boB's Guide to Using ARIA

Robert Oates
University of Nottingham

February 7, 2006

Abstract

This guide provides a basic grounding in the use of ARIA and introduces some of the architecture used in standard, ARIA-based programs. It includes a discussion on the standard techniques associated with the implementation of programs for controlling the Pioneer range of robots. This document is not a substitute for reading the ARIA documentation, but deals with the practical issues associated with getting a platform working in the quickest possible time.

Contents

1 Introduction 3

2 Testing an Installation 3

3 The ARIA Architecture 4
3.1 ArRobot - The Robot Class 4
3.2 Sensors and Actuators 4
3.3 Actions, Action Groups and Modes 5
3.3.1 ArAction 5
3.3.2 ArActionGroup 5
3.3.3 ArMode 7
3.3.4 ArActionDesired 7

4 Compiling an ARIA-Enabled Program 7
4.1 Using Windows 7
4.2 Using Linux 8
4.2.1 g++ 8
4.2.2 Eclipse 8

5 Useful ARIA Commands **9**
5.1 Initialising the Robot 9
5.1.1 Argument Parsing and Robot Communications 9
5.1.2 Key Handling 10
5.1.3 Behaviour Modes 10
5.1.4 Generic Sensor Initialisation 11
5.1.5 Laser Initialisation 11
5.2 General Movement 12
5.3 Acquiring Sensor Data 12
5.4 5DOF Arm Control 13
5.5 Miscellaneous 13

6 Troubleshooting **14**
6.1 Linux Only 14

A Example Program - based on the original ARIA *demo* program **15**
A.1 RobotTest.cpp 15
A.2 ArActionWallFollow.h 19
A.3 ArActionGroupFollow.h 29
A.4 ArModeWallFollow.h 29

Table 1: The Programs/Packages Required to Use ARIA Successfully

Program / Package	**Version Used At the Time of Writing**	**Description**
ARIA	2.4.1	The source code and libraries required to create new ARIA programs and some demo applications.
Mapper3-Basic	1.2.1	A program for generating environments for the robot to navigate
Mobile Sim	0.2	The stage-based simulator from ActivMedia

1 Introduction

This guide is designed to get people up and running with the ActivMedia Robotics Interface for Applications, (ARIA) in the fastest possible time. To understand it, you will need some knowledge of C++. It is assumed that you are working with a machine that has the programs/packages listed in Table 1 downloaded and installed.

2 Testing an Installation

If you have only just installed the relevant programs and want to check that the installation was a success, or if you simply want to see an example of ARIA in action, follow the steps outlined in this section. If not, simply skip ahead to section 3.

Start by running the Mobile Sim program, (via the Start menu in Windows or via the command *MobileSim&* from a Linux console) and select the "No Map" option when prompted. This should present you with a large, blank grid with a robot, (the red dot!) in the middle. You can zoom in and out using the wheel on a wheel mouse; move the map centre by right clicking and dragging; click and drag the robot into new positions; and change the robot's orientation by right clicking and dragging the robot. If you now run the program "demo" (via the Start menu in Windows or via the command

./demo from a Linux console), you should be able to control the robot in "teleop" mode, (the demo program will need focus) and allow the robot to wander in "wander" mode.

You can also try generating your own map using "Mapper3-Basic". The "line" tool will construct walls that will effect the simulated robot's sonar and laser-range finder. You can verify this by loading your map and putting the demo program into "wander" mode.

3 The ARIA Architecture

To be able to write ARIA-based programs, it is first necessary to understand the architecture the ARIA library was written to support. You are not forced to write ARIA programs in this way. If you wish, you can simply create a robot class and call the relevant commands for directly manipulating movement, (see section 5.2) however, there are distinct advantages to using the standard architecture, (not least, the very fact that a well-defined code structure makes co-operative projects considerably easier.)

This section will discuss some of the individual classes that are made available by ARIA, however, for a complete reference for all ARIA classes, see [1].

3.1 ArRobot - The Robot Class

This, unsurprisingly, is the main ARIA class. It encapsulates all of the primary information about the robot being controlled. On its own, an instantiation of this class will represent a standard pioneer base, with no sensors, and only the motors as actuators.

3.2 Sensors and Actuators

To elaborate on the main robot class, one must add the additional sensors and actuators to it. Basic, ranged sensors, such as the laser and the sonar, can be added by instantiating the relevant classes, (ArSick and ArSonarDevice respectively) and adding them using ArRobot's *addRangeDevice* method, passing a pointer to the instantiated sensor class as input. See sections 5.1.5 and 5.1.4 for details on initialising the laser. It is of note that the bumpers are treated as a ranged device which reports objects as either being undetected or as existing at the edge of the robot's radius.

