functions are non-blocking, both mobots will effectively perform the motions simultaneously.

When the first mobot finishes standing, we want it to spin around on its faceplate while the second mobot is still inchworming. We can accomplish this by first waiting for the standing motion to finish, and then moving the faceplate joints of the first mobot, as in the following lines of code.

```
  mobot1.motionWait();
  mobot1.moveNB(360, 0, 0, 360);
```

Before we continue with the program, we wish to ensure that motion has stopped on both mobots. Since the the last command sent to `mobot1` was a `moveNB()` command, we use the `moveWait()` function to wait for that movement to finish. Similarly, the last command sent to `mobot2` was a motion command, and so we use `motionWait()` to wait for the motion to finish. The two lines of code are seen in the program as follows:

```
  mobot1.moveWait();
  mobot2.motionWait();
```

Finally, the following lines,

```
mobot1.motionUnstandNB();
mobot2.motionInchwormRightNB(4);
mobot1.motionWait();
mobot1.motionTumbleLeft(1);
mobot2.motionWait();
```

make the first mobot come back down from its standing position and the second mobot inchworm to the right four times simultaneously. Note that the function `motionWait()` is used to wait for motions, which are compound movements, to finish. This is similar to how the function `moveWait()` is used to wait for individual movements to finish. After `mobot1` finishes unstanding, it will begin a tumbling motion. Since the tumbling motion function is called before the call to `mobot2.motionWait()`, `mobot1` will begin tumbling even if `mobot2` has finished its previous motion, which was inchworming 4 times.

14.3 Controlling Multiple Connected Modules

14.3.1 `lift.ch`, Lifting Demo

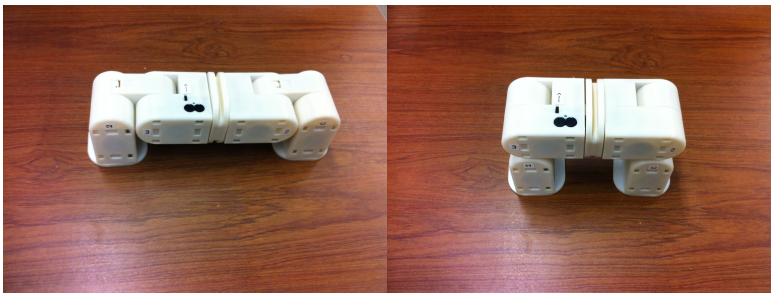
```
/* Filename: lift.ch
   Lift two connected mobots.
   Joint 4 of the 1st mobot should be connected to
   Joint 1 of the 2nd mobot as
           1st                        2nd
  |---------|--------|      |---------|--------|
 1|   2     |   3    | 4 X 1|    2    |   3    | 4
  |---------|--------|      |---------|--------|
*/
#include <mobot.h>
CMobot robot1;
CMobot robot2;

/* Connect robot variables to the robot modules. */
robot1.connect();
robot2.connect();

/* Set the robot to "home" position, where all joint angles are 0 degrees. */
robot1.resetToZeroNB();
robot2.resetToZeroNB();

robot1.moveWait();
robot2.moveWait();

/* First lift */
```



(a) First lift (b) Second lift

Figure 25: Lifting motion with two connected modules

```
robot1.moveToNB(0, -90,  0, 0);
robot2.moveToNB(0, 0, 90, 0);
robot1.moveWait();
robot2.moveWait();
delay(1);

/* Second lift */
robot1.moveToNB(0, 0, 90,  0);
robot2.moveToNB(0,  -90, 0, 0);
robot1.moveWait();
robot2.moveWait();
delay(1);

/* Move to zero position */
robot1.resetToZeroNB();
robot2.resetToZeroNB();
robot1.moveWait();
robot2.moveWait();
```

14.3.2 `lift.ch` Source Code Explanation

This demo is designed to lift two connected modules up into a two-legged standing configuration. The mobots are connected such that the fourth joint of mobot one is connected to the first joint of mobot two. The two connect mobots act as one long mobot.

The first portion of the program initialize two variables called `mobot1` and `mobot2`, which will be used to control the two connected modules. Once the mobots are initialized, they are both moved into a flat zero position.

The next lines,

```
/* First lift */
mobot1.moveNB(0, -90,  0, 0);
mobot2.moveNB(0, 0, 90, 0);
mobot1.moveWait();
mobot2.moveWait();
```

rotate the joints on either end of the compound mobot to perform the first portion of the lift, as shown in Figure 25a. The motion is performed as two non-blocking calls to `moveNB()` and then waiting for both movements to finish.

The next lines rotate the inner joints of the compound mobot to lift the mobot one step higher, as shown is Figure 25b.

```
/* Second lift */
mobot1.moveToNB(0, 0, 90,  0);
mobot2.moveToNB(0,  -90, 0, 0);
mobot1.moveWait();
mobot2.moveWait();
```

Again, these lines perform two calls to the non-blocking function `moveNB()` and then wait for the movements to finish.

Finally, we move both mobots back to their zero positions, which drops the compound mobot back onto the ground.

```
mobot1.resetToZeroNB();
mobot2.resetToZeroNB();
mobot1.moveWait();
mobot2.moveWait();
```

15 Commanding Multiple Mobots to Perform Identical Tasks

The class called `CMobotGroup` can be used to control multiple modules simultaneously. The `CMobotGroup` represents a group of mobots. Any command that is given to the group of modules is duplicated to each member of the group.

The majority of the movement functions available in the `CMobot` class are also available in the `CMobotGroup` class. The detailed information for each member function are presented in Appendix B. Following is a complete listing of the available member functions in the `CMobotGroup` class.

Function	Description
`CMobotGroup()`	The CMobotGroup constructor function. This function is called automatically and should not be called explicitly.
`~CMobotGroup()`	The CMobotGroup destructor function. This function is called automatically and should not be called explicitly.
`addRobot()`	Add a mobot to be a member of the mobot group.
`move()`	Move all four joints of the mobots by specified angles.
`moveDistance()`	Move each wheeled mobot in the group a certain distance.
`moveDistanceNB()`	Identical to `moveDistance()` but non-blocking.
`moveBackward()`	Roll on the faceplates toward the backward direction.
`moveBackwardNB()`	Identical to `moveBackward()` but non-blocking.
`moveForward()`	Roll on the faceplates forwards.
`moveForwardNB()`	Identical to `moveForward()` but non-blocking.
`moveNB()`	Identical to `move()` but non-blocking.
`moveJoint()`	Move a motor from its current position by an angle.
`moveJointNB()`	Identical to `moveJoint()` but non-blocking.
`moveJointTo()`	Set the desired motor position for a joint.
`moveJointToNB()`	Identical to `moveJointTo()` but non-blocking.
`moveJointWait()`	Wait until the specified motor has stopped moving.
`moveTo()`	Move all four joints of the mobots to specified absolute angles.
`moveToNB()`	Identical to `moveTo()` but non-blocking.
`moveToZero()`	Instructs all motors to go to their zero positions.
`moveToZeroNB()`	Identical to `moveToZero()` but non-blocking.
`moveWait()`	Wait until all motors have stopped moving.
`setExitState()`	Set the joint behavior on exit for all mobots in the group.
`setJointMovementStateNB()`	Move a single joint on all mobots continuously.
`setJointMovementStateTime()`	Move a single joint on all mobots continuously for a specific amount of time.
`setJointMovementStateTimeNB()`	Move a single joint on all mobots continuously for a specific amount of time.
`setJointSpeed()`	Set a motor's speed setting in radians per second.
`setJointSpeeds()`	Set all motor speeds in radians per second.
`setJointSpeedRatio()`	Set a joints speed setting to a fraction of its maximum speed, a value between 0 and 1.
`setJointSpeedRatios()`	Set all joint speed settings to a fraction of its maximum speed, expressed as a value from 0 to 1.
`setMovementStateNB()`	Move joints continuously. Joints will move untill stopped.
`setMovementStateTime()`	Move joints continuously for a certain amount of time.
`setMovementStateTimeNB()`	Move joints continuously for a certain amount of time.
`setTwoWheelRobotSpeed()`	Move the mobots at a constant forward velocity.
`stop()`	Stop all currently executing motions of the mobot.
`turnLeft()`	Rotate the mobots counterclockwise.
`turnLeftNB()`	Identical to `turnLeft()` but non-blocking.
`turnRight()`	Rotate the mobots clockwise.
`turnRightNB()`	Identical to `turnRight()` but non-blocking.

Compound Motions	These are convenience functions of commonly used compound motions.
`motionArch()`	Move each mobot in the group into an arched configuration.
`motionArchNB()`	Identical to `motionArch()` but non-blocking.
`motionInchwormLeft()`	Inchworm motion towards the left.
`motionInchwormLeftNB()`	Identical to `motionInchwormLeft()` but non-blocking.
`motionInchwormRight()`	Inchworm motion towards the right.
`motionInchwormRightNB()`	Identical to `motionInchwormRight()` but non-blocking.
`motionSkinny()`	Move the mobots into a skinny configuration.
`motionSkinnyNB()`	Identical to `motionSkinnyNB()` but non-blocking.
`motionStand()`	Stand the mobots up on its end.
`motionStandNB()`	Identical to `motionStandNB()` but non-blocking.
`motionTumbleLeft()`	Tumble the mobots end over end.
`motionTumbleLeftNB()`	Identical to `motionTumbleLeftNB()` but non-blocking.
`motionTumbleRight()`	Tumble the mobots end over end.
`motionTumbleRightNB()`	Identical to `motionTumbleRightNB()` but non-blocking.
`motionUnstand()`	Move each mobot in the group currently standing on its end down into zero position.
`motionUnstandNB()`	Identical to `motionUnstand()` but non-blocking.
`motionWait()`	Wait for preprogrammed mobotic motions to complete.

15.1 Demo program `group.ch`

15.1.1 Source Code

```
/* Filename: group.ch
 * Control multiple Mobot modules simultaneously using the CMobotGroup class */
#include <mobot.h>
#define NUM_ROBOTS 13
CLinkbotI robots[NUM_ROBOTS];
CLinkbotIGroup group;

/* Add the two modules to be members of our group */
group.addRobots(robots);
group.connect();

/* Now, any commands given to "group" will cause both mobot1 and mobot2 to
 * execute the command. */
group.move(360, 0, 360);   /* Joints 1 and 4 rotate 360 degrees */
delay(3);                     /* Mobots stand still for 3 seconds */
group.move(360, 0, 360);   /* Joints 1 and 4 rotate 360 degrees */
```

15.1.2 Demo Explanation

The first lines of interest appear as such:

```
CMobot robot1;
CMobot robot2;
CMobotGroup group;
```

These lines declare two robot variables, and one variable which will represent a group of robots. Next, we connect the robot variables to their physical counterparts.

```
robot1.connect();
robot2.connect();
```

Once they are connected, we wish to add both of these robots to our robot group, which we have named `group`.

```
group.addRobot(robot1);
group.addRobot(robot2);
```

Finally, we wish for all of the robots in our robot group, namely `robot1` and `robot2`, to perform an inchworm motion four times, followed by a standing motion. This is done with the following lines:

```
group.motionInchwormLeft(4);  /* Both robots inchworm left 4 times */
group.motionStand();          /* Both robots stand */
```

After the robots stand up, the line

```
group.move(360, 0, 0, 360);   /* Joints 1 and 4 rotate 360 degrees */
```

makes the robots perform a 360 degree rotation on their faceplates while standing.

The next line,

```
delay(3);                     /* Mobots stand still for 3 seconds */
```

makes the robots stand still for three seconds. After standing still for three seconds, the line

```
group.motionUnstand();        /* Mobots get back down from standing */
```

makes both robots move back down from a standing position into a prone position.

15.2 Demo Program `groups.ch`

15.2.1 `groups.ch` Source Code

```
/* Filename: groups.ch
 * Control multiple Mobot groups simultaneously using the CMobotGroup class */
#include <mobot.h>
CMobot robot1;
CMobot robot2;
CMobot robot3;
CMobot robot4;
CMobotGroup groupA;
CMobotGroup groupB;
CMobotGroup groupC;
CMobotGroup groupD;

/* Connect to the robots listed in the configuration file. */
robot1.connect();
robot2.connect();
robot3.connect();
robot4.connect();

/* Add the robots to our groups. The groups should be organized as such:
 * Group A: 1, 2, 3, 4
 * Group B: 1, 2
 * Group C: 3, 4
 * Group D: 1, 2, 3 */

/* Group A */
```

```
groupA.addRobot(robot1);
groupA.addRobot(robot2);
groupA.addRobot(robot3);
groupA.addRobot(robot4);

/* Group B */
groupB.addRobot(robot1);
groupB.addRobot(robot2);

/* Group C */
groupC.addRobot(robot3);
groupC.addRobot(robot4);

/* Group D */
groupD.addRobot(robot1);
groupD.addRobot(robot2);
groupD.addRobot(robot3);

/* Make group B roll forward and group C roll backward at the same time */
groupB.motionRollForwardNB(360);
groupC.motionRollBackwardNB(360);
groupB.motionWait();
groupC.motionWait();

/* Make all the robot stand up */
groupA.motionStand();

/* Make robots 1 and 2 (Group B) rotate counter-clockwise and robots 3 and 4
 * (Group C) rotate clockwise. */
groupB.moveNB(360, 0, 0, 360);
groupC.moveNB(-360, 0, 0, -360);
groupB.moveWait();
groupC.moveWait();

/* Make robot 4 unstand and inchworm while the remaining robots spin. */
groupD.setMovementStateNB(ROBOT_FORWARD, ROBOT_HOLD, ROBOT_HOLD, ROBOT_FORWARD);
robot4.motionUnstand();
robot4.motionInchwormLeft(2);

/* Make the other 3 robots in groupD unstand */
groupD.motionUnstand();
```

15.2.2 Demo Explanation

The program begins by including the `mobot.h` header file and declaring a number of robot variables and robot group variables.

```
#include <mobot.h>
CMobot robot1;
CMobot robot2;
CMobot robot3;
CMobot robot4;
CMobotGroup groupA;
```

```
CMobotGroup groupB;
CMobotGroup groupC;
CMobotGroup groupD;
```

Next, we connect our four robot variables to the first four robots configured in BaroboLink.

```
robot1.connect();
robot2.connect();
robot3.connect();
robot4.connect();
```

Now we begin adding our robots to their groups. We will have four different groups, and each robot will belong to more than one group. They will be organized as such:

- Group A: robot1, robot2, robot3, robot4
- Group B: robot1, robot2
- Group C: robot3, robot4
- Group D: robot1, robot2, robot3

The following code divides the robots up into our groups.

```
/* Group A */
groupA.addRobot(robot1);
groupA.addRobot(robot2);
groupA.addRobot(robot3);
groupA.addRobot(robot4);

/* Group B */
groupB.addRobot(robot1);
groupB.addRobot(robot2);

/* Group C */
groupC.addRobot(robot3);
groupC.addRobot(robot4);

/* Group D */
groupD.addRobot(robot1);
groupD.addRobot(robot2);
groupD.addRobot(robot3);
```

Now we perform some group oriented motions. First, we will have group B roll forward and group C roll backward simultaneously. This is done with calls to non-blocking functions and using the function `motionWait()` to wait for the non-blocking functions to finish.

```
/* Make group B roll forward and group C roll backward at the same time */
groupB.moveForwardNB(360);
groupC.moveBackwardNB(360);
groupB.motionWait();
groupC.motionWait();
```

Next, we make all the robots, which are members of the group `groupA`, stand up.

```
/* Make all the robot stand up */
groupA.motionStand();
```

While the robots are standing, we want group B to spin counter-clockwise, and group C to spin clockwise. To do this, we use the non-blocking function `moveNB()` and then wait for the movements to finish with `moveWait()`.

```
/* Make robots 1 and 2 (Group B) rotate counter-clockwise and robots 3 and 4
 * (Group C) rotate clockwise. */
groupB.moveNB(360, 0, 0, 360);
groupC.moveNB(-360, 0, 0, -360);
groupB.moveWait();
groupC.moveWait();
```

Finally, we want robots 1, 2, and 3 to spin while robot 4 unstands and inchworms twice. As soon as robot 4 is done inchworming, we want robots 1, 2, and 3 to unstand as well. We do this by using a non-blocking continuous move function, `setMovementStateNB()`. While the group is moving, we unstand robot 4 and inchworm it twice. After robot 4 is done inchworming, we unstand Group D.

```
/* Make robot 4 unstand and inchworm while the remaining robots spin. */
groupD.setMovementStateNB(ROBOT_FORWARD, ROBOT_HOLD, ROBOT_HOLD, ROBOT_FORWARD);
robot4.motionUnstand();
robot4.motionInchwormLeft(2);

/* Make the other 3 robots in groupD unstand */
groupD.motionUnstand();
```

16 Data Acquisition, Data Processing, and Application Examples for Learning Algebra

16.1 Example 1

16.1.1 Problem Statement

Roll a robot forward by rotating the wheels 720 degrees at a constant speed of 45 degrees per second. Record the motion of joint 1 during the motion and display a plot of the joint angle versus time. The motion should show that the joint angle θ is a linear function of time in the form $\theta = 45t$.

16.1.2 `dataAcquisition.ch` Source Code

```
/* Filename: dataAcquisition.ch
 * Make a graph of the mobot's joint angle versus time */

#include <mobot.h>
#include <chplot.h>
#include <numeric.h>
CMobot robot;

/* Connect to the robot */
robot.connect();

double speed = 45; /* Degrees/second */
double angle = 720; /* Degrees */
double timeInterval = 0.1; /* Seconds */

/* Figure out how many data points we will need. First, figure out the
```

```
 * approximate amount of time the movement should take. */
double movementTime = angle / speed; /* Seconds */
/* Add an extra second of recording time to make sure the entire movement is
 * recorded */
movementTime = movementTime + 1;
int numDataPoints = movementTime / timeInterval; /* Unitless */

/* Initialize the arrays to be used to store data for time and angle */
array double time[numDataPoints];
array double angles1[numDataPoints];

/* Declare plotting variables */
CPlot plot1, plot2, plot3;

/* Start the motion. First, move robot to zero position */
robot.resetToZero();
/* Set the joint 1 speed to 45 degrees/second */
robot.setJointSpeed(ROBOT_JOINT1, speed);
robot.setJointSpeed(ROBOT_JOINT4, speed);

/* Start capturing data */
robot.recordAngle(ROBOT_JOINT1, time, angles1, numDataPoints, timeInterval);

/* Move the joint 720 degrees */
robot.move(angle, 0, 0, angle);

/* Wait for recording to finish */
robot.recordWait();

/* Plot the data */
plot1.title("Data for Joint Angle 1 versus Time");
plot1.label(PLOT_AXIS_X, "Time (seconds)");
plot1.label(PLOT_AXIS_Y, "Angle (degrees)");
plot1.data2DCurve(time, angles1, numDataPoints);
plot1.grid(PLOT_ON);
plot1.plotting();
```

16.1.3 `dataAcquisition.ch` Explained

The first block of code,

```
#include <mobot.h>
#include <chplot.h>
#include <numeric.h>
CMobot robot;

/* Connect to the robot */
robot.connect();
```

includes header files and declares our robot variable. The header file `chplot.h` is necessary for plotting figures, which will be done later in the program. The header file `numeric.h` is necessary for creating and manipulating computational arrays, which are also used in this program. Both plotting and computational arrays are covered in *Ch User's Guide*.