There is a useful virtual ranged device that can be added to the robot called the "Forbidden Range Device". This device, (*ArForbiddenRangeDe-*

vice), can be used to provide distance readings from areas marked on the robot's map as "forbidden". You can add "forbidden" lines to a virtual map using Mapper3-Basic. This can be used to create virtual barriers between the robot and hazards that might otherwise be difficult to perceive using ranged sensors, such as stairwells.

The behaviours of other sensors and actuators should be looked up in [1]. Be warned that sensors (with the exception of "ranged sensors"), and actuators, generally attach to the robot (i.e. ArDevice.setRobot(&robot)), as opposed to the robot attaching to them, (i.e. robot.addRangeDevice(&sonar)) as is the case with ranged sensors.

3.3 Actions, Action Groups and Modes

The behaviour of a robot is encapsulated by classes that inherit from the *ArAction*, *ArActionGroup* and *ArMode* classes.

3.3.1 ArAction

The *ArAction* class represents an individual action that the robot can take in response to its circumstances. Pre-defined actions within the Aria library include *ArAvoidFront*, *ArAvoidSide* and *ArActionGoto*, which, unsurprisingly, avoid obstacles in front of the robot, avoid obstacles to the side of the robot, and goto a given position respectively.

The crucial method to override within *ArAction* is *ArAction::fire().* This method receives what the robot currently wants to do as an input, in the form of an *ArActionDesired* object and returns a second *ArActionDesired* object representing what this *ArAction* object would prefer to do. NULL can be returned if the action does not wish to change what the robot is doing.

3.3.2 ArActionGroup

An *ArAction* group is used to simply wrap a collection (one or more) of *ArActions* that collectively implement a behaviour. This is merely for the convenience of being able to simultaneously activate each action with a single call to the group's *activate* method. When adding *ArActions* it is possible to associate a priority with that action. The allocation of priorities to *ArActions* allows the implementation of a subsumption type architecture, with actions being fired from highest level command to lowest, passing their desired output down the chain. Finally, the lowest level ultimately decides whether to implement the high-level request or to override it.

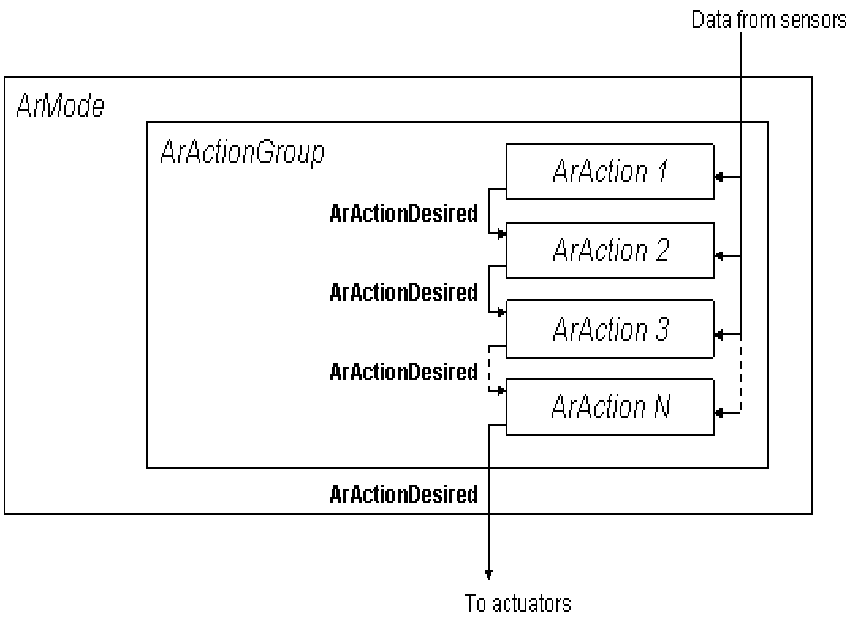

Figure 1: An overview of the relationship between the Action implementation classes. *Italics* marks classes that are inherited from the type specified and **bold** marks actual instantiations of the type specified.

The *ArActionGroup* class is also used by being overridden. The constructor and data member types used in the new class specify the individual *ArAction* types that it is to represent.

3.3.3 ArMode

An *ArMode* encapsulates a robot behaviour. A private data member that is an inherited type from *ArActionGroup* is used to dictate the nature of the *ArMode*. An *ArMode* can be associated with two keyboard characters (usually the lower-case and upper-case of the same letter), which, in the standard ARIA program architecture, will specify the letter the user will push to activate that mode.

3.3.4 ArActionDesired

An ArActionDesired object has various methods for specifying a change in heading or velocity, or absolute headings and velocities. See [1] for a complete list.