Next some variables are declared. First, we declare a variable to hold our desired speed:

```
double speed = 45; /* Degrees/second */
```

Now we declare a variable to hold the angle we want to rotate our faceplate joints:

```
double angle = 720; /* Degrees */
```

And another variable to hold our polling interval:

```
double timeInterval = 0.1; /* Seconds */
```

The variable `timeInterval` holds the time in seconds between data acquisition events. The value 0.1 indicates that new data should be acquired from the robot every 0.1 seconds, or ten times a second, in other words. Lower values will result in faster rates of data acquisition and smoother plots.

Next, we need to determine how long to record the data. This requires us to estimate the amount of time our motion will take. We will perform a motion in which the wheels of the robot turn 720 degrees at 45 degrees a second. To find the amount of time the movement will take, we can use the formula

$$t = \frac{\theta}{\omega}$$

where θ is the angle to turn the joint and ω is the speed at which the joint is turning. The line

```
double movementTime = angle / speed; /* Seconds */
```

performs this calculation and stores the result in a variable called `movementTime`.

Because physical systems are never mathematically precise, we must account for some potential fluctuations during the motion of the robot. Because we want to be sure to record the motion in its entirety, it is beneficial to record for a timespan that is actually longer than the estimated movement time. The next line:

```
movementTime = movementTime + 1;
```

adds another second to the estimated movement time.

Finally, we calculate the number of data points which will be captured during our recording session. This information is necessary so that we know how big to make the computational arrays which will be used to store the data. The number of data points can be determined by the formula

$$N = \frac{t}{\Delta t}$$

where t is the total movement time and Δt is the time interval between readings. We declare a new variable based on the previous equation called `numDataPoints` with the following line of code:

```
int numDataPoints = movementTime / timeInterval; /* Unitless */
```

Next, we declare some computational arrays to hold the data we will record.

```
array double time[numDataPoints];
array double angles1[numDataPoints];
```

Note that we have used the variable `numDataPoints` to specify the size of our arrays. The `recordAngle()` function, used later to record the data, requires two arrays to store data. The first array will store a series of timestamps for each data point. Timestamps are stored for greater accuracy of the data. Although the user is able to request a certain time interval, wireless and communication uncertainties may contribute to some fluctuation in communication times and speeds. In order to ensure accurate data, as the Mobot sends joint data to the computer, the Mobot will place an accurate timestamp on each piece of data as it was recorded on the Mobot itself. These timestamps are stored in the `time` array and the joint angles will be stored in the `angles1` array.

Finally, the last variable declared is `threshold`. This variable holds a value in degrees. This threshold declared here is the threshold used later to detect the time where motion first begins. Because the data recording process typically starts a fraction of a second before the motion begins, the plotted motions may not begin at time zero. When detecting the time of the first motion, if any joint moves by an amount more than the threshold value, that time is taken to be the beginning of the motion.

Before we begin the Mobot's motion, we first move it to zero position and set the joints to the correct speeds with the following lines of code:

```
/* Start the motion. First, move robot to zero position */
robot.resetToZero();
/* Set the joint 1 speed to 45 degrees/second */
robot.setJointSpeed(ROBOT_JOINT1, speed);
robot.setJointSpeed(ROBOT_JOINT4, speed);
```

Next, we start capturing the data by using the `recordAngle()` function:

```
robot.recordAngle(ROBOT_JOINT1, time, angles1, numDataPoints, timeInterval, threshold);
```

The recording process begins immediately and continues in the background. Now, we may perform the motion we wish to record, which is to simply rotate each of the faceplates by the amount desired:

```
robot.move(angle, 0, 0, angle);
```

After we have performed all of our motions, we need to ensure that the recording process which had been running in the background is completed. We used the function `recordWait()` to make the program wait for any currently recording tasks to finish:

```
robot.recordWait();
```

After data has been recorded, we use our previously declared plotting variables to generate plots of the data.

```
plot1.title("Data for Joint Angle 1 versus Time");
plot1.label(PLOT_AXIS_X, "Time (seconds)");
plot1.label(PLOT_AXIS_Y, "Angle (degrees)");
plot1.data2DCurve(time, angles1, numDataPoints);
plot1.grid(PLOT_ON);
plot1.plotting();
```

The previous lines of code set the plot title, set the X-axis label, set the Y-axis label, insert the plot data, turn the grid on, and plot the data, respectively. The plot that is generated is shown in Figure 26.

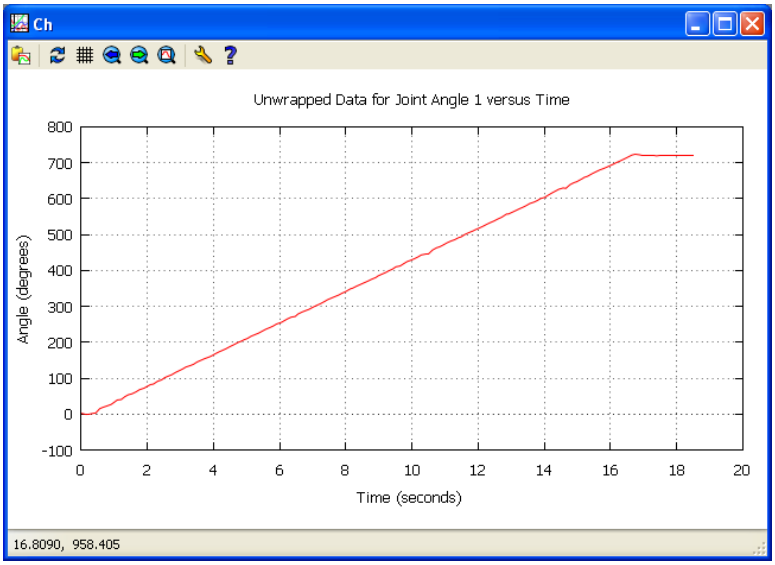


Figure 26: The data recorded by `dataAcquisition.ch`.

16.1.4 `dataAcquisition1.ch` Source Code

```
/* Filename: dataAcquisition1.ch
 * Make a graph of the mobot's joint angle versus time */

#include <mobot.h>
#include <chplot.h>
CMobot robot;

/* Connect to the robot */
robot.connect();

double speed = 45; /* Degrees/second */
double angle = 720; /* Degrees */
double timeInterval = 0.1; /* Seconds */

/* Figure out how many data points we will need. First, figure out the
 * approximate amount of time the movement should take. */
double movementTime = angle / speed; /* Seconds */
/* Add an extra second of recording time to make sure the entire movement is
 * recorded */
movementTime = movementTime + 1;
int numDataPoints = movementTime / timeInterval; /* Unitless */

/* Initialize the arrays to be used to store data for time and angle */
array double time[numDataPoints];
array double angles1[numDataPoints];

/* Declare plotting variables */
CPlot plot;

/* Start the motion. First, move robot to zero position */
```

```
robot.resetToZero();
/* Set the joint 1 speed to 45 degrees/second */
robot.setJointSpeed(ROBOT_JOINT1, speed);
robot.setJointSpeed(ROBOT_JOINT4, speed);

/* Start capturing data */
robot.recordAngle(ROBOT_JOINT1, time, angles1, numDataPoints, timeInterval, 1);

/* Move the joint 720 degrees */
robot.move(angle, 0, 0, angle);

/* Wait for recording to finish */
robot.recordWait();


/* Plot the data */
plot.title("Unwrapped and shifted Data for Joint Angle 1 versus Time");
plot.label(PLOT_AXIS_X, "Time (seconds)");
plot.label(PLOT_AXIS_Y, "Angle (degrees)");
plot.data2DCurve(time, angles1, numDataPoints);
plot.grid(PLOT_ON);
plot.plotting();
```

16.2 Example 2

It is also possible to record the joint angles for all four joints simultaneously with the `recordAngles()` function. The function is conceptually similar to `recordAngle()`, except that it obtains 4 joint angles for each timestamp.

16.2.1 Problem Statement

Record and plot the motion of all 4 joints of the Mobot as it performs the following two motions:

1. Turn right by rotating joints 360 degrees.
2. Inchworm to the left twice.

16.2.2 `dataAcquisition2.ch` Source Code

```
/* Filename: dataAcquisition2.ch
 * Make a graph of the mobot's joint angle versus time */

#include <mobot.h>
#include <chplot.h>
CMobot robot;

/* Connect to the robot */
robot.connect();

double timeInterval = 0.1; /* Seconds */

/* Record for 20 seconds */
double movementTime = 20;
```

```
int numDataPoints = movementTime / timeInterval; /* Unitless */

/* Initialize the arrays to be used to store data */
array double time[numDataPoints];
array double angles1[numDataPoints];
array double angles2[numDataPoints];
array double angles3[numDataPoints];
array double angles4[numDataPoints];

/* Declare plotting variables */
CPlot plot;

/* Set all joint speeds to 45 degrees/second */
robot.setJointSpeeds(45, 45, 45, 45);

/* Start the motion. First, move robot to zero position */
robot.resetToZero();

/* Start capturing data */
robot.recordAngles( time, angles1, angles2, angles3, angles4,
                    numDataPoints, timeInterval, 1);

/* Perform the standing and unstanding motions */
robot.motionTurnRight(360);
robot.motionInchwormLeft(2);

/* Wait for recording to finish */
robot.recordWait();

plot.title("Unwrapped Data for Joint Angles versus Time");
plot.label(PLOT_AXIS_X, "Time (seconds)");
plot.label(PLOT_AXIS_Y, "Angle (degrees)");
plot.data2DCurve(time, angles1, numDataPoints);
plot.data2DCurve(time, angles2, numDataPoints);
plot.data2DCurve(time, angles3, numDataPoints);
plot.data2DCurve(time, angles4, numDataPoints);
plot.legend("Joint 1", 0);
plot.legend("Joint 2", 1);
plot.legend("Joint 3", 2);
plot.legend("Joint 4", 3);
plot.grid(PLOT_ON);
plot.plotting();
```

16.2.3 `dataAcquisition2.ch` Explained

Similar to the previous data acquisition demo, we begin by including header files, declaring our robot variable, and connecting to the robot:

```
#include <mobot.h>
#include <chplot.h>
CMobot robot;

/* Connect to the robot */
```

```
robot.connect();
```

We also set our data acquisition interval, movement time, and calculate the number of data points:

```
double timeInterval = 0.1; /* Seconds */

/* Record for 20 seconds */
double movementTime = 20;
int numDataPoints = movementTime / timeInterval; /* Unitless */
```

For this motion, the value of 20 seconds was decided through trial-and-error. For complex motions such as inchworming, it is difficult to accurately estimate the amount of time the motion will take to complete. Through experimentation and experience, it was determined that 20 seconds allowed for a sufficient amount of time to record the turning motion and the inchworming motion.

Next, we declare our computational arrays for storing data and plotting, similar to the previous demo. One main difference is we declare a separate computational array to store data for each joint angle.

```
/* Initialize the arrays to be used to store data */
array double time[numDataPoints];
array double angles1[numDataPoints];
array double angles2[numDataPoints];
array double angles3[numDataPoints];
array double angles4[numDataPoints];

/* Declare plotting variables */
CPlot plot2;
double threshold = 1.0; /* 1 degree for time shifting */
```

Before we begin the motion, we set all the joint speeds to 45 degrees/second and move the robot into its zero position.

```
/* Set all joint speeds to 45 degrees/second */
robot.setJointSpeeds(45, 45, 45, 45);

/* Start the motion. First, move robot to zero position */
robot.resetToZero();
```

Next, we begin recording the data.

```
robot.recordAngles( time, angles1, angles2, angles3, angles4,
                    numDataPoints, timeInterval, threshold);
```

Similar to `recordAngle()`, the recording process occurs in the background as the main program continues. We next execute our desired motions, which are to turn the robot to the right by rotating each wheel 360 degrees, and then inchworming to the left twice:

```
robot.turnRight(360);
robot.motionInchwormLeft(2);
```

Finally, we wait for the recording to finish.

```
robot.recordWait();
```

Now that all of the data has been acquired, we plot the data.

```
plot.title("Unwrapped Data for Joint Angles versus Time");
plot.label(PLOT_AXIS_X, "Time (seconds)");
plot.label(PLOT_AXIS_Y, "Angle (degrees)");
plot.data2DCurve(time, angles1, numDataPoints);
plot.data2DCurve(time, angles2, numDataPoints);
plot.data2DCurve(time, angles3, numDataPoints);
plot.data2DCurve(time, angles4, numDataPoints);
plot.legend("Joint 1", 0);
plot.legend("Joint 2", 1);
plot.legend("Joint 3", 2);
plot.legend("Joint 4", 3);
plot.grid(PLOT_ON);
plot.plotting();
```

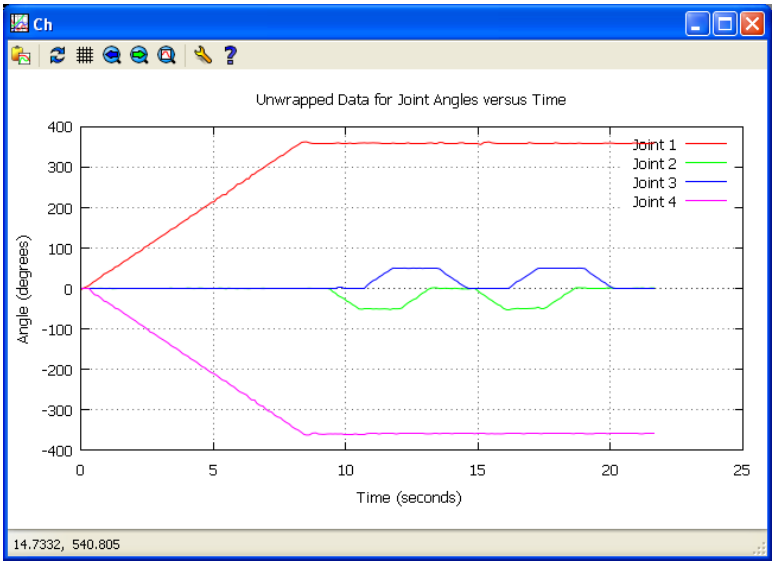


Figure 27: The shifted data recorded by `dataAcquisition2.ch`.

Figure 27 shows the captured data from the Mobot. During the first part of the motions, joints 1 and 4 move in opposite directions to rotate the robot to the right. After rotating 360 degrees, joints 2 and 3 move to inchworm the robot to the left twice.

16.3 Example 3

16.3.1 Problem Statement

A robot is equipped with 3.5 inch diameter wheels. Write a piece of code which rolls the robot forward at constant speed of 2.5 inches per second for a distance of 12 inches. Verify that the data is linear and follows the relation $d = 2.5t$

16.3.2 `dataAcquisition3.ch` Source Code

```
/* Filename: dataAcquisition3.ch
 * Make a graph of the mobot's joint angle versus time */

#include <mobot.h>
```

```
#include <chplot.h>
CMobot robot;

/* Connect to the robot */
robot.connect();

double speed = 2.5; /* inches / second */
double distance = 12; /* inches */
double radius = 3.5/2.0; /* inches */
double angle = distance2angle(radius, distance); /* degrees */

/* Figure out how many data points we will need. First, figure out the
 * approximate amount of time the movement should take. */
double movementTime = distance / speed; /* Seconds */
/* Add an extra second of recording time to make sure the entire movement is
 * recorded */
movementTime = movementTime + 1;
double timeInterval = 0.1; /* seconds */
int numDataPoints = movementTime / timeInterval; /* Unitless */

/* Initialize the arrays to be used to store data */
array double time[numDataPoints];
array double angles1[numDataPoints];
array double distances[numDataPoints];

/* Declare plotting variables */
CPlot plot;

/* Start the motion. First, move robot to zero position */
robot.resetToZero();
/* Set robot wheel speed */
robot.setTwoWheelRobotSpeed(speed, radius);

/* Start capturing data */
robot.recordAngle(ROBOT_JOINT1, time, angles1, numDataPoints, timeInterval);

/* Roll the robot the calculated distance */
robot.motionRollForward(angle);

/* Wait for recording to finish */
robot.recordWait();

/* Convert angles to displacement */
distances = angle2distance(radius, angles1);

/* Plot the data */
plot.title("Displacement versus Time");
plot.label(PLOT_AXIS_X, "Time (seconds)");
plot.label(PLOT_AXIS_Y, "Displacement (inches)");
plot.data2DCurve(time, distances, numDataPoints);
plot.grid(PLOT_ON);
plot.plotting();
```

16.3.3 `dataAcquisition3.ch` Explained

The first lines include header files and set up the robot similar to the previous demos:

```
#include <mobot.h>
#include <chplot.h>
CMobot robot;

/* Connect to the robot */
robot.connect();
```

Next, we initialize some variables. First, we make a variable to store the speed we want to move the robot.

```
double speed = 2.5; /* inches / second */
```

Next, a variable for the distance.

```
double distance = 12; /* inches */
```

And a variable for the wheel radius...

```
double radius = 3.5/2.0; /* inches */
```

Now we wish to calculate the angle that the wheels need to turn in order to travel a distance of 12 inches. To do this, we use a function called `distance2angle()`. The `distance2angle()` function takes a wheel radius and the distance as arguments and returns the angle the wheel must turn to travel that distance.

```
double angle = distance2angle(radius, distance); /* degrees */
```

Internally, the angle in degrees is calculated by the formula

$$\theta = \left(\frac{d}{r}\right) * 180/\pi$$

where θ is the angle in degrees, d is the distance traveled, and r is the radius of the wheel.