4 Compiling an ARIA-Enabled Program

4.1 Using Windows

These instructions were written for Visual Studio .NET 2003.

- Create a new WIN32 console application.
- Open the project properties dialog (*Project⇒[PROJECT NAME] Properties)*
- Select "All Configurations"
- Navigate to the *C/C++⇒General* Section
- Add the ARIA include paths to the *Additional Include Directories*
- Navigate to the *Linker⇒Input* Section
- Add the ARIA library to the *Additional Dependencies*
- Ensure that the location of Aria.dll is stored in your system's path variable

Note that if you intend to run your program on a Linux-based platform you must avoid using any MFC-specific classes and make sure that the data types specified for the "main" function are portable. You can achieve this using conditional compilation.

```
#ifdef WIN32
int _tmain(int argc, _TCHAR* argv[])
```

```
#else
int main (int argc, char** argv)
#endif
```

When compiling for Linux, you will need to comment out references to stdafx.h. Sadly the Microsoft compiler's reliance on stdafx.h means that you cannot use conditional compilation to remove it automatically.

4.2 Using Linux

4.2.1 g++

When compiling an ARIA-enabled program under Linux using g++ on the command line you need to specify additional options to ensure that the appropriate links take place. An example command line follows.

```
g++ -Wall -o OutputFile -lAria -ldl -lpthread -L/usr/local/Aria/lib
-I/usr/local/Aria/include InputFile.cpp
```

In the example command line the program is compiled using the following options: the "Wall" option provides useful warnings about potential coding problems; the "o OutputFile" specifies the output executable name; the "lAria" specifies that the program will require the ARIA library; the "ldl" and "lpthread" arguments specify additional libraries required by ARIA; the "L/usr/local/Aria/lib" specifies the standard location of the ARIA library; the "I/usr/local/Aria/include" specifies the standard location of the ARIA headers; the final argument specifies the main input file for the project.

4.2.2 Eclipse

If you prefer working with an IDE such as Eclipse you will need to specify the information provided to g++ via the menu options.

The *Project⇒Properties* menu option will allow you to configure all of these options. Simply add the relevant information specified in section 4.2.1 via the *GCC C++ Compiler⇒Directories* and *GCC C++ Linker⇒Libraries* menu options. The "-X" labels at the top of each section correspond to a "-X" compiler option from section 4.2.1.

5 Useful ARIA Commands

5.1 Initialising the Robot

Before one can use the ARIA library, it is important that it be initialised. This can be done simply by calling:

```
Aria::init();
```

This should be done at the beginning of the program. At the end of the program, instead of using the standard *return* to leave *main*, call the following, passing the desired return value.

```
Aria::exit(0);
```

5.1.1 Argument Parsing and Robot Communications

To maintain uniformity between ARIA-based programs, the library comes with a standard argument parser. During the initialisation of a program, this can be used to ensure that all the configurable elements of an ARIA program, (robot IP address etc) can be passed in the same way to any ARIA program. *ArArgumentParser::loadDefaultArguments()* will allocate the defaults required to connect to the local host, (either a MobileSim simulation on the current machine or a real robot.)

```
ArArgumentParser parser(&argc, argv);
parser.loadDefaultArguments();
```

Actually connecting to a robot, requires the *ArSimpleConnector* class. This can receive an *ArArgumentParser* class as an initialiser to its constructor. The values specified within the latter class dictate the connection settings.

```
ArSimpleConnector connector(&parser);

if (!connector.parseArgs()) {
    //output some error message
    exit(1);
}
```

Actually connecting to the robot requires an *ArRobot* class, (no initialisation required) to be attached to the connector. Once connected, the robot can be placed into asynchronous mode, which ensures that if the connection is lost, the robot will cease to operate. This is recommended for safety reasons.

```
ArRobot robot;

if (!connector.connectRobot(&robot)) {
    //Some error message here
    exit(1);
}
robot.runAsync(true);
```

Before attempting to run the robot, it is recommended that you place the motors into an enabled and safe state. This is achieved via the command *ArRobot::comInt(ArCommands::ENABLE, 1)*. You can use *ArRobot::lock()* and *ArRobot::unlock()* to ensure that the initialisation command is not interfered with by other users erroneously connecting to the robot.

```
robot.lock();
robot.comInt(ArCommands::ENABLE, 1);
robot.unlock();
```

There are other tasks that benefit from being performed whilst the robot is in the locked state, see section 5.1.3

5.1.2 Key Handling

If you want to be able to control the robot via the terminal, you will require a means of processing key-events. In the ARIA architecture, this is done via the *ARKeyHandler* class. A key handler should be attached to a robot class, before you attempt to connect to the robot. However, you should not attach a key handler if you wish to add the program to the start-up behaviour of the operating system, as this can potentially result in requests being sent to the console, before it is actually allocated, resulting in serious errors on boot up.