For the next step, we must calculate our estimated movement time, similar to the `dataAcquisition.ch` demo presented earlier.

```
double movementTime = distance / speed; /* Seconds */
```

Is in previous demos, we include an extra second in our movement time. We also calculate the number of data points based on the total movement time and the time interval.

```
movementTime = movementTime + 1;
double timeInterval = 0.1; /* seconds */
int numDataPoints = movementTime / timeInterval; /* Unitless */
```

Now, we allocate our computational arrays for storing data. We have arrays for the time, angles, and also distances.

```
array double time[numDataPoints];
array double angles1[numDataPoints];
array double distances[numDataPoints];

/* Declare plotting variables */
CPlot plot;
double threshold = 1.0; /* Degrees */
```

Before beginning the motion, we move the robot to zero position and set the wheel speeds.

```
/* Start the motion. First, move robot to zero position */
robot.resetToZero();
/* Set robot wheel speed */
robot.setTwoWheelRobotSpeed(speed, radius);
```

Start recording the angle of the first joint.

```
robot.recordAngle(ROBOT_JOINT1, time, angles1, numDataPoints, timeInterval, threshold);
```

Move the robot forward by the angle calculated earlier with `distance2angle()`.

```
robot.moveForward(angle);
```

Wait for the recording to finish.

```
robot.recordWait();
```

We convert the angles back to a distance using the `angle2distance()` function, which takes the radius of a wheel and the angle moved in degrees as arguments and return the distance traveled. The arguments can be single variables or computational arrays. If the angle argument is a computational array, the value returned is also a computational array. The `angle2distance()` function converts the angle to a distance using the following formula.

$$d = \left(\theta \frac{\pi}{180}\right) r$$

where d is the distance traveled, θ is the angle rotated in degrees, and r is the radius of the wheel. The units of distance for d will be the same units as those chosen for r. For instance, if r is expressed in inches, the result

```
/* Convert angles to displacement */
distances = angle2distance(radius, angles1);
```

Finally, we create a plot of the data.

```
plot.title("Displacement versus Time");
plot.label(PLOT_AXIS_X, "Time (seconds)");
plot.label(PLOT_AXIS_Y, "Displacement (inches)");
plot.data2DCurve(time, distances, numDataPoints);
plot.grid(PLOT_ON);
plot.plotting();
```

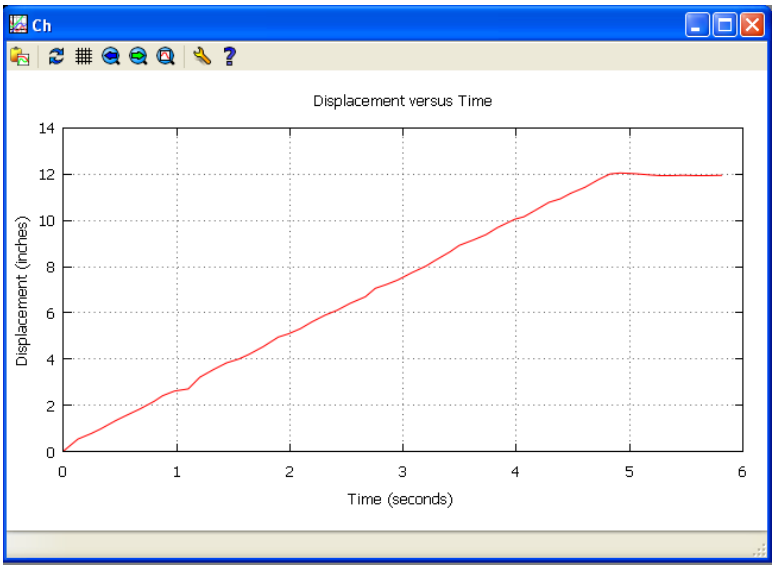


Figure 28: The data recorded by `dataAcquisition2.ch`.

Figure 28 displays the distance data acquired during the motion. It can be seen that the robot traveled to a distance of 12 inches, and then stopped. The expected relation between time and distance is the linear function

$$d = 2.5t$$

which may be verified on the graph. Furthermore, the time that it should take to reach 12 inches is

$$t = d/2.5 = 12/2.5 = 4.8 \text{ seconds} \tag{1}$$

which may also be verified from the graph.

A CMobot Class

A.1 Data Types

The data types defined in the header file `mobot.h` are described in this appendix. These data types are used by the Mobot library to represent certain values, such as joint id's and motor directions.

Data Type	Description
`robotJointId_t`	An enumerated value that indicates a Mobot joint.
`robotJointState_t`	The current state of a Mobot joint.
`robotRecordData_t`	Recorded time and angle data.

A.2 `robotJointId_t`

This datatype is an enumerated type used to identify a joint on the Mobot. Valid values for this type are:

```
typedef enum mobot_joints_e {
  ROBOT_JOINT1 = 1,
  ROBOT_JOINT2 = 2,
  ROBOT_JOINT3 = 3,
  ROBOT_JOINT4 = 4
} robotJointId_t;
```

Value	Description
`ROBOT_JOINT1`	Joint number 1 on the Mobot, which is a faceplate joint.
`ROBOT_JOINT2`	Joint number 2 on the Mobot, which is a body joint.
`ROBOT_JOINT3`	Joint number 3 on the Mobot, which is a body joint.
`ROBOT_JOINT4`	Joint number 4 on the Mobot, which is a faceplate joint.

A.3 `robotJointState_t`

This datatype is an enumerated type used to designate the current movement state of a joint. The values may be retrieved from the mobot with the `getJointState()` function and may be set with the `setMovementState()` family of functions. Valid values are:

```
typedef enum mobot_joint_state_e {
  ROBOT_NEUTRAL   = 0,
  ROBOT_FORWARD   = 1,
  ROBOT_BACKWARD  = 2,
  ROBOT_HOLD      = 3
} robotJointState_t;
```

Value	Description
`ROBOT_NEUTRAL`	This value indicates that the joint is not moving and is not actuated. The joint is freely back-drivable.
`ROBOT_FORWARD`	This value indicates that the joint is currently moving forward.
`ROBOT_BACKWARD`	This value indicates that the joint is currently moving backward.
`ROBOT_HOLD`	This value indicates that the joint is currently not moving and is holding its current position. The joint is not currently back-drivable.

A.4 CMobot API Functions

The header file **mobot.h** defines all the data types, macros and function prototypes for the mobot API library. The header file declares a class called `CMobot` which contains member functions which may be used to control the mobot.

Table 1: CMobot Member Functions.

Function	Description
`CMobot()`	The CMobot constructor function. This function is called automatically and should not be called explicitly.
`~CMobot()`	The CMobot destructor function. This function is called automatically and should not be called explicitly.
`blinkLED()`	Blink the LED on a Mobot module.
`connect()`	Connect to a remote mobot module. This function connects to the first mobot listed in the Barobo configuration file. To edit the configuration file, use the mobot control graphical user interface, and select the menu item "Mobot → Configure Mobot Bluetooth".
`connectWithBluetoothAddress()`	Connect to a mobot module by specifying its Bluetooth address.
`connectWithIPAddress()`	Connect to a mobot module by specifying a remote computer's IP address.
`disconnect()`	Disconnect from a mobot module.
`disableButtonCallback()`	Reverts a Mobot's buttons to their factory-default functions.
`driveTo()`	Move all four joints of the mobot to specified absolute angles.
`driveToNB()`	Move all four joints of the mobot to specified absolute angles.
`enableButtonCallback()`	Enables a user-driven callback for handling button-press events on a module.
`getJointAngle()`	Get a joint's angle.
`getJointAngleAverage()`	Get a joint's angle.
`getJointAngles()`	Get joint angles for all joints.
`getJointAnglesAverage()`	Get joint angles for all joints.
`getJointMaxSpeed()`	Get a joint's maximum speed in radians per second.
`getJointSafetyAngle()`	Get the Mobot's safety angle limit.
`getJointSafetyAngleTimeout()`	Get the Mobot's safety angle limit timeout.
`getJointSpeed()`	Get a motor's current speed setting in radians per second.
`getJointSpeeds()`	Get all motor's current speed settings in radians per second.
`getJointSpeedRatio()`	Get a motor's speed as a ratio of the motor's maximum speed.
`getJointSpeedRatios()`	Get all motor speeds as ratios of the motor's maximum speed.
`getJointState()`	Get a motor's current status.
`isConnected()`	This function is used to check the connection to a mobot.
`isMoving()`	This function is used to check if any joints are currently in motion.
`move()`	Move all four joints of the mobot by specified angles.
`moveNB()`	Identical to `move()` but non-blocking.
`moveBackward()`	Roll on the faceplates toward the backward direction.
`moveBackwardNB()`	Identical to `moveBackward()` but non-blocking.
`moveForward()`	Roll on the faceplates forwards.
`moveForwardNB()`	Identical to `moveForward()` but non-blocking.
`moveJoint()`	Move a joint.
`moveJointNB()`	Move a joint.
`moveJointTo()`	Set the desired motor position for a joint.
`moveJointToNB()`	Identical to `moveJointTo()` but non-blocking.
`moveJointWait()`	Wait until the specified motor has stopped moving.
`moveTo()`	Move all four joints of the mobot to specified absolute angles.
`moveToNB()`	Identical to `moveTo()` but non-blocking.
`moveToZero()`	Instruct all motors to go to their zero positions.
`moveToZeroNB()`	Identical to `moveToZero()` but non-blocking.
`moveWait()`	Wait until all motors have stopped moving.
`recordAngle()`	Begin recording the angle of a joint.
`recordAngleBegin()`	Begin recording the angle of a joint.
`recordAngleEnd()`	End recording the angle of a joint.
`recordAngles()`	Begin recording the angle of all joints.
`recordAnglesBegin()`	Begin recording the angle of all joints.
`recordAnglesEnd()`	End recording the angle of all joints.
`recordDistanceBegin()`	Begin recording the distance travelled of a joint.
`recordDistanceEnd()`	End recording the distance of a joint.

Table 1: CMobot Member Functions (Continued)

Function	Description
`resetToZero()`	Instruct all motors to go to their zero positions.
`setExitState()`	Set the exit state of a joint.
`setJointMovementStateNB()`	Set the movement state of a joint.
`setJointMovementStateTime()`	Set the movement state of a joint for a period of time.
`setJointMovementStateTimeNB()`	Set the movement state of a joint for a period of time.
`setJointSafetyAngle()`	Set the Mobot's safety angle limit. The default value is 10 degrees.
`setJointSafetyAngleTimeout()`	Set the Mobot's safety angle limit timeout. The default value is 0.5 seconds.
`setJointSpeed()`	Set a motor's speed setting in radians per second.
`setJointSpeeds()`	Set all motor speeds in radians per second.
`setJointSpeedRatio()`	Set a joints speed setting to a fraction of its maximum speed, a value between 0 and 1.
`setJointSpeedRatios()`	Set all joint speed settings to a fraction of its maximum speed, expressed as a value from 0 to 1.
`setMovementStateNB()`	Change the movement states of the joints. Movement states include moving forward, backward, neutral, or holding their current position.
`setMovementStateTime()`	Change the movement states of the joints for a certain amount of time.
`setMovementStateTimeNB()`	Change the movement states of the joints for a certain amount of time.
`stopAllJoints()`	Stop all currently executing motions of the mobot.
`stopOneJoint()`	Stop a joint on the mobot.
`stopTwoJoints()`	Stop two joints on the mobot.
`stopThreeJoints()`	Stop three joints on the mobot.
`turnLeft()`	Rotate the mobot counterclockwise.
`turnLeftNB()`	Identical to `turnLeft()` but non-blocking.
`turnRight()`	Rotate the mobot clockwise.
`turnRightNB()`	Identical to `turnRight()` but non-blocking.

Table 2: CMobot Member Functions for Compound Motions.

Compound Motions	These are convenience functions of commonly used compound motions.
`motionArch()`	Arch the mobot for better ground clearance.
`motionArchNB()`	Identical to `motionArch` but non-blocking.
`moveDistance()`	Roll a wheeled Mobot a certain distance.
`moveDistanceNB()`	Identical to `moveDistance` but non-blocking.
`motionInchwormLeft()`	Inchworm motion towards the left.
`motionInchwormLeftNB()`	Identical to `motionInchwormLeft` but non-blocking.
`motionInchwormRight()`	Inchworm motion towards the right.
`motionInchwormRightNB()`	Identical to `motionInchwormRight` but non-blocking.
`motionSkinny()`	Move the mobot into a skinny profile.
`motionSkinnyNB()`	Identical to `motionSkinny()` but non-blocking.
`motionStand()`	Stand the mobot up on its end.
`motionStandNB()`	Identical to `motionStand()` but non-blocking.
`motionTumbleLeft()` ..	Perform the tumbling motion.
`motionTumbleLeftNB()`	Identical to `motionTumbleLeft()` but non-blocking.
`motionTumbleRight()` .	Perform the tumbling motion in the opposite direction of "motion-TumbleLeft".
`motionTumbleRightNB()`	Identical to `motionTumbleRight()` but non-blocking.
`motionWait()`	Wait for a motion to finish.

CMobot::blinkLED()

Synopsis
```
#include <mobot.h>
int CMobot::blinkLED(double delay, int numBlinks);
```

Purpose
Blink the on-board LED on a Mobot module.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters

`delay`	The amount of time between blinks.
`numBlinks`	The number of times to blink the LED.

Description
This function is used to blink or flash the LED on a Mobot module. The first argument, `delay`, is used to control the speed of the blinking LED, and the second argument, `numBlinks`, controls the number of times to blink. For instance, the line

```
mobot.blinkLED(0.1, 10);
```

would cause a mobot to blink 10 times fast, and the line

```
mobot.blinkLED(1, 2);
```

would cause the mobot to do two slow blinks.
Example

See Also

CMobot::connect()

Synopsis
```
#include <mobot.h>
int CMobot::connect();
```

Purpose
Connect to a remote mobot via Bluetooth.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters
None.

Description
This function is used to connect to a mobot. The function looks inside of a Barobo configuration file and connects to the first mobot listed in the file. The configuration file may be created and/or modified using the Mobot Controller Interface, and selecting the "Mobot → Configure Mobot Bluetooth" menu item.

Example
Please see the example in Section 9.1.2 on page 30.

See Also
`connectWithBluetoothAddress(), disconnect()`

CMobot::connectWithBluetoothAddress()

Synopsis
```
#include <mobot.h>
int CMobot::connectWithBluetoothAddress(char address[], int channel = 1);
```

Purpose
Connect to a remote mobot via Bluetooth by specifying the specific Bluetooth address of the device.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters
`address` The Bluetooth address of the mobot.
`channel` (optional) The Bluetooth channel that the listening program is listening on. The default channel is channel 1.

Description
This function is used to connect to a mobot.
Example

```
mobot.connectWithBluetoothAddress("00:06:66:45:DA:02", 1);
mobot.connectWithBluetoothAddress("00:06:66:45:DA:F3");
```

See Also
`connect(), disconnect()`

CMobot::connectWithIPAddress()

Synopsis
```
#include <mobot.h>
int CMobot::connectWithIPAddress(char address[], char port[] = "5768");
```

Purpose
Connect to a remote mobot connected to a remote computer over the internet.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters
`address` The IP address of the remote computer.
`port` (optional) The port to connect to.

Description
This function is used to connect to a mobot.
Example

```
mobot.connectWithIPAddress("192.168.0.132", "8734");
mobot.connectWithIPAddress("mobots.example.com", "5768");
mobot.connectWithIPAddress("192.168.100.221");
```

See Also
`connect()`, `disconnect()`

CMobot::disconnect()

Synopsis
```
#include <mobot.h>
int CMobot::disconnect();
```

Purpose
Disconnect from a remote mobot.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters
None.

Description
This function is used to disconnect from a mobot. A call to this function is not necessary before the termination of a program. It is only necessary if another connection will be established within the same program at a later time.

Example

See Also
`connect()`, `connectWithBluetoothAddress()`

CMobot::driveJointTo()
CMobot::driveJointToNB()

Synopsis
```
#include <mobot.h>
int CMobot::driveJointTo(moboJointId_t id, double angle);
int CMobot::driveJointToNB(moboJointId_t id, double angle);
```

Purpose
Move all of the joints of a mobot to the specified positions.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters

`id`	The id of the joint to move.
`angle`	The absolute position to move the joint, expressed in degrees.

Description

Note that this function is similar to the `moveJointTo()` family of functions, except that this variant ignores the speed settings of the robot joints. This function causes the robot to move its joints towards its goal using a Proportional-Integral-Derivative (PID) controller.

CMobot::driveJointTo()
This function moves all of the joints of a mobot to the specified absolute positions.

CMobot::driveJointToNB()
This function moves all of the joints of a mobot to the specified absolute positions.

The function `driveJointToNB()` is the non-blocking version of the `driveJointTo()` function, which means that the function will return immediately and the physical mobot motion will occur asynchronously. For more details on blocking and non-blocking functions, please refer to Section 13 on page 44.

Example

See Also

CMobot::driveTo()
CMobot::driveToNB()

Synopsis

```
#include <mobot.h>
int CMobot::driveTo(double angle1, double angle2, double angle3, double angle4);
int CMobot::driveToNB(double angle1, double angle2, double angle3, double angle4);
```

Purpose
Move all of the joints of a mobot to the specified positions.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters

`angle1`	The absolute position to move joint 1, expressed in degrees.
`angle2`	The absolute position to move joint 2, expressed in degrees.
`angle3`	The absolute position to move joint 3, expressed in degrees.
`angle4`	The absolute position to move joint 4, expressed in degrees.

Description

Note that this function is similar to the `moveTo()` family of functions, except that this variant ignores the speed settings of the robot joints. This function causes the robot to move its joints towards its goal using a Proportional-Integral-Derivative (PID) controller.

CMobot::driveTo()
This function moves all of the joints of a mobot to the specified absolute positions.

CMobot::driveToNB()
This function moves all of the joints of a mobot to the specified absolute positions.

The function `driveToNB()` is the non-blocking version of the `driveTo()` function, which means that the function will return immediately and the physical mobot motion will occur asynchronously. For more details on blocking and non-blocking functions, please refer to Section 13 on page 44.

Example

See Also

CMobot::getJointAngle()
CMobot::getJointAngleAbs()

Synopsis
```
#include <mobot.h>
int CMobot::getJointAngle(mobotJointId_t id, double &angle);
int CMobot::getJointAngleAbs(mobotJointId_t id, double &angle);
```

Purpose
Retrieve a mobot joint's current angle.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters

`id`	The joint number. This is an enumerated type discussed in Section **??** on page **??**.
`angle`	A variable to store the current angle of the mobot motor. The contents of this variable will be overwritten with a value that represents the motor's angle in degrees.

Description
This function gets the current motor angle of a Mobot's motor. The angle returned is in units of degrees and is accurate to roughly ± 0.17 degrees.

The function `getJointAngle()` always returns an angle between -180 and +180 degrees. The `getJointAngleAbs()` function, however, gets the total angle the joint has turned since the mobot has been powered on. For instance, if the faceplate joint 1 has been rotated two full rotations after initial power up, the function `getJointAngle()` will report that the joint is at angle 0, whereas the function `getJointAngleAbs()` will report that the joint angle is 720 degrees.

Example

See Also

CMobot::getJointAngleAverage()

Synopsis
```
#include <mobot.h>
int CMobot::getJointAngleAverage(mobotJointId_t id, double &angle, int numReadings = 10);
```

Purpose
Retrieve a mobot joint's current angle.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters

`id`	The joint number. This is an enumerated type discussed in Section **??** on page **??**.
`angle`	A variable to store the current angle of the mobot motor. The contents of this variable will be overwritten with a value that represents the motor's angle in degrees.
`numReadings`	(Optional) The number of independent angle readings to measure and average. More readings taken may yield a more accurate angle measurement, but will also take more time to perform. If this value is omitted, the default value of 10 readings is used.

Description
This function gets the current motor angle of a Mobot's motor. In most cases, this function may yield a more accurate value than the `getJointAngle()` function because this function averages a number of separate angle readings. Averaging multiple readings may reduce the effect of noise on the angle measurements.

Example

See Also

CMobot::getJointAngles()
CMobot::getJointAnglesAbs()

Synopsis
```
#include <mobot.h>
int CMobot::getJointAngles(
    double &angle1,
    double &angle2,
    double &angle3,
    double &angle4);
int CMobot::getJointAnglesAbs(
    double &angle1,
    double &angle2,
    double &angle3,
    double &angle4);
```

Purpose
Retrieve a Mobot's current joint angles.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters

`angle1`	A variable to store the current angle of the mobot motor. The contents of this variable will be overwritten with a value that represents the motor's angle in degrees.
`angle2`	...
`angle3`	...
`angle4`	...