A key handler needs to be registered with the main ARIA library, and then attached to a specific robot.

```
ArKeyHandler keyHandler;

Aria::setKeyHandler(&keyHandler);

robot.attachKeyHandler(&keyHandler);
```

5.1.3 Behaviour Modes

You can add any number of behaviour modes to a robot. If a key handler is assigned to the robot, you can assign short-cut keys to each behaviour, and switch between them at run time. It is a good idea to add new modes whilst the robot is in the locked state, to avoid other people erroneously connecting to the robot whilst you are adding the new modes. The code extract which follows, shows two new behaviours being added to a robot, the standard wander mode, and the standard teleop mode. The wander mode is activated, thus making it the "default" behaviour of the robot. Note that the letters "w" and "t" are associated with the behaviours, so the key handler can be used to switch between them. Also note that in this instance the robot object is "added" to the behaviour, rather than the other way round.

```
robot.lock();
ArModeWander wander(&robot, "wander", 'w', 'W');
ArModeTeleop teleop(&robot, "teleop", 't', 'T');

wander.activate();
robot.unlock();
```

5.1.4 Generic Sensor Initialisation

There are two main types of sensor, ranged and non-ranged. Ranged sensors, (sonar, lasers and, within the ARIA environment, bumpers), are added by creating an object of the appropriate type, and adding the object to the robot, using the *addRangeDevice* command. The example below illustrates a sonar being added to the robot.

```
 ArSonarDevice sonarDev;

robot.addRangeDevice(&sonarDev);
```

Non-ranged sensors are added by creating an object of the appropriate type, and adding the ROBOT to the OBJECT. In some cases this is done as part of the class initialisation. Below is the command to add a gyro to the robot.

```
 ArAnalogGyro gyro(&robot);
```

5.1.5 Laser Initialisation

Lasers are a ranged sensor, so are added using the *addRangeDevice* command. However, lasers require additional initialisation. The laser control object, *ArSick* is explicitly added to the ArSimpleConnector object used by the robot. This means that the control scheme used to operate the laser is also specified separately. Again, for safety purposes, an asynchronous control technique is recommended. Once attached to a connector, the laser can then be attached to the robot and used in the same way as a sonar sensor.

```
ArSick sick;


sick.runAsync();

if (!connector.connectLaser(&sick)) {
    //Some error message
    Aria::exit(2);
}

robot.addRangeDevice(&sick);
```

5.2 General Movement

As well as setting movement options in the form of an *ArActionDesired* from an *ArAction* it is also possible to use methods of the *ArRobot* object to manipulate the actuators directly. Common commands include:

- *ArRobot::setVel(double vel)* - Sets the forward speed of the robot
  ArRobot::setVel2(double vel1, double vel2) - Sets the individual speeds of the motors
- *ArRobot::setHeading(double heading)* - Sets the angle the robot should turn to (degrees)
- *ArRobot::setRotVel(double vel)* - Sets the rotational speed of the robot

For a full list, see [1].

5.3 Acquiring Sensor Data

Acquiring sensor data is largely specific to the type of sensor. However, a useful command to use is:

`ArRobot::checkRangeDevicesCurrentPolar(double start, double end)`

This command will check all the robot's ranged sensors that cover the specified range of angles, (specified in degrees) and return the smallest value, (i.e. the closest object.) There are however, two important issues to consider when using this command. Firstly, the command always assumes a counter-clockwise rotation from the start angle to the end angle, (specified in the range $\pm 180^o$ where positive numbers denote a counter-clockwise rotation) so it is important to ensure that the angles are provided in the correct order. Secondly the distance given is the distance from the point assumed to be in the centre of the robot. To acquire the absolute distance between the detected object and the robot itself, you should subtract the result of *ArRobot::getRobotRadius()* from the answer.

5.4 5DOF Arm Control

Controlling the 5DOF Arm via ARIA requires the use of the *ArP2Arm* class. The arm must be initialised prior to use, and the safety instructions for the arm recommend that the arm be placed into the "home" position before activation, to prevent sudden, potentially-damaging, movements in the joints. In order to initialise the arm, the *ArP2Arm* object must be attached to an *ArRobot* object, that has already been connected and placed into asynchronous communication mode. The first thing that should be done after attaching the arm to a robot is to power the arm on. *ArP2Arm::powerOn()* by default inserts a two second wait after power is applied to arm, to allow the initial vibrations to settle. This can be overridden by explicitly setting the "sleep" flag to false, (*ArP2Arm::powerOn(false)*) but this is not recommended. Explicitly un-initialising the arm will mean that it will automatically return to the home state before shutting down. Below is an example of the commands used to control a 5DOF arm.

```
ArP2Arm myArm;
//It is assumed that the robot is already
//connected and running asynchronously
myArm.setRobot(&robot);
myArm.init(); myArm.powerOn();
//This will move the first joint, (1-based index)
//to the 45 degree position, using the default velocity
myArm.moveTo(1, 45, 0);
```

It is of note that if the gripper is held in the shut position for more than 10 minutes it can cause damage to the servo-mechanism that drives it. By default, ARIA will keep track of the time the gripper has been shut for and should open it when the danger time approaches. However, this has not been tested by the author and it is advised that gripper use be minimised.

5.5 Miscellaneous

When all initialisation is complete, a call to *ArRobot::waitForRunExit()* will prevent the program from exiting until a termination signal is received from the robot controller. If a key handler is associated with the robot, you can force a termination signal, by hitting escape.

6 Troubleshooting

I didn't understand a word of that - where can I get more help?

Try [1] or the ARIA newsgroup. At the time of writing, you could find the link to the user groups here: `http://robots.mobilerobots.com/`.

6.1 Linux Only

When I run my program I get strange segmentation faults

There are two main causes for segmentation faults from the ARIA library. Firstly, try rebuilding your code on the target machine, rather than building it elsewhere and moving the executable. This resolves many segmentation fault issues. Secondly, the ARIA library itself may not be fully installed properly. Try performing the following steps as root to rebuild it from source.

```
cd /usr/local/ARIA
make clean
make
```

References

[1] ARIA Overview (html), Available with the standard install of ARIA (index.html)

A Example Program - based on the original ARIA *demo* program

This program implements a program similar to the original ”demo” program, distributed with the Aria library. It has the original ”wander mode” and the original ”teleop mode”. An additional custom mode for following the right hand wall of a pen is also added using the standard architecture.

A.1 RobotTest.cpp

This is the main function of the program. It initialises the robot and adds all of the sensors and behaviours. A function ”msg” is implemented to facilitate pumping useful feedback to the user.

```
#include "Aria.h"

#define LASER

//output header
#include "stdio.h"

//Custom mode
#include "armodewallfollow.h"

//Too lazy to write \n all the time
void msg(char* opMesg) {
         printf(opMesg);
         printf("\n");
}

int main(int argc, char* argv[]) {
         int number;

         msg("Initialising the robot");

         //Initiate the library - doing this at the beginning and "aria::exit"
at the end, seems to stop certain problems
         //in the destructors of some of the objects
         Aria::init();

         msg("Creating an argument parser");
         //An encapsulation of the valid input arguments to an ARIA program
         ArArgumentParser parser(&argc, argv);

         msg("Creating a connector");
         //A means of connecting to the robot
         ArSimpleConnector connector(&parser);

         msg("Creating a robot");
         //The main robot class
         ArRobot robot;


#ifdef LASER
         // a laser in case one is used
         ArSick sick;
#endif

         msg("Creating a sonar device");
         //Create a sonar device
         ArSonarDevice sonarDev;

         //Assign the defaults required to connect to the simulation
         msg("Assigning default values");
         parser.loadDefaultArguments();

         //This will parse the arguments
         msg("Parsing default values");
         if (!connector.parseArgs()) {
                 msg("Unable to identify default settings");
                 scanf("%d",&number);
                 exit(1);
         }

         // a key handler so we can do our key handling
         msg("Creating a key handler");
         ArKeyHandler keyHandler;

         msg("Attaching key handler");
         // let the global aria stuff know about it
         Aria::setKeyHandler(&keyHandler);
```

```
        // attach to the robot
        robot.attachKeyHandler(&keyHandler);
        msg("You may press escape to exit");

        //Add the sonar to the robot
        msg("Attaching sonar device");
        robot.addRangeDevice(&sonarDev);
#ifdef LASER
        // add the laser to the robot
        robot.addRangeDevice(&sick);
        // add a gyro, it'll see if it should attach to the robot or not
        ArAnalogGyro gyro(&robot);
#endif

        //Connect to the robot
        msg("Connecting to the robot");
        if (!connector.connectRobot(&robot)) {
                msg("Unable to connect to simulation");

                scanf("%d",&number);
                exit(1);
        }

        // start the robot running, true so that if we lose connection the run
stops
        msg("Run the robot!");
        robot.runAsync(true);
#ifdef LASER
        // set up the laser before handing it to the laser mode
        sick.runAsync();

        // connect the laser if it was requested
        if (!connector.connectLaser(&sick))
        {
                printf("Could not connect to laser... exiting\n");
                Aria::exit(2);
        }
#endif

        //Prevent anyone from tampering whilst we set up the robot
        msg("lock the robot!");
        robot.lock();
        //SET UP ROBOT HERE

        // turn on the motors
        msg("Turn on the motors");
        robot.comInt(ArCommands::ENABLE, 1);

        ArModeWander wander(&robot, "wander", ’w’, ’W’);
        ArModeTeleop teleop(&robot, "teleop", ’t’, ’T’);
        ArModeWallFollow follow(&robot, "follow", ’f’, ’F’);

        //The default behaviour
        wander.activate();

        // turn on the motors
        robot.comInt(ArCommands::ENABLE, 1);

        msg("Unlock the robot");
        robot.unlock();

        robot.waitForRunExit();

        Aria::exit(0);
}
```