Description
This function gets the current motor angles of a Mobot's motors. The angle returned is in units of degrees and is accurate to roughly ± 0.17 degrees.

The function `getJointAngles()` always returns an angle between -180 and +180 degrees. The `getJointAnglesAbs()` function, however, gets the total angle the joint has turned since the mobot has been powered on. For instance, if the faceplate joint 1 has been rotated two full rotations after initial power up, the function

`getJointAngles()` will report that the joint is at angle 0, whereas the function `getJointAnglesAbs()` will report that the joint angle is 720 degrees.

Example

See Also

CMobot::getJointAnglesAverage()

Synopsis

```
#include <mobot.h>
int CMobot::getJointAnglesAverage(
    double &angle1,
    double &angle2,
    double &angle3,
    double &angle4,
    int numReadings = 10);
```

Purpose

Retrieve a Mobot's current joint angles.

Return Value

The function returns 0 on success and non-zero otherwise.

Parameters

`angle1`	A variable to store the current angle of the mobot motor. The contents of this variable will be overwritten with a value that represents the motor's angle in degrees.
`angle2`	...
`angle3`	...
`angle4`	...
`numReadings`	(Optional) The number of independent angle readings to measure and average. More readings taken may yield a more accurate angle measurement, but will also take more time to perform. If this value is omitted, the default value of 10 readings is used.

Description

This function gets the current motor angles of a Mobot's motors. In most cases, this function may yield a more accurate value than the `getJointAngles()` function because this function averages a number of separate angle readings. Averaging multiple readings may reduce the effect of noise on the angle measurements.

Example

See Also

CMobot::getJointMaxSpeed()

Synopsis

```
#include <mobot.h>
int CMobot::getJointMaxSpeed(mobotJointId_t id, double &speed);
```

Purpose

Get the maximum speed of a joint on the mobot.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters

`id`	The joint number. This is an enumerated type discussed in Section **??** on page **??**.
`speed`	A variable of type `double`. The value of this variable will be overwritten with the maximum speed setting of the joint, which is in units of degrees per second.

Description
This function is used to find the maximum speed setting of a joint. This is the maximum speed at which the joint will accept speed setting from the function `setJointSpeed()`. The values are in units of degrees per second.
Example

See Also
`getJointSpeed()`, `getJointMaxSpeedRatio()`, `setJointSpeed()`, `setJointSpeedRatio()`

CMobot::getJointSafetyAngle()

Synopsis
```
#include <mobot.h>
int CMobot::getJointSafetyAngle(double &degrees);
```

Purpose
Get the current angle safety limit of the Mobot.

Return Value
The function returns 0 on success and -1 on failure.

Parameters
A variable which will be overwritten with the safety angle limit in degrees.

Description
The Mobot is equipped with a safety feature to protect itself and its surrounding environment. When a motor deviates by a certain amount from its expected value, the Mobot will shut off all power to the motor, in case it has hit an obstacle, or for any other reason. The amount of deviation required to trigger the safety protocol is the joint safety angle which can be retrieved using this function. The default setting is 10 degrees.
Example

See Also
`getJointSafetyAngleTimeout()`, `setJointSafetyAngle()`, `setJointSafetyAngleTimeout()`

CMobot::getJointSafetyAngleTimeout()

Synopsis
```
#include <mobot.h>
int CMobot::getJointSafetyAngleTimeout(double &seconds);
```

Purpose
Get the current angle safety limit timeout of the Mobot.

Return Value
The function returns 0 on success and -1 on failure.

Parameters
A variable which will be overwritten with the safety angle limit timeout in seconds.

Description
The Mobot is equipped with a safety feature to protect itself and its surrounding environment. When a motor deviates by a certain amount from its expected value, the Mobot will shut off all power to the motor after a certain period of time, in case it has hit an obstacle, or for any other reason. The period of time that the mobot waits before shutting the motor off is the joint safety angle timeout, which can be retrieved with this function. The default value for the timeout is 500 milliseconds, or 0.5 seconds.
Example

See Also
`getJointSafetyAngle()`, `setJointSafetyAngle()`, `setJointSafetyAngleTimeout()`

CMobot::getJointSpeed()

Synopsis
```
#include <mobot.h>
int CMobot::getJointSpeed(mobotJointId_t id, double &speed);
```

Purpose
Get the speed of a joint on the mobot.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters

`id`	The joint number to pose. This is an enumerated type discussed in Section **??** on page **??**.
`speed`	A variable of type `double`. The value of this variable will be overwritten with the current speed setting of the joint, which is in units of degrees per second.

Description
This function is used to find the current speed setting of a joint. This is the speed at which the joint will move when given motion commands. The values are in units of degrees per second.
Example

See Also
`getJointMaxSpeed()`, `getJointSpeedRatio()`, `setJointSpeed()`, `setJointSpeedRatio()`

CMobot::getJointSpeedRatio()

Synopsis
```
#include <mobot.h>
int CMobot::getJointSpeedRatio(mobotJointId_t id, double &ratio);
```

Purpose
Get the speed ratio settings of a joint on the mobot.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters

`id`	Retrieve the speed ratio setting of this joint. This is an enumerated type discussed in Section **??** on page **??**.
`ratio`	A variable of type double. The value of this variable will be overwritten with the current speed ratio setting of the joint.

Description
This function is used to find the speed ratio setting of a joint. The speed ratio setting of a joint is the percentage of the maximum joint speed, and the value ranges from 0 to 1. In other words, if the ratio is set to 0.5, the joint will turn at 50% of its maximum angular velocity while moving continuously or moving to a new goal position.

Example

See Also
`setJointSpeeds()`, `getJointSpeedRatio()`, `getJointSpeed()`

CMobot::getJointSpeedRatios()

Synopsis
```
#include <mobot.h>
int CMobot::getJointSpeedRatios(double &ratio1, double &ratio2, double &ratio3, double &ratio4);
```

Purpose
Get the speed ratio settings of all joints on the mobot.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters

`ratio1`	A variable to store the speed ratio of joint 1.
`ratio2`	A variable to store the speed ratio of joint 2.
`ratio3`	A variable to store the speed ratio of joint 3.
`ratio4`	A variable to store the speed ratio of joint 4.

Description
This function is used to retrieve all four joint speed ratio settings of a mobot simultaneously. The speed ratios are as a value from 0 to 1.

Example

See Also
`setJointSpeeds()`, `getJointSpeedRatios()`, `getJointSpeed()`

CMobot::getJointSpeeds()

Synopsis
```
#include <mobot.h>
int CMobot::getJointSpeeds(double &speed1, double &speed2, double &speed3, double &speed4);
```

Purpose
Get the speed settings of all joints on the mobot.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters
- `speed1` The joint speed setting for joint 1.
- `speed2` The joint speed setting for joint 2.
- `speed3` The joint speed setting for joint 3.
- `speed4` The joint speed setting for joint 4.

Description
This function is used to retrieve all four joint speed settings of a mobot simultaneously. The speeds are in degrees per second.

Example

See Also
`setJointSpeeds()`, `getJointSpeedRatios()`, `getJointSpeed()`

CMobot::getJointState()

Synopsis
```
#include <mobot.h>
int CMobot::getJointState(mobotJointId_t id, mobotJointState_t &state);
```

Purpose
Determine whether a motor is moving or not.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters
- `id` The joint number. This is an enumerated type discussed in Section **??** on page **??**.
- `state` An integer variable which will be overwritten with the current state of the motor. This is an enumerated type discussed in Section **??** on page **??**.

Description
This function is used to determine the current state of a motor. Valid states are listed below.

Value	Description
`MOBOT_NEUTRAL`	This value indicates that the joint is not moving and is not actuated. The joint is freely backdrivable.
`MOBOT_FORWARD`	This value indicates that the joint is currently moving forward.
`MOBOT_BACKWARD`	This value indicates that the joint is currently moving backward.
`MOBOT_HOLD`	This value indicates that the joint is currently not moving and is holding its current position. The joint is not currently backdrivable.

Example

See Also
`isMoving()`

CMobot::isConnected()

Synopsis
```
#include <mobot.h>
int CMobot::isConnected();
```

Purpose
Check to see if currently connected to a remote mobot via Bluetooth.

Return Value
The function returns zero if it is not currently connected to a mobot or if an error has occurred, or 1 if the mobot is connected.

Parameters
None.

Description
This function is used to check if the software is currently connected to a mobot.

Example

See Also
`connect()`, `disconnect()`

CMobot::isMoving()

Synopsis
```
#include <mobot.h>
int CMobot::isMoving();
```

Purpose
Check to see if a mobot is currently moving any of its joints.

Return Value
This function returns 0 if none of the joints are being driven or if an error has occurred, or 1 if any joint is being driven. A value of 1 is equivalent to either a joint state of `MOBOT_FORWARD` or `MOBOT_BACKWARD` from `getJointState()`

Parameters
None.

Description
This function is used to determine if a mobot is currently moving any of its joints.

Example

See Also
`getJointState()`

CMobot::motionArch()
CMobot::motionArchNB()

Synopsis

```
#include <mobot.h>
int CMobot::motionArch(double angle);
int CMobot::motionArchNB(double angle);
```

Purpose
Arch the mobot for more ground clearance.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters

`angle` The angle in degrees to arch. This number can range from 0 degrees, which is no arch, to 180 degrees, which is a fully curled up position.

Description

CMobot::motionArch()
This function causes the mobot to Arch up for better ground clearance while rolling.

CMobot::motionArchNB()
This function causes the mobot to Arch up for better ground clearance while rolling.

The non-blocking function, `motionArchNB()`, will return immediately, and the motion will be performed asynchronously.

See Also

CMobot::moveDistance()
CMobot::moveDistanceNB()

Synopsis
```
#include <mobot.h>
int CMobot::moveDistance(double distance, double radius);
int CMobot::moveDistanceNB(double distance, double radius);
```

Purpose
Use the faceplates as wheels to roll forward a certain distance.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters

`distance` The distance to roll the Mobot.
`radius` The radius of the wheels attached to the faceplates of the Mobot.

Description

CMobot::moveDistance()
This function causes each of the faceplates to rotate to roll the mobot forward. The distance to roll the wheels is specified by the argument, `distance`. The `radius` argument specifies the radius of the wheel. Note that the unit of measurement for `radius` must match that of `distance`.

This function has both a blocking and non-blocking version. The blocking version, `moveDistance()`, will block until the mobot motion has completed. The non-blocking version, `moveDistanceNB()`, will return immediately, and the motion will be performed asynchronously.

See Also
`moveBackward()`

CMobot::motionInchwormLeft() CMobot::motionInchwormLeftNB()

Synopsis
```
#include <mobot.h>
int CMobot::motionInchwormLeft(int num);
int CMobot::motionInchwormLeftNB(int num);
```

Purpose
Perform the inch-worm gait to the left.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters

`num`	The number of times to perform the inchworm gait.

Description

CMobot::motionInchwormLeft()
This function causes the mobot to perform a single cycle of the inchworm gait to the left.

CMobot::motionInchwormLeftNB()
This function causes the mobot to perform a single cycle of the inchworm gait to the left.

This is the non-blocking version of the function `CMobot::motionInchwormLeft()`, meaning that the function will return immediately while the motion is performed asynchronously.

See Also
`motionInchwormRight()`

CMobot::motionInchwormRight() CMobot::motionInchwormRightNB()

Synopsis
```
#include <mobot.h>
int CMobot::motionInchwormRight(int num);
int CMobot::motionInchwormRightNB(int num);
```

Purpose
Perform the inch-worm gait to the right.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters

`num` The number of times to perform the inchworm gait.

Description

CMobot::motionInchwormRight()
This function causes the mobot to perform a single cycle of the inchworm gait to the right.

CMobot::motionInchwormRightNB()
This function causes the mobot to perform a single cycle of the inchworm gait to the right.

This function has both a blocking and non-blocking version. The blocking version, `motionInchwormRight()`, will block until the mobot motion has completed. The non-blocking version, `motionInchwormRightNB()`, will return immediately, and the motion will be performed asynchronously.

See Also
`motionInchwormLeft()`

CMobot::motionSkinny()
CMobot::motionSkinnyNB()

Synopsis
```
#include <mobot.h>
int CMobot::motionSkinny(double angle);
int CMobot::motionSkinnyNB(double angle);
```

Purpose
Move the mobot into a skinny profile.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters

`angle` The angle in degrees to move the joints. A value of zero means a completely flat profile, while a value of 90 degrees means a fully skinny profile.

Description

CMobot::motionSkinny()
This function makes the mobot assume a skinny rolling profile.

CMobot::motionSkinnyNB()
This function makes the mobot assume a skinny rolling profile.

This function has both a blocking and non-blocking version. The blocking version, `motionSkinny()`, will block until the mobot motion has completed. The non-blocking version, `motionSkinnyNB()`, will return immediately, and the motion will be performed asynchronously.

See Also

CMobot::motionStand()
CMobot::motionStandNB()

Synopsis

```
#include <mobot.h>
int CMobot::motionStand();
int CMobot::motionStandNB();
```

Purpose
Stand the mobot up on a faceplate.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters
None.

Description

CMobot::motionStand()
This function causes the mobot to motionStand up into the camera platform.

CMobot::motionStandNB()
This function causes the mobot to motionStand up into the camera platform.

This function has both a blocking and non-blocking version. The blocking version, `motionStand()`, will block until the mobot motion has completed. The non-blocking version, `motionStandNB()`, will return immediately, and the motion will be performed asynchronously.

See Also

CMobot::motionTumbleLeft() CMobot::motionTumbleLeftNB()

Synopsis
```
#include <mobot.h>
int CMobot::motionTumbleLeft(int num);
int CMobot::motionTumbleLeftNB(int num);
```

Purpose
Make the mobot tumble end over end.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters

`num`	The number of times to tumble.

Description

CMobot::motionTumbleLeft()
This causes the mobot to tumble end over end. The argument, `num`, indicates the number of times to tumble.

CMobot::motionTumbleLeftNB()
This causes the mobot to tumble end over end. The argument, `num`, indicates the number of times to tumble.

This function has both a blocking and non-blocking version. The blocking version, `motionTumbleLeft()`, will block until the mobot motion has completed. The non-blocking version, `motionTumbleLeftNB()`, will return immediately, and the motion will be performed asynchronously.

See Also

CMobot::motionTumbleRight()
CMobot::motionTumbleRightNB()

Synopsis

```
#include <mobot.h>
int CMobot::motionTumbleRight(int num);
int CMobot::motionTumbleRightNB(int num);
```

Purpose

Make the mobot tumble end over end.

Return Value

The function returns 0 on success and non-zero otherwise.

Parameters

`num`	The number of times to tumble.

Description

CMobot::motionTumbleRight()
This causes the mobot to tumble end over end. The argument, `num`, indicates the number of times to tumble.

CMobot::motionTumbleRightNB()
This causes the mobot to tumble end over end. The argument, `num`, indicates the number of times to tumble.

This function has both a blocking and non-blocking version. The blocking version, `motionTumbleRight()`, will block until the mobot motion has completed. The non-blocking version, `motionTumbleRightNB()`, will return immediately, and the motion will be performed asynchronously.

See Also

CMobot::motionUnstand()
CMobot::motionUnstandNB()

Synopsis

```
#include <mobot.h>
int CMobot::motionUnstand();
int CMobot::motionUnstandNB();
```

Purpose

Move a mobot currently standing on a faceplate back down into a prone position.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters
None.

Description

CMobot::motionUnstand()
This function causes the mobot to move down from the camera platform.

CMobot::motionUnstandNB()
This function causes the mobot to move down from the camera platform.

This function has both a blocking and non-blocking version. The blocking version, `motionUnstand()`, will block until the mobot motion has completed. The non-blocking version, `motionUnstandNB()`, will return immediately, and the motion will be performed asynchronously.

See Also

CMobot::motionWait()

Synopsis
```
#include <mobot.h>
int CMobot::motionWait();
```

Purpose
Wait for a motion to complete execution.

Return Value
The function returns 0 on success and non-zero otherwise.

Description
This function is used to wait for a motion function to fully complete its cycle. The `CMobot` motion functions are those member functions which begin with "motion" as part of their name, such as `motionInchwormLeft()`.
Example

See Also

CMobot::move()
CMobot::moveNB()

Synopsis
```
#include <mobot.h>
int CMobot::move(double angle1, double angle2, double angle3, double angle4);
int CMobot::moveNB(double angle1, double angle2, double angle3, double angle4);
```

Purpose
Move all of the joints of a mobot by specified angles.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters

`angle1`	The amount to move joint 1, expressed in degrees, relative to the current position.
`angle2`	The amount to move joint 2, expressed in degrees, relative to the current position.
`angle3`	The amount to move joint 3, expressed in degrees, relative to the current position.
`angle4`	The amount to move joint 4, expressed in degrees, relative to the current position.

Description

CMobot::move()
This function moves all of the joints of a mobot by the specified number of degrees from their current positions.

CMobot::moveNB()
This function moves all of the joints of a mobot by the specified number of degrees from their current positions.

The function `moveNB()` is the non-blocking version of the `move()` function, which means that the function will return immediately and the physical mobot motion will occur asynchronously. For more information on blocking and non-blocking functions, please refer to Section 13 on page 44.

Example
Please see the demo at Section 9.1.2 on page 30.

See Also

CMobot::moveBackward()
CMobot::moveBackwardNB()

Synopsis
```
#include <mobot.h>
int CMobot::moveBackward(double angle);
int CMobot::moveBackwardNB(double angle);
```

Purpose
Use the faceplates as wheels to roll backward.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters

`angle`	The angle to turn the wheels, specified in degrees.

Description

CMobot::moveBackward()
This function causes each of the faceplates to rotate to roll the mobot backward. The amount to roll the wheels is specified by the argument, `angle`.

CMobot::moveBackwardNB()
This function causes each of the faceplates to rotate to roll the mobot backward. The amount to roll the wheels is specified by the argument, `angle`.

This function has both a blocking and non-blocking version. The blocking version, `moveBackward()`, will block until the mobot motion has completed. The non-blocking version, `moveBackwardNB()`, will return immediately, and the motion will be performed asynchronously.

See Also
`moveForward()`

CMobot::moveForward()
CMobot::moveForwardNB()

Synopsis
```
#include <mobot.h>
int CMobot::moveForward(double angle);
int CMobot::moveForwardNB(double angle);
```

Purpose
Use the faceplates as wheels to roll forward.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters

`angle`	The angle to turn the wheels, specified in degrees.

Description

CMobot::moveForward()
This function causes each of the faceplates to rotate to roll the mobot forward. The amount to roll the wheels is specified by the argument, `angle`.

CMobot::moveForwardNB()
This function causes each of the faceplates to rotate to roll the mobot forward. The amount to roll the wheels is specified by the argument, `angle`.

This function has both a blocking and non-blocking version. The blocking version, `moveForward()`, will block until the mobot motion has completed. The non-blocking version, `moveForwardNB()`, will return immediately, and the motion will be performed asynchronously.

See Also
`moveBackward()`

CMobot::moveJoint()
CMobot::moveJointNB()

Synopsis
```
#include <mobot.h>
int CMobot::moveJoint(mobotJointId_t id, double angle);
int CMobot::moveJointNB(mobotJointId_t id, double angle);
```

Purpose
Move a joint on the mobot by a specified angle with respect to the current position.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters

`id`	The joint number to move.
`angle`	The angle in degrees to move the motor relative to its current position.

Description

CMobot::moveJoint()
This function commands the motor to move by an angle relative to the joint's current position at the joints current speed setting. The current motor speed may be set with the `setJointSpeed()` member function. Please note that if the motor speed is set to zero, the motor will not move after calling the `moveJoint()` function.

CMobot::moveJointNB()
This function commands the motor to move by an angle relative to the joint's current position at the joints current speed setting. The current motor speed may be set with the `setJointSpeed()` member function. Please note that if the motor speed is set to zero, the motor will not move after calling the `moveJointNB()` function.

The function `moveJointNB()` is the non-blocking version of the `moveJoint()` function, which means that the function will return immediately and the physical mobot motion will occur asynchronously. For more details on blocking and non-blocking functions, please refer to Section 13 on page 44.

Example
Please see the example in Section 9.1.2 on page 30.

See Also
`connectWithBluetoothAddress()`

CMobot::moveJointTo()
CMobot::moveJointToNB()

Synopsis
```
#include <mobot.h>
int CMobot::moveJointTo(mobotJointId_t id, double angle);
int CMobot::moveJointToNB(mobotJointId_t id, double angle);
```

Purpose
Move a joint on the mobot to an absolute position.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters

`id`	The joint number to wait for.
`angle`	The absolute angle in degrees to move the motor to.

Description

CMobot::moveJointTo()
This function commands the motor to move to a position specified in radians at the current motor's speed. The current motor speed may be set with the `setJointSpeed()` member function. Please note that if the motor speed is set to zero, the motor will not move after calling the `moveJointTo()` function.

CMobot::moveJointToNB()
This function commands the motor to move to a position specified in radians at the current motor's speed. The current motor speed may be set with the `setJointSpeed()` member function. Please note that if the motor speed is set to zero, the motor will not move after calling the `moveJointToNB()` function.

The function `moveJointToNB()` is the non-blocking version of the `moveJointTo()` function, which means that the function will return immediately and the physical mobot motion will occur asynchronously. For more details on blocking and non-blocking functions, please refer to Section 13 on page 44.

Example
Please see the example in Section 9.1.2 on page 30.

See Also
`connectWithBluetoothAddress()`

CMobot::moveJointWait()

Synopsis
```
#include <mobot.h>
int CMobot::moveJointWait(mobotJointId_t id);
```

Purpose
Wait for a joint to stop moving.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters

`id`	The joint number to wait for.

Description
This function is used to wait for a joint motion to finish. Functions such as `moveNB()` and `moveJointNB()` do not wait for a joint to finish moving before continuing to allow multiple joints to move at the same time. The `moveJointWait()` function is used to wait for a mobotic joint motion to complete.

Please note that if this function is called after a motor has been commanded to turn indefinitely, this function may never return and your program may hang.

Example
Please see the example in Section 9.1.2 on page 30.

See Also
`moveWait()`

CMobot::moveTo()
CMobot::moveToNB()

Synopsis

```
#include <mobot.h>
int CMobot::moveTo(double angle1, double angle2, double angle3, double angle4);
int CMobot::moveToNB(double angle1, double angle2, double angle3, double angle4);
```

Purpose

Move all of the joints of a mobot to the specified positions.

Return Value

The function returns 0 on success and non-zero otherwise.

Parameters

`angle1`	The absolute position to move joint 1, expressed in degrees.
`angle2`	The absolute position to move joint 2, expressed in degrees.
`angle3`	The absolute position to move joint 3, expressed in degrees.
`angle4`	The absolute position to move joint 4, expressed in degrees.

Description

CMobot::moveTo()
This function moves all of the joints of a mobot to the specified absolute positions.

CMobot::moveToNB()
This function moves all of the joints of a mobot to the specified absolute positions.

The function `moveToNB()` is the non-blocking version of the `moveTo()` function, which means that the function will return immediately and the physical mobot motion will occur asynchronously. For more details on blocking and non-blocking functions, please refer to Section 13 on page 44.

Example

Please see the demo at Section 9.1.2 on page 30.

See Also

CMobot::moveToZero()
CMobot::moveToZeroNB()

Synopsis

```
#include <mobot.h>
int CMobot::moveToZero();
int CMobot::moveToZeroNB();
```

Purpose

Move all of the joints of a mobot to their zero position.

Return Value

The function returns 0 on success and non-zero otherwise.

Parameters
None.

Description

CMobot::moveToZero()
This function moves all of the joints of a mobot to their zero position.

CMobot::moveToZeroNB()
This function moves all of the joints of a mobot to their zero position.

The function `moveToZeroNB()` is the non-blocking version of the `moveToZero()` function, which means that the function will return immediately and the physical mobot motion will occur asynchronously. For more details on blocking and non-blocking functions, please refer to Section 13 on page 44.

Example
Please see the demo at Section 9.1.2 on page 30.

See Also

CMobot::moveWait()

Synopsis
```
#include <mobot.h>
int CMobot::moveWait();
```

Purpose
Wait for all joints to stop moving.

Return Value
The function returns 0 on success and non-zero otherwise.

Description
This function is used to wait for all joint motions to finish. Functions such as `move()` and `moveTo()` do not wait for a joint to finish moving before continuing to allow multiple joints to move at the same time. The `moveWait()` function is used to wait for mobotic motions to complete.

Please note that if this function is called after a motor has been commanded to turn indefinitely, this function may never return and your program may hang.

Example
See the sample program in Section 9.1.2 on page 30.
See Also
`moveWait(), moveJointWait()`

CMobot::recordAngle()

Synopsis
```
#include <mobot.h>
int CMobot::recordAngle(
    mobotJointId_t id,
    double time[],
```

```
    double angle[],
    int num,
    double seconds,
    int shiftData= 1);
```

Purpose
Record joint angle data for a joint for a set amount of time at a specified time interval.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters

`id`	The joint number. This is an enumerated type discussed in Section **??** on page **??**.
`time`	An array which will store time values for each of the angle readings.
`angle`	An array which will store the angle values for each time.
`num`	The size of the arrays.
`seconds`	The number of seconds between angle readings. The minimum value allowed for this variable is 0.05.
`threshold`	(optional) This argument indicates whether or not to align the first detected motion to the y-axis.

Description
This function is used to accurately record the motion of a Mobot joint at a relatively fast rate. The function will fill the `time` and `angle` arrays with data at the rate specified by `seconds`. If the communication speed cannot maintain the requested rate, (if `msecs` is too low, in other words), the function will collect data as fast as possible. The minimum value for `seconds` is 0.05, but the actual minimum time will depend on other factors, such as communication noise and distance to the mobot.

The length of time to collect the data can be calculated by the formula

$$\text{Total Time} = (\text{num} \times \text{seconds})$$

This function is a non-blocking function. After calling this function, a call to `recordWait()` should be performed to ensure that the data has been fully collected.

Example

See Also
`recordAngles()`, `recordWait()`

CMobot::recordAngleBegin()

Synopsis
```
#include <mobot.h>
int CMobot::recordAngleBegin(
    mobotJointId_t id,
    mobotRecordData_t &time,
    mobotRecordData_t &angle,
    double seconds,
    int shiftData = 1);
```

Purpose
Record joint angle data for a joint for a set amount of time at a specified time interval.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters

`id`	The joint number. This is an enumerated type discussed in Section **??** on page **??**.
`time`	A variable which will store time values for each of the angle readings.
`angle`	A variable which will store the angle values for each time.
`seconds`	The number of seconds between angle readings. The minimum value allowed for this variable is 0.05.
`shiftData`	(optional) This argument indicates whether or not to align the first detected motion to the y-axis.

Description
This function is used in conjunction with the `recordAngleEnd()` function to accurately record the motion of a Mobot joint at a relatively fast rate.

The function will allocate memory and fill the `time` and `angle` variables with data at the rate specified by `seconds`.

If the communication speed cannot maintain the requested rate, (if `seconds` is too low, in other words), the function will collect data as fast as possible. The minimum value for `seconds` is 0.05, but the actual minimum time will depend on other factors, such as communication noise and distance to the mobot.

When the `recordAngleBegin()` function is called, the Mobot library will immediately begin recording angle data. The `recordAngleEnd()` function must be called to halt the recording process.

Example

See Also
`recordAngleEnd()`

CMobot::recordAngleEnd()

Synopsis
```
#include <mobot.h>
int CMobot::recordAngleEnd(mobotJointId_t id, int &num);
```

Purpose
End a joint recording process that is currently running.

Return Value
This function returns zero on success and -1 on failure.

Parameters

`id`	The joint number. This is an enumerated type discussed in Section **??** on page **??**.
`num`	An integer variable that will be set to the number of recorded elements.

Description
This function is used in conjunction with the `recordAngleBegin()` function. This function stops the recording process and returns the number of valid data points allocated for the arrays.

Example

See Also
`recordAngleBegin()`

CMobot::recordAngles()

Synopsis

```
#include <mobot.h>
int CMobot::recordAngles(double time[], double angle1[], double angle2[],
                         double angle3[], double angle4[], int num,
                         double seconds, int shiftData = 1);
```

Purpose

Record joint angle data for all joint for a set amount of time at a specified time interval.

Return Value

The function returns 0 on success and non-zero otherwise.

Parameters

`time`	An array which will store time values for each of the angle readings.
`angle1`	An array which will store the angle values for joint 1.
`angle2`	An array which will store the angle values for joint 2.
`angle3`	An array which will store the angle values for joint 3.
`angle4`	An array which will store the angle values for joint 4.
`num`	The number of elements of the arrays.
`seconds`	The number of seconds between angle readings.
`threshold`	(optional) This argument indicates whether or not to align the first detected motion to the y-axis.

Description

This function is used to accurately record the motion of a Mobot joint at a relatively fast rate. The function will fill the `time`, `angle1`, `angle2`, `angle3`, and `angle4` arrays with data at the rate specified by `seconds`. A typical value for `seconds` is 0.1, or polling 10 times a second. If the communication speed cannot maintain the requested rate, (if `msecs` is too low, in other words), the function will collect data as fast as possible. The lowest allowable rate is 0.05, however there is no guarantee that data will actually be collected at that rate, due to communication noise, distance to the module, etc.

The length of time to collect the data can be calculated by the formula

$$\text{Total Time} = (\text{num} \times \text{seconds})$$

This function is a non-blocking function. After calling this function, a call to `recordWait()` should be performed to ensure that the data has been fully collected.

Example

See Also

`recordAngle()`, `recordWait()`

CMobot::recordAnglesBegin()

Synopsis

```
#include <mobot.h>
int CMobot::recordAnglesBegin(double* &time,
                              double* &angle1,
                              double* &angle2,
                              double* &angle3,
```

```
                                double* &angle4,
                                double seconds,
                                int shiftData = 1);
```

Purpose
Record joint angle data for all joints for a set amount of time at a specified time interval.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters

`time`	A pointer to an array which will store time values for each of the angle readings.
`angle1`	A pointer to an array which will store the angle values for each time.
`angle2`	...
`angle3`	...
`angle4`	...
`seconds`	The number of seconds between angle readings. The minimum value allowed for this variable is 0.05.
`threshold`	(optional) This argument indicates whether or not to align the first detected motion to the y-axis.

Description
This function is used in conjunction with the `recordAnglesEnd()` function to accurately record the motion of a Mobot joint at a relatively fast rate.

The function will allocate memory and fill the `time` and `angle` arrays with data at the rate specified by `seconds`. The `time` and `angle` arrays should be freed by the user to prevent memory leaks.

If the communication speed cannot maintain the requested rate, (if `seconds` is too low, in other words), the function will collect data as fast as possible. The minimum value for `seconds` is 0.05, but the actual minimum time will depend on other factors, such as communication noise and distance to the mobot.

When the `recordAnglesBegin()` function is called, the Mobot library will immediately begin recording angle data. The `recordAngleEnd()` function must be called to halt the recording process.

Example

See Also
`recordAngleEnd()`

CMobot::recordAnglesEnd()

Synopsis
```
#include <mobot.h>
int CMobot::recordAnglesEnd(int &num);
```

Purpose
End a joint recording process that is currently running.

Return Value
This function returns 0 on success or -1 on failure.

Parameters

`num`	An integer variable that will be set to the number of recorded elements.

Description
This function is used in conjunction with the `recordAngleBegin()` function. This function stops the recording process and returns the number of valid data points allocated for the arrays.

Example

See Also
`recordAngleBegin()`

CMobot::recordDistanceBegin()

Synopsis
```
#include <mobot.h>
int CMobot::recordDistanceBegin(
    mobotJointId_t id,
    mobotRecordData_t &time,
    mobotRecordData_t &distance,
    double radius,
    double seconds,
    int shiftData = 1);
```

Purpose
Record joint angle data for a joint for a set amount of time at a specified time interval.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters

`id`	The joint number. This is an enumerated type discussed in Section **??** on page **??**.
`time`	A variable which will store time values for each of the angle readings.
`distance`	A variable which will store the distance values for each time.
`radius`	The radius of the wheels.
`seconds`	The number of seconds between angle readings. The minimum value allowed for this variable is 0.05.
`threshold`	(optional) This argument indicates whether or not to align the first detected motion to the y-axis.

Description
This function is used in conjunction with the `recordDistanceEnd()` function to accurately record the distance traveled of a Mobot wheel at a relatively fast rate.

The function will allocate memory and fill the `time` and `distance` variables with data at the rate specified by `seconds`.

If the communication speed cannot maintain the requested rate, (if `seconds` is too low, in other words), the function will collect data as fast as possible. The minimum value for `seconds` is 0.05, but the actual minimum time will depend on other factors, such as communication noise and distance to the mobot.

When the `recordDistanceBegin()` function is called, the Mobot library will immediately begin recording data. The `recordDistanceEnd()` function must be called to halt the recording process.
Example

See Also
`recordDistanceEnd()`

CMobot::recordDistanceEnd()

Synopsis
```
#include <mobot.h>
int CMobot::recordDistanceEnd(mobotJointId_t id, int &num);
```

Purpose
End a joint recording process that is currently running.

Return Value
This function returns zero on success and -1 on failure.

Parameters

`id`	The joint number. This is an enumerated type discussed in Section **??** on page **??**.
`num`	An integer variable that will be set to the number of recorded elements.

Description
This function is used in conjunction with the `recordDistanceBegin()` function. This function stops the recording process and returns the number of valid data points allocated for the arrays.

Example

See Also
`recordDistanceBegin()`

CMobot::recordWait()

Synopsis
```
#include <mobot.h>
int CMobot::recordWait();
```

Purpose
Wait for a joint recording operation to finish.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters
None

Description
This function is used in conjunction with the `recordAngle()` function and/or the `recordAngles()` function. The `recordAngle()` and `recordAngles()` functions both initiate a recording operation that runs in the background for a certain amount of time. The `recordWait()` function is used to pause the main program until the recording operation has finished.

Example

See Also
`recordAngle(), recordAngles()`

CMobot::resetToZero()

CMobot::resetToZeroNB()

Synopsis
```
#include <mobot.h>
int CMobot::resetToZero();
int CMobot::resetToZeroNB();
```

Purpose
Move all of the joints of a mobot to their zero position.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters
None.

Description

CMobot::resetToZero()
This function moves all of the joints of a mobot to their zero position. If the joint is currently multiple rotations away from its zero position, this function will reset the joint revolutions and move the joint directly to the zero position.

CMobot::resetToZeroNB()
This function moves all of the joints of a mobot to their zero position.

The function `resetToZeroNB()` is the non-blocking version of the `resetToZero()` function, which means that the function will return immediately and the physical mobot motion will occur asynchronously. For more details on blocking and non-blocking functions, please refer to Section 13 on page 44.

Example
Please see the demo at Section 9.1.2 on page 30.

See Also

CMobot::setExitState()

Synopsis
```
#include <mobot.h>
int CMobot::setExitState(mobotJointState_t state);
```

Purpose
Sets the behaviour of joints when the program exits.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters

`state` The desired behaviour of the robot joints on program exit. By default, the behavior is set to `MOBOT_NEUTRAL`, which relaxes all of the joints. Alternatively, the state may be set to `MOBOT_HOLD` to hold the joints on program exit.

Description
This function is used to change the behaviour of the joints when the program exits. By default, when a Mobot program exits, the all joints of the connected Mobots go to the `MOBOT_NEUTRAL` state. Using this function, the exit state may be changed to `MOBOT_HOLD` or `MOBOT_NEUTRAL`. All other joint states are invalid when used with this function.
Example

See Also

CMobot::setJointMovementStateNB()

Synopsis
```
#include <mobot.h>
int CMobot::setJointMovementStateNB(mobotJointId_t id, mobotJointState_t dir1);
```

Purpose
Move a joint of a mobot continuously in the specified directions.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters

`id`	The joint number to move.
`dir`	This parameter specifies the direction the joint should move.

Description
The `dir` parameter specifies the direction the joint should move. The types are enumerated in `mobot.h` and have the following values:

Value	Description
`MOBOT_NEUTRAL`	This value indicates that the joint is not moving and is not actuated. The joint is freely backdrivable.
`MOBOT_FORWARD`	This value indicates that the joint is currently moving forward.
`MOBOT_BACKWARD`	This value indicates that the joint is currently moving backward.
`MOBOT_HOLD`	This value indicates that the joint is currently not moving and is holding its current position. The joint is not currently backdrivable.

More documentation about these types may be found at Section **??** on page **??**.

This function causes joints of a mobot to begin moving at the previously set speed. The joints will continue moving until the joint hits a joint limit, or the joint is stopped by setting the speed to zero. This function is a non-blocking function.

Example

See Also

CMobot::setJointMovementStateTime()
CMobot::setJointMovementStateTimeNB()

Synopsis

```
#include <mobot.h>
int CMobot::setJointMovementStateTime(mobotJointId_t id, mobotJointState_t dir1, double seconds);
int CMobot::setJointMovementStateTimeNB(mobotJointId_t id, mobotJointState_t dir1, double seconds);
```

Purpose
Move a joint of a mobot continuously in the specified directions for a certain amount of time.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters

`id`	The joint number to move.
`dir`	This parameter specifies the direction the joint should move.
`seconds`	The number of seconds to rotate the joint.

Description
The `dir` parameter specifies the direction the joint should move. The types are enumerated in `mobot.h` and have the following values:

Value	Description
`MOBOT_NEUTRAL`	This value indicates that the joint is not moving and is not actuated. The joint is freely backdrivable.
`MOBOT_FORWARD`	This value indicates that the joint is currently moving forward.
`MOBOT_BACKWARD`	This value indicates that the joint is currently moving backward.
`MOBOT_HOLD`	This value indicates that the joint is currently not moving and is holding its current position. The joint is not currently backdrivable.