A.2 ArActionWallFollow.h

This file implements the class that controls the wall following behaviour. It is too high-level to be part of a subsumption architecture, so is the only action within the "wall follow" action group. It is used here merely as a proof of concept.

```
#ifndef ARACTIONWALLFOLLOW #define ARACTIONWALLFOLLOW

//Standard aria type definitions
#include "ariaTypedefs.h"

//The ArAction class, that this class will inherit from
#include "ArAction.h"

//Thresholds for being "too near" to a wall
#define TOO_NEAR_UPPER           500 #define NEAR_UPPER
1000

#define TOO_NEAR_UPPER_B         200 #define NEAR_UPPER_B
500

//Rotational speed limits
#define CLOCK_MAX                -40 #define CLOCK_MIN
-10 #define CLOCK_MID                   -20 #define COUNTER_MAX
40 #define COUNTER_MIN               20 #define NO_ROT
0

//Translational speed limits
#define FORWARD_MIN              100 #define FORWARD_MAX
200 #define STOP                        0 #define BACKWARD_MIN
-100


//The wall following class
class ArActionWallFollow : public ArAction {

public:
         //Constructor - do standard init, specifying the purpose of the
function
         ArActionWallFollow(const char *name = "Follows the right-hand wall"):
  ArAction(name, "follows the right hand wall") {
```

```
                //These angle selections basically mean that "right" and
"left" have four sensors each, (two from the front and two from
the back
                //The "front" and "back" are also given four sensors each
                FRONT.startAngle = -22;
                FRONT.endAngle = 22;

                LEFT.startAngle = 22;
                LEFT.endAngle = 112;

                BACK.startAngle = 112;
                BACK.endAngle = -112;

                RIGHT.startAngle = -112;
                RIGHT.endAngle = -22;

                //This simply allocates memory to the message buffer used for
giving feedback
                opMessage = (char*)malloc(255*sizeof(char));
        }

        //Destructor - free the message buffer's allocated memory
        ~ArActionWallFollow() {
                free(opMessage);
        }


        //The "firing" routine
        ArActionDesired* fire(ArActionDesired currentDesired) {
                FUZZY_DISTANCE leftDist, rightDist, frontDist, backDist;

                //Gather the sensor data
                leftDist = GetReading(LEFT);
                rightDist = GetReading(RIGHT);
                frontDist = GetReading(FRONT);
                backDist = GetReading(BACK);

                //Reset my output
                myDesired.reset();
                situ = UNDECIDED;
```

```
                 //If too close on two opposing walls, just stop - there may
not be room to manouvre
                 if (
                         ((leftDist == TOO_NEAR) && (rightDist == TOO_NEAR))
                         ||
                         ((frontDist == TOO_NEAR) && (backDist == TOO_NEAR))
                         )
                 {
                         myDesired.setRotVel(0);
                         myDesired.setVel(0);
                         situ = TRAPPED;
                 } else {
                         //Check for corner too close conditions
                         if ((frontDist == TOO_NEAR) && (leftDist == TOO_NEAR))
{
                                 //front-left
                                 myDesired.setRotVel(CLOCK_MAX);
                                 myDesired.setVel(BACKWARD_MIN);
                                 situ = CORNER_FL;
                         } else if ((frontDist == TOO_NEAR) && (rightDist ==
TOO_NEAR)) {
                                 //front-right
                                 myDesired.setRotVel(COUNTER_MAX);
                                 myDesired.setVel(BACKWARD_MIN);
                                 situ = CORNER_FR;
                         } else if ((backDist == TOO_NEAR) && (leftDist ==
TOO_NEAR)) {
                                 //back left
                                 myDesired.setRotVel(CLOCK_MAX);
                                 myDesired.setVel(STOP);
                                 situ = CORNER_BL;
                         } else if ((backDist == TOO_NEAR) && (leftDist ==
TOO_NEAR)) {
                                 //back-right
                                 myDesired.setRotVel(COUNTER_MIN);
                                 myDesired.setVel(FORWARD_MIN);
                                 situ = CORNER_BR;
                         } else {
                                 //Check for single wall too close conditions
```

```
                                if (frontDist == TOO_NEAR) {
                                //front
                                        myDesired.setRotVel(COUNTER_MAX);
                                        myDesired.setVel(BACKWARD_MIN);
                                        situ = FRONT_CLOSE;
                                } else if (backDist == TOO_NEAR) {
                                        //back
                                        myDesired.setRotVel(NO_ROT);
                                        myDesired.setVel(FORWARD_MAX);
                                        situ = BACK_CLOSE;
                                } else if (leftDist == TOO_NEAR) {
                                        //left
                                        myDesired.setRotVel(CLOCK_MAX);
                                        myDesired.setVel(STOP);
                                        situ = LEFT_CLOSE;
                                } else if (rightDist == TOO_NEAR) {
                                        //right
                                        myDesired.setRotVel(COUNTER_MIN);