More documentation about these types may be found at Section **??** on page **??**.

This function causes joints of a mobot to begin moving at the previously set speed. The joints will continue moving until the joint hits a joint limit, the joint is stopped by setting the speed to zero, or the specified amount of time has expired.

The function `setJointMovementStateTimeNB()` is the non-blocking version, and the timing of the timed motion will occur in the background while the main program continues.

Example

See Also

CMobot::setJointSafetyAngle()

Synopsis
```
#include <mobot.h>
int CMobot::setJointSafetyAngle(double degrees);
```

Purpose
Set the current angle safety limit of the Mobot.

Return Value
The function returns 0 on success and -1 on failure.

Parameters
The requested joint safety angle limit for the Mobot.

Description
The Mobot is equipped with a safety feature to protect itself and its surrounding environment. When a motor deviates by a certain amount from its expected value, the Mobot will shut off all power to the motor, in case it has hit an obstacle, or for any other reason. The amount of deviation required to trigger the safety protocol is the joint safety angle which can be set using this function. The default setting is 10 degrees. Higher values indicate "less safe" behavior of the Mobot because the Mobot will not engage safety protocols until the joint has deviated by a greater amount. Values greater than 90 degrees effectively disengage the Mobot's safety protocols altogether, and this function should be used with care.
Example

See Also
`getJointSafetyAngle()`, `getJointSafetyAngleTimeout()`, `setJointSafetyAngleTimeout()`

CMobot::setJointSafetyAngleTimeout()

Synopsis
```
#include <mobot.h>
int CMobot::setJointSafetyAngleTimeout(double seconds);
```

Purpose
Set the current angle safety limit timeout of the Mobot.

Return Value
The function returns 0 on success and -1 on failure.

Parameters
A variable which will be overwritten with the safety angle limit timeout in seconds.

Description
The Mobot is equipped with a safety feature to protect itself and its surrounding environment. When a motor deviates by a certain amount from its expected value, the Mobot will shut off all power to the motor after a certain period of time, in case it has hit an obstacle, or for any other reason. The period of time that the mobot waits before shutting the motor off is the joint safety angle timeout, which can be set with this function. The default value for the timeout is 500 milliseconds, or 0.5 seconds.
Example

See Also
`getJointSafetyAngle()`, `setJointSafetyAngle()`, `getJointSafetyAngleTimeout()`

CMobot::setJointSpeed()

Synopsis
```
#include <mobot.h>
int CMobot::setJointSpeed(mobotJointId_t id, double speed);
```

Purpose
Set the speed of a joint on the mobot.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters

`id`	The joint number to pose.
`speed`	An variable of type `double` for the requested average angular speed in degrees per second.

Description
This function is used to set the angular speed of a joint of a mobot. The maximum possible angular speed for a particular joint may be obtained by using the function `getJointMaxSpeed()`.
Example

See Also

CMobot::setJointSpeedRatio()

Synopsis
```
#include <mobot.h>
int CMobot::setJointSpeedRatio(mobotJointId_t id, double ratio);
```

Purpose
Set the speed ratio settings of a joint on the mobot.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters

`id`	Set the speed ratio setting of this joint. This is an enumerated type discussed in Section **??** on page **??**.
`ratio`	A variable of type double with a value from 0 to 1.

Description
This function is used to set the speed ratio setting of a joint. The speed ratio setting of a joint is the percentage of the maximum joint speed, and the value ranges from 0 to 1. In other words, if the ratio is set to 0.5, the joint will turn at 50% of its maximum angular velocity while moving continuously or moving to a new goal position.

Example

See Also
`setJointSpeeds()`, `setJointSpeedRatio()`, `getJointSpeed()`

CMobot::setJointSpeedRatios()

Synopsis
```
#include <mobot.h>
int CMobot::setJointSpeedRatios(double ratio1, double ratio2, double ratio3, double ratio4);
```

Purpose
Set the speed ratio settings of all joints on the mobot.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters

`ratio1` The speed ratio setting for the first joint.
`ratio2` The speed ratio setting for the second joint.
`ratio3` The speed ratio setting for the third joint.
`ratio4` The speed ratio setting for the fourth joint.

Description
This function is used to simultaneously set the angular speed ratio settings of all four joints of a mobot. The speed ratio is a percentage of the maximum speed of a joint, expressed in a value from 0 to 1.

Example

See Also
`getJointSpeeds()`, `setJointSpeed()`, `getJointSpeed()`

CMobot::setJointSpeeds()

Synopsis
```
#include <mobot.h>
int CMobot::setJointSpeeds(double speed1, double speed2, double speed3, double speed4);
```

Purpose
Set the speed settings of all joints on the mobot.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters
`speed1` The speed for the first joint, in degrees per second.
`speed2` The speed for the second joint, in degrees per second.
`speed3` The speed for the third joint, in degrees per second.
`speed4` The speed for the fourth joint, in degrees per second.

Description
This function is used to simultaneously set the angular speed settings of all four joints of a mobot.

Example

See Also
`getJointSpeeds()`, `setJointSpeed()`, `getJointSpeed()`

CMobot::setMovementStateNB()

Synopsis
```
#include <mobot.h>
int CMobot::setMovementStateNB(
  mobotJointState_t dir1,
  mobotJointState_t dir2,
  mobotJointState_t dir3,
  mobotJointState_t dir4);
```

Purpose
Move the joints of a mobot continuously in the specified directions.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters
Each parameter specifies the direction the joint should move. The types are enumerated in `mobot.h` and have the following values:

Value	Description
`MOBOT_NEUTRAL`	This value indicates that the joint is not moving and is not actuated. The joint is freely backdrivable.
`MOBOT_FORWARD`	This value indicates that the joint is currently moving forward.
`MOBOT_BACKWARD`	This value indicates that the joint is currently moving backward.
`MOBOT_HOLD`	This value indicates that the joint is currently not moving and is holding its current position. The joint is not currently backdrivable.

More documentation about these types may be found at Section **??** on page **??**.

Description
This function causes joints of a mobot to begin moving at the previously set speed. The joints will continue moving until the joint hits a joint limit, or the joint is stopped by setting the speed to zero. This function is a non-blocking function.

Example

See Also

CMobot::setMovementStateTime() CMobot::setMovementStateTimeNB()

Synopsis
```
#include <mobot.h>
int CMobot::setMovementStateTime( mobotJointState_t dir1,
                                  mobotJointState_t dir2,
                                  mobotJointState_t dir3,
                                  mobotJointState_t dir4,
                                  double seconds);
int CMobot::setMovementStateTimeNB( mobotJointState_t dir1,
                                  mobotJointState_t dir2,
                                  mobotJointState_t dir3,
                                  mobotJointState_t dir4,
                                  double seconds);
```

Purpose
Move the joints of a mobot continuously in the specified directions for some amount of time.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters
Each direction parameter specifies the direction the joint should move. The types are enumerated in `mobot.h` and have the following values:

Value	Description
MOBOT_NEUTRAL	This value indicates that the joint is not moving and is not actuated. The joint is freely backdrivable.
MOBOT_FORWARD	This value indicates that the joint is currently moving forward.
MOBOT_BACKWARD	This value indicates that the joint is currently moving backward.
MOBOT_HOLD	This value indicates that the joint is currently not moving and is holding its current position. The joint is not currently backdrivable.

The seconds parameter is the time to perform the movement, in seconds.

Description
This function causes joints of a mobot to begin moving. The joints will continue moving until the joint hits a joint limit, or the time specified in the seconds parameter is reached. The setMovementStateTime() function will block until the motion is completed, whereas the setMovementStateTimeNB() function will not block.
Example

See Also

CMobot::setTwoWheelRobotSpeed()

Synopsis
```
#include <mobot.h>
int CMobot::setTwoWheelRobotSpeed(double speed, double radius);
```

Purpose
Roll the mobot at a certain speed in a straight line.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters

speed	The speed at which to roll the mobot. The units used will be the units specified in the unit parameter.
radius	The radius of the wheels attached to the mobot. The units of the parameter should match the units provided in the unit parameter.

speed	radius
cm/s	cm
m/s	m
inch/s	inch
foot/s	foot

Description
This function is used to make a two wheeled mobot roll at a certain speed. The desired speed and radius of the wheels is provided and the function will rotate the wheels at the appropriate rate in order to achieve the desired speed.
Example

See Also

CMobot::stopAllJoints()
CMobot::stopOneJoint()
CMobot::stopTwoJoints()
CMobot::stopThreeJoints()

Synopsis

```
#include <mobot.h>
int CMobot::stopAllJoints();
int CMobot::stopOneJoint(mobotJointId_t id);
int CMobot::stopTwoJoints(mobotJointId_t id1, mobotJointId_t id2);
int CMobot::stopThreeJoints(
    mobotJointId_t id1,
    mobotJointId_t id2,
    mobotJointId_t id3);
```

Purpose
These functions stop joints on a mobot.

Return Value
The function returns 0 on success and non-zero otherwise.

Description
These functions are used to stop joints on a mobot. A stopped joint will immediately cease any and all actuation and go limp.
Example

See Also
`setJointSpeed()`, `setJointSpeeds()`

CMobot::turnLeft()
CMobot::turnLeftNB()

Synopsis

```
#include <mobot.h>
int CMobot::turnLeft(double angle);
int CMobot::turnLeftNB(double angle);
```

Purpose
Rotate the mobot using the faceplates as wheels.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters

`angle` The angle in degrees to turn the wheels. The wheels will turn in opposite directions by the amount specified by this argument in order to turn the mobot to the left.

Description

CMobot::turnLeft()
This function causes the mobot to rotate the faceplates in opposite directions to cause the mobot to rotate counter-clockwise.

CMobot::turnLeftNB()
This function causes the mobot to rotate the faceplates in opposite directions to cause the mobot to rotate counter-clockwise.

This function has both a blocking and non-blocking version. The blocking version, `turnLeft()`, will block until the mobot motion has completed. The non-blocking version, `turnLeftNB()`, will return immediately, and the motion will be performed asynchronously.

See Also
`turnRight()`

CMobot::turnRight()
CMobot::turnRightNB()

Synopsis
```
#include <mobot.h>
int CMobot::turnRight(double angle);
int CMobot::turnRightNB(double angle);
```

Purpose
Rotate the mobot using the faceplates as wheels.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters

`angle` The angle in degrees to turn the wheels. The wheels will turn in opposite directions by the amount specified by this argument in order to turn the mobot to the right.

Description

CMobot::turnRight()
This function causes the mobot to rotate the faceplates in opposite directions to cause the mobot to rotate clockwise.

CMobot::turnRightNB()
This function causes the mobot to rotate the faceplates in opposite directions to cause the mobot to rotate clockwise.

This function has both a blocking and non-blocking version. The blocking version, `turnRight()`, will block until the mobot motion has completed. The non-blocking version, `turnRightNB()`, will return immediately, and the motion will be performed asynchronously.

See Also
`turnRight()`

B CMobotGroup API

The `CMobotGroup` class is used to control multiple modules simultaneously. The member functions of the `CMobotGroup` class closely mimic those of the `CMobot` group. The main difference is that the member functions of the `CMobot` class affect a single mobot, whereas the member functions of the `CMobotGroup` class move and affect a group of many mobots.

Table 3: CMobotGroup Member Functions.

Function	Description
`CMobotGroup()`	The CMobotGroup constructor function. This function is called automatically and should not be called explicitly.
`~CMobotGroup()`	The CMobotGroup destructor function. This function is called automatically and should not be called explicitly.
`addRobot()`	Add a mobot to be a member of the mobot group.
`move()`	Move all four joints of the mobots by specified angles.
`moveDistance()`	Move each wheeled mobot in the group a certain distance.
`moveDistanceNB()`	Identical to `moveDistance()` but non-blocking.
`moveBackward()`	Roll on the faceplates toward the backward direction.
`moveBackwardNB()`	Identical to `moveBackward()` but non-blocking.
`moveForward()`	Roll on the faceplates forwards.
`moveForwardNB()`	Identical to `moveForward()` but non-blocking.
`moveNB()`	Identical to `move()` but non-blocking.
`moveJoint()`	Move a motor from its current position by an angle.
`moveJointNB()`	Identical to `moveJoint()` but non-blocking.
`moveJointTo()`	Set the desired motor position for a joint.
`moveJointToNB()`	Identical to `moveJointTo()` but non-blocking.
`moveJointWait()`	Wait until the specified motor has stopped moving.
`moveTo()`	Move all four joints of the mobots to specified absolute angles.
`moveToNB()`	Identical to `moveTo()` but non-blocking.
`moveToZero()`	Instructs all motors to go to their zero positions.
`moveToZeroNB()`	Identical to `moveToZero()` but non-blocking.
`moveWait()`	Wait until all motors have stopped moving.
`setExitState()`	Set the joint behavior on exit for all mobots in the group.
`setJointMovementStateNB()`	Move a single joint on all mobots continuously.
`setJointMovementStateTime()`	Move a single joint on all mobots continuously for a specific amount of time.
`setJointMovementStateTimeNB()`	Move a single joint on all mobots continuously for a specific amount of time.
`setJointSpeed()`	Set a motor's speed setting in radians per second.
`setJointSpeeds()`	Set all motor speeds in radians per second.
`setJointSpeedRatio()`	Set a joints speed setting to a fraction of its maximum speed, a value between 0 and 1.
`setJointSpeedRatios()`	Set all joint speed settings to a fraction of its maximum speed, expressed as a value from 0 to 1.
`setMovementStateNB()`	Move joints continuously. Joints will move untill stopped.
`setMovementStateTime()`	Move joints continuously for a certain amount of time.
`setMovementStateTimeNB()`	Move joints continuously for a certain amount of time.
`setTwoWheelRobotSpeed()`	Move the mobots at a constant forward velocity.
`stop()`	Stop all currently executing motions of the mobot.
`turnLeft()`	Rotate the mobots counterclockwise.
`turnLeftNB()`	Identical to `turnLeft()` but non-blocking.
`turnRight()`	Rotate the mobots clockwise.
`turnRightNB()`	Identical to `turnRight()` but non-blocking.

Table 4: CMobotGroup Member Functions for Compound Motions.

Compound Motions	These are convenience functions of commonly used compound motions.
`motionArch()`	Move each mobot in the group into an arched configuration.
`motionArchNB()`	Identical to `motionArch()` but non-blocking.
`motionInchwormLeft()`	Inchworm motion towards the left.
`motionInchwormLeftNB()`	Identical to `motionInchwormLeft()` but non-blocking.
`motionInchwormRight()`	Inchworm motion towards the right.
`motionInchwormRightNB()`	Identical to `motionInchwormRight()` but non-blocking.
`motionSkinny()`	Move the mobots into a skinny configuration.
`motionSkinnyNB()`	Identical to `motionSkinnyNB()` but non-blocking.
`motionStand()`	Stand the mobots up on its end.
`motionStandNB()`	Identical to `motionStandNB()` but non-blocking.
`motionTumbleLeft()`	Tumble the mobots end over end.
`motionTumbleLeftNB()`	Identical to `motionTumbleLeftNB()` but non-blocking.
`motionTumbleRight()`	Tumble the mobots end over end.
`motionTumbleRightNB()`	Identical to `motionTumbleRightNB()` but non-blocking.
`motionUnstand()`	Move each mobot in the group currently standing on its end down into zero position.
`motionUnstandNB()`	Identical to `motionUnstand()` but non-blocking.
`motionWait()`	Wait for preprogrammed mobotic motions to complete.

CMobotGroup::addRobot()

Synopsis
```
#include <mobot.h>
int CMobotGroup::addRobot(CMobot &mobot);
```

Purpose
Add a mobot to a mobot group.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters
A mobot handle attached to the mobot to add to the group.

Description
This function is used to add a mobot to a mobot group.
Example

See Also

CMobotGroup::driveJointTo()
CMobotGroup::driveJointToNB()

Synopsis
```
#include <mobot.h>
int CMobotGroup::driveJointTo(moboJointId_t id, double angle);
int CMobotGroup::driveJointToNB(moboJointId_t id, double angle);
```

Purpose
Move all of the joints of a mobot to the specified positions.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters

`id`	The id of the joint to move.
`angle`	The absolute position to move the joint, expressed in degrees.

Description

Note that this function is similar to the `moveJointTo()` family of functions, except that this variant ignores the speed settings of the robot joints. This function causes the robot to move its joints towards its goal using a Proportional-Integral-Derivative (PID) controller.

CMobotGroup::driveJointTo()
This function moves all of the joints of a mobot to the specified absolute positions.

CMobotGroup::driveJointToNB()
This function moves all of the joints of a mobot to the specified absolute positions.

The function `driveJointToNB()` is the non-blocking version of the `driveJointTo()` function, which means that the function will return immediately and the physical mobot motion will occur asynchronously. For more details on blocking and non-blocking functions, please refer to Section 13 on page 44.

Example

See Also

CMobotGroup::driveTo()
CMobotGroup::driveToNB()

Synopsis

```
#include <mobot.h>
int CMobotGroup::driveTo(double angle1, double angle2, double angle3, double angle4);
int CMobotGroup::driveToNB(double angle1, double angle2, double angle3, double angle4);
```

Purpose

Move all of the joints of a mobot to the specified positions.

Return Value

The function returns 0 on success and non-zero otherwise.

Parameters

`angle1`	The absolute position to move joint 1, expressed in degrees.
`angle2`	The absolute position to move joint 2, expressed in degrees.
`angle3`	The absolute position to move joint 3, expressed in degrees.
`angle4`	The absolute position to move joint 4, expressed in degrees.

Description

Note that this function is similar to the `moveTo()` family of functions, except that this variant ignores the speed settings of the robot joints. This function causes the robot to move its joints towards its goal using a Proportional-Integral-Derivative (PID) controller.

CMobotGroup::driveTo()
This function moves all of the joints of a mobot to the specified absolute positions.

CMobotGroup::driveToNB()
This function moves all of the joints of a mobot to the specified absolute positions.

The function `driveToNB()` is the non-blocking version of the `driveTo()` function, which means that the function will return immediately and the physical mobot motion will occur asynchronously. For more details on blocking and non-blocking functions, please refer to Section 13 on page 44.

Example

See Also

CMobotGroup::motionArch()
CMobotGroup::motionArchNB()

Synopsis

```
#include <mobot.h>
int CMobotGroup::motionArch(double angle);
int CMobotGroup::motionArchNB(double angle);
```

Purpose

Arch the mobots in the group for more ground clearance.

Return Value

The function returns 0 on success and non-zero otherwise.

Parameters

`angle`	The angle which to arch. This number can range from 0 degrees, which is no arch, to 180 degrees, which is a fully curled up position.

Description

CMobot::motionArch()
This function causes the mobots to Arch up for better ground clearance while rolling.

CMobot::motionArch()
This function causes the mobots to Arch up for better ground clearance while rolling.

This function has both a blocking and non-blocking version. The blocking version, `motionArch()`, will block until the mobot motion has completed. The non-blocking version, `motionArchNB()`, will return immediately, and the motion will be performed asynchronously.

See Also

CMobotGroup::moveDistance()
CMobotGroup::moveDistanceNB()

Synopsis

```
#include <mobot.h>
int CMobotGroup::moveDistance(double distance, double radius);
int CMobotGroup::moveDistanceNB(double distance, double radius);
```

Purpose

Use the faceplates as wheels to roll mobots forward a certain distance.

Return Value

The function returns 0 on success and non-zero otherwise.

Parameters

`distance`	The distance to roll the Mobot.
`radius`	The radius of the wheels.

Description

CMobot::moveDistance()
This function is used to roll a group of wheeled Mobots forward a certain distance. Note that the unit of measurement for the radius of the wheels much match the units used for distance.

This function has both a blocking and non-blocking version. The blocking version, `moveDistance()`, will block until the mobot motion has completed. The non-blocking version, `moveDistanceNB()`, will return immediately, and the motion will be performed asynchronously.

See Also

CMobotGroup::motionInchwormLeft() CMobotGroup::motionInchwormLeftNB()

Synopsis

```
#include <mobot.h>
int CMobotGroup::motionInchwormLeft(int num);
int CMobotGroup::motionInchwormLeftNB(int num);
```

Purpose
Make all mobots in the group perform the inch-worm gait to the left.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters

`num`	The number of times to perform the inchworm gait.

Description

CMobot::motionInchwormLeft()
This function causes the mobots to perform a single cycle of the inchworm gait to the left.

CMobot::motionInchwormLeftNB()
This function causes the mobots to perform a single cycle of the inchworm gait to the left.

The function `motionInchwormLeft()` is blocking, and the function will hang until the motion has finished. The alternative function, `motionInchwormLeftNB()` will return immediately, and the motion will execute asynchronously.

See Also
`motionInchwormRight()`

CMobotGroup::motionInchwormRight() CMobotGroup::motionInchwormRightNB()

Synopsis

```
#include <mobot.h>
int CMobotGroup::motionInchwormRight(int num);
int CMobotGroup::motionInchwormRightNB(int num);
```

Purpose
Make all the mobots in the group perform the inch-worm gait to the right.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters

`num`	The number of times to perform the inchworm gait.

Description

CMobot::motionInchwormRight()
This function causes the mobots to perform a single cycle of the inchworm gait to the right.

CMobot::motionInchwormRightNB()
This function causes the mobots to perform a single cycle of the inchworm gait to the right.

This function has both a blocking and non-blocking version. The blocking version, `motionInchwormRight()`, will block until the mobot motion has completed. The non-blocking version, `motionInchwormRightNB()`, will return immediately, and the motion will be performed asynchronously.

See Also
`motionInchwormLeft()`

CMobotGroup::motionSkinny()
CMobotGroup::motionSkinnyNB()

Synopsis
```
#include <mobot.h>
int CMobotGroup::motionSkinny(double angle);
int CMobotGroup::motionSkinnyNB(double angle);
```

Purpose
Move the mobots in the group into a skinny profile.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters

`angle`	The angle in degrees to move the joints. A value of zero means a completely flat profile, while a value of 90 degrees means a fully skinny profile.

Description

CMobot::motionSkinny()
This function makes the mobots assume a skinny rolling profile.

CMobot::motionSkinnyNB()
This function makes the mobots assume a skinny rolling profile.

This function has both a blocking and non-blocking version. The blocking version, `motionSkinny()`, will block until the mobot motion has completed. The non-blocking version, `motionSkinnyNB()`, will return immediately, and the motion will be performed asynchronously.

See Also

CMobotGroup::motionStand()
CMobotGroup::motionStandNB()

Synopsis

```
#include <mobot.h>
int CMobotGroup::motionStand();
int CMobotGroup::motionStandNB();
```

Purpose

Stand mobots up on a faceplate.

Return Value

The function returns 0 on success and non-zero otherwise.

Parameters

None.

Description

CMobot::motionStand()
This function causes the mobots to stand up into the camera platform.

CMobot::motionStandNB()
This function causes the mobots to stand up into the camera platform.

This function has both a blocking and non-blocking version. The blocking version, `motionStand()`, will block until the mobot motion has completed. The non-blocking version, `motionStandNB()`, will return immediately, and the motion will be performed asynchronously.

See Also

CMobotGroup::motionTumbleLeft()
CMobotGroup::motionTumbleLeftNB()

Synopsis

```
#include <mobot.h>
int CMobotGroup::motionTumbleLeft(int num);
int CMobotGroup::motionTumbleLeftNB(int num);
```

Purpose

Make the mobots in the group tumble end over end.

Return Value

The function returns 0 on success and non-zero otherwise.

Parameters

`num`	The number of times to tumble.

Description

CMobot::motionTumbleLeft()
This causes the mobot to tumble end over end. The argument, `num`, indicates the number of times to tumble.

CMobot::motionTumbleLeftNB()
This causes the mobot to tumble end over end. The argument, `num`, indicates the number of times to tumble.

This function has both a blocking and non-blocking version. The blocking version, `motionTumbleLeft()`, will block until the mobot motion has completed. The non-blocking version, `motionTumbleLeftNB()`, will return immediately, and the motion will be performed asynchronously.

See Also

CMobotGroup::motionTumbleRight()
CMobotGroup::motionTumbleRightNB()

Synopsis
```
#include <mobot.h>
int CMobotGroup::motionTumbleRight(int num);
int CMobotGroup::motionTumbleRightNB(int num);
```

Purpose
Make the mobots in the group tumble end over end.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters

`num`	The number of times to tumble.

Description

CMobot::motionTumbleRight()
This causes the mobot to tumble end over end. The argument, `num`, indicates the number of times to tumble.

CMobot::motionTumbleRightNB()
This causes the mobot to tumble end over end. The argument, `num`, indicates the number of times to tumble.

This function has both a blocking and non-blocking version. The blocking version, `motionTumbleRight()`, will block until the mobot motion has completed. The non-blocking version, `motionTumbleRightNB()`, will return immediately, and the motion will be performed asynchronously.

See Also

CMobotGroup::motionUnstand()
CMobotGroup::motionUnstandNB()

Synopsis

```
#include <mobot.h>
int CMobotGroup::motionUnstand();
int CMobotGroup::motionUnstandNB();
```

Purpose
Move mobots currently standing on a faceplate back down into a prone position.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters
None.

Description

CMobot::motionUnstand()
This function causes the mobot to move down from the camera platform.

CMobot::motionUnstandNB()
This function causes the mobot to move down from the camera platform.

This function has both a blocking and non-blocking version. The blocking version, `motionUnstand()`, will block until the mobot motion has completed. The non-blocking version, `motionUnstandNB()`, will return immediately, and the motion will be performed asynchronously.

See Also

CMobotGroup::motionWait()

Synopsis
```
#include <mobot.h>
int CMobotGroup::motionWait();
```

Purpose
Wait for a preprogrammed mobotic motion to finish.

Return Value
The function returns 0 on success and non-zero otherwise.

Description
This function is used to wait for a preprogrammed motion to finish. Functions such as `motionInchwormLeftNB()` and `moveForwardNB()` do not wait for the motion to finish moving before continuing. The `motionWait()` function is used to wait for preprogrammed motions to complete. See Section 11 for a list of all preprogrammed mobotic motions.

Example
See the sample program in Section 9.1.2 on page 30.
See Also
`motionWait(), moveJointWait()`

CMobotGroup::move()

CMobotGroup::moveNB()

Synopsis

```
#include <mobot.h>
int CMobotGroup::move(double angle1, double angle2, double angle3, double angle4);
int CMobotGroup::moveNB(double angle1, double angle2, double angle3, double angle4);
```

Purpose

Move all of the joints of mobots in a group by specified angles.

Return Value

The function returns 0 on success and non-zero otherwise.

Parameters

`angle1`	The amount to move joint 1, expressed in degrees relative to the current position.
`angle2`	The amount to move joint 2, expressed in degrees relative to the current position.
`angle3`	The amount to move joint 3, expressed in degrees relative to the current position.
`angle4`	The amount to move joint 4, expressed in degrees relative to the current position.

Description

CMobot::move()

This function moves all of the joints of a mobot by the specified number of degrees from their current positions.

CMobot::moveNB()

This function moves all of the joints of a mobot by the specified number of degrees from their current positions.

The function `moveNB()` is the non-blocking version of the `move()` function, which means that the function will return immediately and the physical mobot motion will occur asynchronously. For more information on blocking and non-blocking functions, please refer to Section 13 on page 44.

Example

See Also

CMobotGroup::moveBackward()
CMobotGroup::moveBackwardNB()

Synopsis

```
#include <mobot.h>
int CMobotGroup::moveBackward(double angle);
int CMobotGroup::moveBackwardNB(double angle);
```

Purpose

Use the faceplates as wheels to roll all the mobots in a group backward.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters

`angle`	The angle to turn the wheels, specified in degrees.

Description

CMobot::moveBackward()
This function causes each of the faceplates to rotate 90 degrees to roll the mobots backward.

CMobot::moveBackwardNB()
This function causes each of the faceplates to rotate 90 degrees to roll the mobots backward.

This function has both a blocking and non-blocking version. The blocking version, `moveBackward()`, will block until the mobot motion has completed. The non-blocking version, `moveBackwardNB()`, will return immediately, and the motion will be performed asynchronously.

See Also
`moveForward()`

CMobotGroup::moveForward()
CMobotGroup::moveForwardNB()

Synopsis
```
#include <mobot.h>
int CMobotGroup::moveForward(double angle);
int CMobotGroup::moveForwardNB(double angle);
```

Purpose
Use the faceplates as wheels to roll mobots forward.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters

`angle`	The angle to turn the wheels, specified in degrees.

Description

CMobot::moveForward()
This function causes each of the faceplates to rotate 90 degrees to roll the mobots forward.

CMobot::moveForwardNB()
This function causes each of the faceplates to rotate 90 degrees to roll the mobots forward.

This function has both a blocking and non-blocking version. The blocking version, `moveForward()`, will block until the mobot motion has completed. The non-blocking version, `moveForwardNB()`, will return immediately, and the motion will be performed asynchronously.

See Also
`moveBackward()`

CMobotGroup::moveJoint()
CMobotGroup::moveJointNB()

Synopsis

```
#include <mobot.h>
int CMobotGroup::moveJoint(mobotJointId_t id, double angle);
int CMobotGroup::moveJointNB(mobotJointId_t id, double angle);
```

Purpose

Move a joint on the mobots in the group by a specified angle with respect to the current position.

Return Value

The function returns 0 on success and non-zero otherwise.

Parameters

`id`	The joint number to wait for.
`angle`	The angle in degrees to move the motor, relative to the current position.

Description

CMobot::moveJoint()
This function commands the motor to move by an angle relative to the joint's current position at the joints current speed setting. The current motor speed may be set with the `setJointSpeed()` member function. Please note that if the motor speed is set to zero, the motor will not move after calling the `moveJoint()` function.

CMobot::moveJointNB()
This function commands the motor to move by an angle relative to the joint's current position at the joints current speed setting. The current motor speed may be set with the `setJointSpeed()` member function. Please note that if the motor speed is set to zero, the motor will not move after calling the `moveJointNB()` function.

The function `moveJoint()` is a blocking function, which means that the function will not return until the commanded motion is completed. The function `moveJointNB()` is the non-blocking version of the `moveJoint()` function, which means that the function will return immediately and the physical mobot motion will occur asynchronously. For more details on blocking and non-blocking functions, please refer to Section 13 on page 44.

Example

Please see the example in Section 9.1.2 on page 30.

See Also

`connectWithAddress()`

CMobotGroup::moveJointTo()
CMobotGroup::moveJointToNB()

Synopsis

```
#include <mobot.h>
int CMobotGroup::moveJointTo(mobotJointId_t id, double angle);
int CMobotGroup::moveJointToNB(mobotJointId_t id, double angle);
```

Purpose
Move a joint on mobots to an absolute position.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters

`id`	The joint number to wait for.
`angle`	The absolute angle in degrees to move the motor to.

Description

CMobot::moveJointTo()
This function commands the motor on mobots in a group to move to a position specified in degrees at the current motor's speed. The current motor speed may be set with the `setJointSpeed()` member function. Please note that if the motor speed is set to zero, the motor will not move after calling the `moveJointTo()` function.

CMobot::moveJointToNB()
This function commands the motor on mobots in a group to move to a position specified in degrees at the current motor's speed. The current motor speed may be set with the `setJointSpeed()` member function. Please note that if the motor speed is set to zero, the motor will not move after calling the `moveJointToNB()` function.

The function `moveJointTo()` is a blocking function, which means that the function will not return until the commanded motion is completed. The function `moveJointToNB()` is the non-blocking version of the `moveJointTo()` function, which means that the function will return immediately and the physical mobot motion will occur asynchronously. For more details on blocking and non-blocking functions, please refer to Section 13 on page 44.

Example
Please see the example in Section 9.1.2 on page 30.

See Also

CMobotGroup::moveJointWait()

Synopsis
```
#include <mobot.h>
int CMobotGroup::moveJointWait(mobotJointId_t id);
```

Purpose
Wait for a joint to stop moving on all mobots in a group.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters

`id`	The joint number to wait for.

Description
This function is used to wait for a joint motion to finish. Functions such as `moveJointToNB()` and `moveJointNB()`

do not wait for a joint to finish moving before continuing to allow multiple joints to move at the same time. The `moveWait()` or `moveJointWait()` functions are used to wait for mobotic joint motions to complete.

Please note that if this function is called after a motor has been commanded to turn indefinitely, this function may never return and your program may hang.

Example
Please see the example in Section 9.1.2 on page 30.

See Also
`moveWait()`

CMobotGroup::moveTo()
CMobotGroup::moveToNB()

Synopsis
```
#include <mobot.h>
int CMobotGroup::moveTo(double angle1, double angle2, double angle3, double angle4);
int CMobotGroup::moveToNB(double angle1, double angle2, double angle3, double angle4);
```

Purpose
Move all of the joints of mobots in the group to the specified positions.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters

`angle1`	The absolute position to move joint 1, expressed in degrees.
`angle2`	The absolute position to move joint 2, expressed in degrees.
`angle3`	The absolute position to move joint 3, expressed in degrees.
`angle4`	The absolute position to move joint 4, expressed in degrees.

Description

CMobot::moveTo()
This function moves all of the joints of mobots in the group to the specified absolute positions.

CMobot::moveToNB()
This function moves all of the joints of mobots in the group to the specified absolute positions.

The function `moveTo()` is a blocking function, which means that the function will not return until the commanded motion is completed. The function `moveToNB()` is the non-blocking version of the `moveTo()` function, which means that the function will return immediately and the physical mobot motion will occur asynchronously. For more details on blocking and non-blocking functions, please refer to Section 13 on page 44.

Example
Please see the demo at Section 9.1.2 on page 30.

See Also

CMobotGroup::moveToZero()

CMobotGroup::moveToZeroNB()

Synopsis
```
#include <mobot.h>
int CMobotGroup::moveToZero();
int CMobotGroup::moveToZeroNB();
```

Purpose
Move all of the joints of mobots in the group to their zero position.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters
None.

Description

CMobot::moveToZero()
This function moves all of the joints of mobots in the group to their zero position. Please note that this function is non-blocking and will return immediately. Use this function in conjunction with the `moveWait()` function to block until the movement completes.

CMobot::moveToZeroNB()
This function moves all of the joints of mobots in the group to their zero position. Please note that this function is non-blocking and will return immediately. Use this function in conjunction with the `moveWait()` function to block until the movement completes.

The function `moveToZero()` is a blocking function, which means that the function will not return until the commanded motion is completed. The function `moveToZeroNB()` is the non-blocking version of the `moveToZero()` function, which means that the function will return immediately and the physical mobot motion will occur asynchronously. For more details on blocking and non-blocking functions, please refer to Section 13 on page 44.

Example
Please see the demo at Section 9.1.2 on page 30.

See Also

CMobotGroup::moveWait()

Synopsis
```
#include <mobot.h>
int CMobotGroup::moveWait();
```

Purpose
Wait for all joints of all mobots in the group to stop moving.

Return Value
The function returns 0 on success and non-zero otherwise.

Description
This function is used to wait for all joint motions to finish. Functions such as `moveJointToNB()` and

`moveJointNB()` do not wait for a joint to finish moving before continuing to allow multiple joints to move at the same time. The `moveWait()` or `moveJointWait()` functions are used to wait for mobotic motions to complete.

Please note that if this function is called after a motor has been commanded to turn indefinitely, this function may never return and your program may hang.

Example
See the sample program in Section 9.1.2 on page 30.
See Also
`moveWait(), moveJointWait()`

CMobotGroup::resetToZero()
CMobotGroup::resetToZeroNB()

Synopsis
```
#include <mobot.h>
int CMobotGroup::resetToZero();
int CMobotGroup::resetToZeroNB();
```

Purpose
Move all of the joints of mobots in the group to their zero position.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters
None.

Description

CMobot::resetToZero()
This function moves all of the joints of mobots in the group to their zero position. Please note that this function is non-blocking and will return immediately. Use this function in conjunction with the `moveWait()` function to block until the movement completes.

This function differs from the `moveToZero()` function by limiting the total rotation back to the zero position to 180 degrees max, whereas the `moveToZero()` function will completely "rewind" back to the zero position, possible turning the faceplate joints more than 180 degrees.

CMobot::resetToZeroNB()
This function moves all of the joints of mobots in the group to their zero position. Please note that this function is non-blocking and will return immediately. Use this function in conjunction with the `moveWait()` function to block until the movement completes.

The function `resetToZero()` is a blocking function, which means that the function will not return until the commanded motion is completed. The function `resetToZeroNB()` is the non-blocking version of the `resetToZero()` function, which means that the function will return immediately and the physical mobot motion will occur asynchronously. For more details on blocking and non-blocking functions, please refer to Section 13 on page 44.

Example
Please see the demo at Section 9.1.2 on page 30.

See Also

CMobotGroup::setExitState()

Synopsis
```
#include <mobot.h>
int CMobotGroup::setExitState(mobotJointState_t state);
```

Purpose
Sets the behaviour of joints when the program exits.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters

`state`	The desired behaviour of the robot joints on program exit. By default, the behavior is set to `MOBOT_NEUTRAL`, which relaxes all of the joints. Alternatively, the state may be set to `MOBOT_HOLD` to hold the joints on program exit.

Description
This function is used to change the behaviour of the joints when the program exits. By default, when a Mobot program exits, the all joints of the connected Mobots go to the `MOBOT_NEUTRAL` state. Using this function, the exit state may be changed to `MOBOT_HOLD` or `MOBOT_NEUTRAL`. All other joint states are invalid when used with this function.
Example

See Also

CMobotGroup::setJointMovementStateNB()

Synopsis
```
#include <mobot.h>
int CMobotGroup::setJointMovementStateNB(mobotJointId_t id, mobotJointState_t dir1);
```

Purpose
Move a joint of a mobot continuously in the specified directions.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters

`id`	The joint number to move.
`dir`	This parameter specifies the direction the joint should move.