                                        myDesired.setVel(FORWARD_MIN);
                                        situ = RIGHT_CLOSE;
                                } else {
                                        //Acceptable single wall conditions
                                        if (rightDist == ACCEPTABLY_NEAR) {
                                        //Acceptably close to right wall

                                                //Check for imminent front wall
collision

                                                if (frontDist ==
ACCEPTABLY_NEAR) {
                                      myDesired.setRotVel(COUNTER_MAX);
                                      myDesired.setVel(FORWARD_MIN);
                                      situ =
RIGHT_ACCEPTABLE_FRONTCLOSE;
                                                } else {
                                                myDesired.setRotVel(CLOCK_MIN);
                                                myDesired.setVel(FORWARD_MAX);
                                                situ =
RIGHT_ACCEPTABLE_FRONTFAR;
                                                }
                                        } else if (frontDist ==
ACCEPTABLY_NEAR) {
```

```
                                                    //Acceptably close to front
wall
                                                  myDesired.setRotVel(COUNTER_MIN);
                                                  myDesired.setVel(STOP);
                                                  situ =
FRONT_ACCEPTABLE;
                                            } else if (leftDist ==
ACCEPTABLY_NEAR) {
                                                    //Acceptably close to left
wall
                                                    myDesired.setRotVel(CLOCK_MAX);
                                                    myDesired.setVel(STOP);
                                                    situ = LEFT_ACCEPTABLE;
                                            } else if (backDist ==
ACCEPTABLY_NEAR) {
                                                    //else acceptably close to
back wall
                                                    myDesired.setRotVel(CLOCK_MID);
                                                    myDesired.setVel(STOP);
                                                    situ = BACK_ACCEPTABLE;
                                            } else if (
                                                    (backDist == FAR_AWAY)
                                                    &&
                                                    (frontDist == FAR_AWAY)
                                                    &&
                                                    (leftDist == FAR_AWAY)
                                                    &&
                                                    (rightDist == FAR_AWAY)
                                                    ) {
                                            //far from all walls
                                                    myDesired.setRotVel(CLOCK_MIN);
                                                    myDesired.setVel(FORWARD_MAX);
                                                    situ = OPEN_ROAD;
                                            }
                                    }
                            }
                    }

                    if (situ == UNDECIDED) {
                            //not thought about this scenario
                            myDesired.setRotVel(NO_ROT);
```

```
                        myDesired.setVel(STOP);
                        situ = NOT_TAUGHT;
                }

                return (&myDesired);
        }

        //For now just blindly return the desired structure.  Should really
make that a pointer, to facilitate passing NULL
        ArActionDesired *getDesired(void) { return &myDesired; }

        //Returns a plain text explanation about what's going on!
        char *GetExplanation() {
                switch (situ) {
                        case TRAPPED:
                                opMessage = "Too close to two opposing walls,
stopping for safety";
                                break;

                        case NOT_TAUGHT:
                                opMessage = "I was not told what to do in this
situation!";
                                break;

                        case CORNER_FR:
                                opMessage = "Too close to front-right corner";
                                break;

                        case CORNER_FL:
                                opMessage = "Too close to front-left corner";
                                break;

                        case CORNER_BR:
                                opMessage = "Too close to back-right corner";
                                break;

                        case CORNER_BL:
                                opMessage = "Too close to back-left corner";
                                break;

                        case FRONT_CLOSE:
                                opMessage = "Too close to the front wall";
                                break;

                        case BACK_CLOSE:
                                opMessage = "Too close to the back wall";
                                break;

                        case LEFT_CLOSE:
                                opMessage = "Too close to the left wall";
                                break;

                        case RIGHT_CLOSE:
                                opMessage = "Too close to the right wall";
                                break;

                        case FRONT_ACCEPTABLE:
                                opMessage = "Front wall is acceptably close -