Description
The `dir` parameter specifies the direction the joint should move. The types are enumerated in `mobot.h` and have the following values:

Value	Description
MOBOT_NEUTRAL	This value indicates that the joint is not moving and is not actuated. The joint is freely backdrivable.
MOBOT_FORWARD	This value indicates that the joint is currently moving forward.
MOBOT_BACKWARD	This value indicates that the joint is currently moving backward.
MOBOT_HOLD	This value indicates that the joint is currently not moving and is holding its current position. The joint is not currently backdrivable.

More documentation about these types may be found at Section **??** on page **??**.

This function causes joints of a mobot to begin moving at the previously set speed. The joints will continue moving until the joint hits a joint limit, or the joint is stopped by setting the speed to zero. This function is a non-blocking function.

Example

See Also

CMobotGroup::setJointMovementStateTime()
CMobotGroup::setJointMovementStateTimeNB()

Synopsis

```
#include <mobot.h>
int CMobotGroup::setJointMovementStateTime(mobotJointId_t id, mobotJointState_t dir1, double seconds);
int CMobotGroup::setJointMovementStateTimeNB(mobotJointId_t id, mobotJointState_t dir1, double seconds)
```

Purpose

Move a joint of a mobot continuously in the specified directions for a certain amount of time.

Return Value

The function returns 0 on success and non-zero otherwise.

Parameters

id	The joint number to move.
dir	This parameter specifies the direction the joint should move.
seconds	The number of seconds to rotate the joint.

Description

The dir parameter specifies the direction the joint should move. The types are enumerated in mobot.h and have the following values:

Value	Description
MOBOT_NEUTRAL	This value indicates that the joint is not moving and is not actuated. The joint is freely backdrivable.
MOBOT_FORWARD	This value indicates that the joint is currently moving forward.
MOBOT_BACKWARD	This value indicates that the joint is currently moving backward.
MOBOT_HOLD	This value indicates that the joint is currently not moving and is holding its current position. The joint is not currently backdrivable.

More documentation about these types may be found at Section **??** on page **??**.

This function causes joints of a mobot to begin moving at the previously set speed. The joints will continue moving until the joint hits a joint limit, the joint is stopped by setting the speed to zero, or the

specified amount of time has expired.

Example

See Also

CMobotGroup::setJointSafetyAngle()

Synopsis
```
#include <mobot.h>
int CMobotGroup::setJointSafetyAngle(double degrees);
```

Purpose
Set the current angle safety limit of the Mobot.

Return Value
The function returns 0 on success and -1 on failure.

Parameters
The requested joint safety angle limit for the Mobot.

Description
The Mobot is equipped with a safety feature to protect itself and its surrounding environment. When a motor deviates by a certain amount from its expected value, the Mobot will shut off all power to the motor, in case it has hit an obstacle, or for any other reason. The amount of deviation required to trigger the safety protocol is the joint safety angle which can be set using this function. The default setting is 10 degrees. Higher values indicate "less safe" behavior of the Mobot because the Mobot will not engage safety protocols until the joint has deviated by a greater amount. Values greater than 90 degrees effectively disengage the Mobot's safety protocols altogether, and this function should be used with care.
Example

See Also
`setJointSafetyAngleTimeout()`

CMobotGroup::setJointSafetyAngleTimeout()

Synopsis
```
#include <mobot.h>
int CMobotGroup::setJointSafetyAngleTimeout(double seconds);
```

Purpose
Set the current angle safety limit timeout of the Mobot.

Return Value
The function returns 0 on success and -1 on failure.

Parameters
A variable which will be overwritten with the safety angle limit timeout in seconds.

Description
The Mobot is equipped with a safety feature to protect itself and its surrounding environment. When a

motor deviates by a certain amount from its expected value, the Mobot will shut off all power to the motor after a certain period of time, in case it has hit an obstacle, or for any other reason. The period of time that the mobot waits before shutting the motor off is the joint safety angle timeout, which can be set with this function. The default value for the timeout is 500 milliseconds, or 0.5 seconds.

Example

See Also

`setJointSafetyAngle()`

CMobotGroup::setJointSpeed()

Synopsis

```
\vspace{-8pt}
#include <mobot.h>
int CMobotGroup::setJointSpeed(mobotJointId_t id, double speed);
```

Purpose

Set the speed of a joint on all mobots in the group.

Return Value

The function returns 0 on success and non-zero otherwise.

Parameters

`id`	The joint number to pose.
`speed`	An variable of type `double` for the requested average angular speed in degrees per second.

Description

This function is used to set the angular speed of a joint of all mobots in the group.

Example

See Also

CMobotGroup::setJointSpeedRatio()

Synopsis

```
#include <mobot.h>
int CMobotGroup::setJointSpeedRatio(mobotJointId_t id, double ratio);
```

Purpose

Set the speed ratio settings of a joint on all mobots in the group.

Return Value

The function returns 0 on success and non-zero otherwise.

Parameters

`id`	Set the speed ratio setting of this joint. This is an enumerated type discussed in Section **??** on page **??**.
`ratio`	A variable of type double with a value from 0 to 1.

Description

This function is used to set the speed ratio setting of a joint for all mobots in the group. The speed ratio

setting of a joint is the percentage of the maximum joint speed, and the value ranges from 0 to 1. In other words, if the ratio is set to 0.5, the joint will turn at 50% of its maximum angular velocity while moving continuously or moving to a new goal position.

Example

See Also
`setJointSpeeds()`, `setJointSpeedRatio()`

CMobotGroup::setJointSpeedRatios()

Synopsis
```
#include <mobot.h>
int CMobotGroup::setJointSpeedRatios(double ratio1, double ratio2, double ratio3, double ratio4);
```

Purpose
Set the speed ratio settings of all joints on the mobots in the group.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters
- `ratio1` The speed ratio setting for the first joint.
- `ratio2` The speed ratio setting for the second joint.
- `ratio3` The speed ratio setting for the third joint.
- `ratio4` The speed ratio setting for the fourth joint.

Description
This function is used to simultaneously set the angular speed ratio settings of all four joints of a mobot for all mobots in the group. The speed ratio is a percentage of the maximum speed of a joint, expressed in a value from 0 to 1.

Example

See Also
`getJointSpeeds()`, `setJointSpeed()`, `getJointSpeed()`

CMobotGroup::setJointSpeeds()

Synopsis
```
#include <mobot.h>
int CMobotGroup::setJointSpeeds(double speed1, double speed2, double speed3, double speed4);
```

Purpose
Set the speed settings of all joints on all mobot in the group.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters
- `speed1` The speed setting for the first joint, in units of degrees per second.
- `speed2` The speed setting for the second joint, in units of degrees per second.
- `speed3` The speed setting for the third joint, in units of degrees per second.
- `speed4` The speed setting for the fourth joint, in units of degrees per second.

Description
This function is used to simultaneously set the angular speed settings of all four joints of all mobots in the group. The joint speeds are expressed in degrees per second.

Example

See Also
`setJointSpeed()`

CMobotGroup::setMovementStateNB()

Synopsis
```
#include <mobot.h>
int CMobotGroup::setMovementStateNB(
  mobotJointState_t dir1,
  mobotJointState_t dir2,
  mobotJointState_t dir3,
  mobotJointState_t dir4);
```

Purpose
Move the joints of grouped mobots continuously in the specified directions.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters
Each integer parameter specifies the direction the joint should move. The types are enumerated in `mobot.h` and have the following values:

Value	Description
`MOBOT_NEUTRAL`	This value indicates that the joint is not moving and is not actuated. The joint is freely backdrivable.
`MOBOT_FORWARD`	This value indicates that the joint is currently moving forward.
`MOBOT_BACKWARD`	This value indicates that the joint is currently moving backward.
`MOBOT_HOLD`	This value indicates that the joint is currently not moving and is holding its current position. The joint is not currently backdrivable.

More documentation about these types may be found at Section **??** on page **??**.

Description
This function causes joints of mobots to begin moving at the previously set speed. The joints will continue moving until the joint hits a joint limit, or the joint is stopped by setting the speed to zero. This function is a non-blocking function.

Example

See Also

CMobotGroup::setMovementStateTime()
CMobotGroup::setMovementStateTimeNB()

Synopsis

```
#include <mobot.h>
int CMobotGroup::setMovementStateTime(mobotJointState_t dir1,
                                      mobotJointState_t dir2,
                                      mobotJointState_t dir3,
                                      mobotJointState_t dir4,
                                      double seconds);
int CMobotGroup::setMovementStateTimeNB(mobotJointState_t dir1,
                                      mobotJointState_t dir2,
                                      mobotJointState_t dir3,
                                      mobotJointState_t dir4,
                                      double seconds);
```

Purpose
Move the joints of mobots continuously in the specified directions.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters
Each of the direction parameters, `dir1, dir2, dir3,` and `dir4`, specifies the direction the joint should move. The types are enumerated in `mobot.h` and have the following values:

Value	Description
`MOBOT_NEUTRAL`	This value indicates that the joint is not moving and is not actuated. The joint is freely backdrivable.
`MOBOT_FORWARD`	This value indicates that the joint is currently moving forward.
`MOBOT_BACKWARD`	This value indicates that the joint is currently moving backward.
`MOBOT_HOLD`	This value indicates that the joint is currently not moving and is holding its current position. The joint is not currently backdrivable.

The `seconds` parameter is the time to perform the movement, in seconds.

Description
This function causes joints of mobots to begin moving. The joints will continue moving until the joint hits a joint limit, or the time specified in the `seconds` parameter is reached. This function will block until the motion is completed.

Example

See Also

CMobotGroup::setTwoWheelRobotSpeed()

Synopsis
```
#include <mobot.h>
int CMobotGroup::setTwoWheelRobotSpeed(double speed, double radius);
```

Purpose
Roll the mobots in the group at a certain speed in a straight line.

Return Value
The function returns 0 on success and non-zero otherwise.

Parameters

`speed` The speed at which to roll the mobot. The units used will be the units specified in the `unit` parameter.

`radius` The radius of the wheels attached to the mobot. The units of the parameter should match the units provided in the `unit` parameter.

speed	radius
cm/s	cm
m/s	m
inch/s	inch
foot/s	foot

Description

This function is used to make a two wheeled mobot roll at a certain speed. The desired speed and radius of the wheels is provided and the function will rotate the wheels at the appropriate rate in order to achieve the desired speed.

Example

See Also

CMobotGroup::stopAllJoints()
CMobotGroup::stopOneJoint()
CMobotGroup::stopTwoJoints()
CMobotGroup::stopThreeJoints()

Synopsis

```
#include <mobot.h>
int CMobotGroup::stopAllJoints();
int CMobotGroup::stopOneJoint(mobotJointId_t id);
int CMobotGroup::stopTwoJoints(mobotJointId_t id1, mobotJointId_t id2);
int CMobotGroup::stopThreeJoints(
    mobotJointId_t id1,
    mobotJointId_t id2,
    mobotJointId_t id3);
```

Purpose

Stop all current motions on all mobot in the group.

Return Value

The function returns 0 on success and non-zero otherwise.

Description

This function stops all currently occurring movements on the mobot. Internally, this function simply sets all motor speeds to zero. If it is only required to stop a single motor, use the `setJointSpeed()` function to set the motor's speed to zero.

Example

See Also

`setJointSpeed()`, `setJointSpeeds()`

CMobotGroup::turnLeft()
CMobotGroup::turnLeftNB()

Synopsis

```
#include <mobot.h>
int CMobotGroup::turnLeft(double angle);
int CMobotGroup::turnLeftNB(double angle);
```

Purpose

Rotate the mobots using the faceplates as wheels.

Return Value

The function returns 0 on success and non-zero otherwise.

Parameters

`angle` The angle in degrees to turn the wheels. The wheels will turn in opposite directions by the amount specified by this argument in order to turn the mobot to the left.

Description

This function causes the mobots to rotate the faceplates in opposite directions to cause the mobot to rotate counter-clockwise.

This function has both a blocking and non-blocking version. The blocking version, `turnLeft()`, will block until the mobot motion has completed. The non-blocking version, `turnLeftNB()`, will return immediately, and the motion will be performed asynchronously.

See Also

`turnRight()`

CMobotGroup::turnRight()
CMobotGroup::turnRightNB()

Synopsis

```
#include <mobot.h>
int CMobotGroup::turnRight(double angle);
int CMobotGroup::turnRightNB(double angle);
```

Purpose

Rotate the mobots using the faceplates as wheels.

Return Value

The function returns 0 on success and non-zero otherwise.

Parameters

`angle` The angle in degrees to turn the wheels. The wheels will turn in opposite directions by the amount specified by this argument in order to turn the mobot to the right.

Description

This function causes the mobots to rotate the faceplates in opposite directions to cause the mobot to rotate clockwise.

This function has both a blocking and non-blocking version. The blocking version, `turnRight()`, will block until the mobot motion has completed. The non-blocking version, `turnRightNB()`, will return immediately, and the motion will be performed asynchronously.

See Also
`turnLeft()`

C Miscellaneous Utility Functions

There are several utility functions which are useful when programming for the Mobot.

Table 5: Mobot Utility Functions.

Function	Description
`angle2distance()`	Calculates the angle a wheel has turned from the radius and distance traveled.
`angles2distances()`	Calculates the angle a wheel has turned from the radius and distance traveled. Use this function when the data is stored in normal C arrays rather than computational arrays.
`deg2rad()`	Converts degrees to radians.
`delay()`	Puts a pause into a program.
`distance2angle()`	Calculates the distance traveled by a wheel from the wheel's radius and angle turned.
`rad2deg()`	Converts radians to degrees.
`shiftTime()`	Shift the data of a plot.

angle2distance()

Synopsis

```
#include <mobot.h>
double angle2distance(double radius, double angle);
array double angle2distance(double radius, array double angle[:])[:];
```

Purpose

Calculate the distance a wheel has traveled from the radius of the wheel and the angle the wheel has turned.

Return Value

The value returned is the distance traveled by the wheel. If the angle argument is an array of angles, then the value returned is an array of distances. Each element of the distance array returned is the distance calculated from the respective element in the angle array.

Parameters

`radius`	The radius of the wheel.
`angle`	This value is the angle the wheel has turned. This parameter may be of `double` type, or a Ch computational array.

Description

This function calculates the angle a wheel has turned given the wheel radius and distance traveled. The equation used is

$$d = r\theta$$

where d is the distance traveled, r is the radius of the wheel, and θ is the angle the wheel has turned in radians.

Example

See Also

`distance2angle()`

angles2distances()

Synopsis

```
#include <mobot.h>
void angles2distances(double radius, double *angles, double *distances, int num);
```

Purpose

Calculate the distance a wheel has traveled from the radius of the wheel and the angle the wheel has turned.

Return Value

The value returned is the distance traveled by the wheel. If the angle argument is an array of angles, then the value returned is an array of distances. Each element of the distance array returned is the distance calculated from the respective element in the angle array.

Parameters

`radius`	The radius of the wheel.
`angles`	(In) An array of angle values.
`distances`	(Out) An array that will be filled with distance values.
`num`	The number of elements in the `angles` array.

Description
This function calculates the angle a wheel has turned given the wheel radius and distance traveled. The equation used is

$$d = r\theta$$

where d is the distance traveled, r is the radius of the wheel, and θ is the angle the wheel has turned in radians.

Example

See Also
`distance2angle()`

---

deg2rad()

Synopsis
```
#include <mobot.h>
double deg2rad(double degrees);
array double deg2rad(double degrees[:])[:];
```

Purpose
Convert degrees to radians.

Return Value
The angle parameter converted to radians.

Parameters

`degrees`	The angle to convert, in degrees.

Description
This function converts an angle expressed in degrees into radians. Degrees and radians are two popular ways to express an angle, though they are not interchangable. The following equation is used to convert degrees to radians:

$$\theta = \delta * \frac{\pi}{180}$$

where θ is the angle in radians and δ is the angle in degrees.
Example

See Also
`rad2deg()`

---

delay()

Synopsis
```
#include <mobot.h>
void delay(double seconds);
```

Purpose
Pause a program for a set amount of time.

Return Value
None.

Parameters

`seconds` The number of seconds to delay.

Description
This function delays or pauses a program for a number of seconds. For instance, the code

```
delay(0.5);
printf("Hello.\n");
delay(2);
printf("Goodbye.\n");
```

will pause for half a second, print the text `Hello.`, delay for 2 seconds, and then print the text `Goodbye.`.
Example

See Also

---

distance2angle()

Synopsis
```
#include <mobot.h>
double distance2angle(double radius, double distance);
array double distance2angle(double radius, array double distance[:])[:];
```

Purpose
Calculate the angle a wheel has turned from the radius of the wheel and the distance the wheel has traveled.

Return Value
The value returned is the angle turned by the wheel in degrees. If the distance argument is an array of distances, then the value returned is an array of angles. Each element of the angle array returned is the angle calculated from the respective element in the distance array.

Parameters
`radius` The radius of the wheel.
`distance` This value is the distance the wheel has traveled. This parameter may be of `double` type, or a Ch computational array.

Description
This function calculates the distance a wheel has turned given the wheel radius and angle turned. The equation used is

$$\theta = \frac{d}{r}$$

where d is the distance traveled, r is the radius of the wheel, and θ is the angle the wheel has turned in radians. A further conversion is done in the code to convert the angle from radians into degrees before returning the value.

Example

See Also
`angle2distance()`

---

rad2deg()

Synopsis

```
#include <mobot.h>
double rad2deg(double radians);
array double rad2deg(double radians[:])[:];
```

Purpose
Convert radians to degrees.

Return Value
The angle parameter converted to degrees.

Parameters

`radians`	The angle to convert, in radians.

Description
This function converts an angle expressed in radians into degrees. Degrees and radians are two popular ways to express an angle, though they are not interchangable. The following equation is used to convert radians to degrees:

$$\delta = \theta * \frac{180}{\pi}$$

where θ is the angle in radians and δ is the angle in degrees.
Example

See Also
`deg2rad()`

---

shiftTime()

Synopsis
```
#include <mobot.h>
int shiftTime(double tolerance, int numDataPoints, double time[], double data1[], ...);
```

Syntax
```
#include <mobot.h>
shiftTime(tolerance, numDataPoints, time, angles1);
shiftTime(tolerance, numDataPoints, time, angles1, angles2);
shiftTime(tolerance, numDataPoints, time, angles1, angles2, angles3);
shiftTime(tolerance, numDataPoints, time, angles1, angles2, angles3, angles4);
etc...
```

Purpose
This function is used to shift the data in one or more plots to the left. It is commonly used to line up the beginning of mobot motions with the y-axis on plots.

Return Value
The return value is the number of elements which have been shifted of the plots.

Parameters

`tolerance`	The angle tolerance to detect the beginning of the motion. A lower tolerance is more sensitive to small motions, but also more sensitive to noise. A higher tolerance will reject noise, but may yield an inaccurate shift in time such that the motion does not appear to begin at time 0.
`numDataPoints`	The number of elements in the arrays.
`time`	The array holding time or "x-axis" values.
`data1`	An array holding data.
`...`	Additional arrays holding data.

Description
This function is used to shift data to the left to align motion start times with the y-axis. This is done by detecting a change in value in any of the data arrays provided to the function. If there is a change greater than the value provided as the tolerance, that time is labeled as the beginning of the motion. All data points prior to the beginning of the motion are deleted, and the beginning of the motion is aligned with time 0.
Example

See Also

Index