following";
                                break;

                        case BACK_ACCEPTABLE:
                                opMessage = "Back wall is acceptably close -
following";
                                break;

                        case LEFT_ACCEPTABLE:
                                opMessage = "Left wall is acceptably close -
following";
                                break;

                        case RIGHT_ACCEPTABLE_FRONTCLOSE:
                                opMessage = "Turning to avoid iminent wall
direction change";
                                break;

                        case RIGHT_ACCEPTABLE_FRONTFAR:
                                opMessage = "Following right wall";
                                break;

                        case OPEN_ROAD:
                                opMessage = "Searching for a wall";
                                break;

                        default:
                                opMessage = "Eeek - I don't know what I'm
thinking!";
                                break;
                }

                return(opMessage);
        }

private:
        //Encapsulates a direction
        typedef struct {
                double startAngle;
                double endAngle;
        } DIRECTION;

        //Not really "fuzzy" in its truest sense, but it'll do!
        //Just for the purposes of classfiying the distances
        typedef enum {
                TOO_NEAR,
                ACCEPTABLY_NEAR,
                FAR_AWAY
        } FUZZY_DISTANCE;

        //Situation enumerations - see method "getExplanation" for a full
definition of each
        typedef enum {
                TRAPPED,
                NOT_TAUGHT,
                CORNER_FR,
                CORNER_FL,
                CORNER_BR,
                CORNER_BL,
                FRONT_CLOSE,
                BACK_CLOSE,
                LEFT_CLOSE,
                RIGHT_CLOSE,
                FRONT_ACCEPTABLE,
                BACK_ACCEPTABLE,
                LEFT_ACCEPTABLE,
                RIGHT_ACCEPTABLE_FRONTCLOSE,
                RIGHT_ACCEPTABLE_FRONTFAR,
                OPEN_ROAD,
                UNDECIDED
        } SITUATION;

        //The desired action
        ArActionDesired myDesired;

        //The robot's current situation
        SITUATION situ;

        char*   opMessage;

        //Simplifies code readability for the main algorithm
        DIRECTION FRONT;
        DIRECTION LEFT;
        DIRECTION RIGHT;
        DIRECTION BACK;

        //Useful function for polling the sensors and returning a
classification of where the nearest object in the given direction
is
        FUZZY_DISTANCE GetReading(DIRECTION dir) {
                //Value to be returned
                FUZZY_DISTANCE retVal;

                //The raw sensor data
                double realReading =
(myRobot->checkRangeDevicesCurrentPolar(dir.startAngle,
dir.endAngle) - myRobot->getRobotRadius());

                //Thresholds for classification
                double tooNearThresh, nearThresh;

                //Decided to use different classifications for the back sensor
data, as it is not as important to achieving the task
                //Is this the back?
                if (
                        (dir.startAngle == BACK.startAngle)
```

```
                                &&
                        (dir.endAngle == BACK.endAngle)
                        )

                {
                        //Use the front-specific classification thresholds
                        tooNearThresh = TOO_NEAR_UPPER_B;
                        nearThresh = NEAR_UPPER_B;
                } else {
                        //Use the general classification thresholds
                        tooNearThresh = TOO_NEAR_UPPER;
                        nearThresh = NEAR_UPPER;
                }

                //Classification
                if (realReading < tooNearThresh) {
                        //Reading is less than the "too near" threshold
                        retVal = TOO_NEAR;
                } else if (realReading < nearThresh) {
                        //Reading between the "too near" and the "near"
thresholds
                        retVal = ACCEPTABLY_NEAR;
                } else {
                        //Reading is above the "near" threshold
                        retVal = FAR_AWAY;
                }

                return(retVal);
        }
}; #endif
```

A.3 ArActionGroupFollow.h

This implements the action group that implements the wall following behaviour. For demonstration purposes, only the action defined in A.2 is added. Note that a priority of 100 is given to the action as it is added. An additional method was implemented to handle the passing of messages from the low-level action to the main function. This was simply for the purposes of user-feedback and debugging. The ”userTask” method is utilised to pump feedback messages from the low-level controller, about what state it is in.

```
#ifndef ARACTIONGROUPFOLLOW #define ARACTIONGROUPFOLLOW #include
"Aria.h" #include "aractionwallfollow.h"

class ArActionGroupFollow : public ArActionGroup { public:
         ArActionGroupFollow(ArRobot* robot)  : ArActionGroup(robot)
         {
                 myAction = new ArActionWallFollow("Right Wall Follow");
                 addAction(myAction, 100);
         }

         char *GetStatus() {
                 return(myAction->GetExplanation());
         }
protected:
         ArActionWallFollow* myAction;
}; #endif
```

A.4 ArModeWallFollow.h

This file implements the ”mode” for wall following. It simply allocates memory to the action group, and uses the base class activation checker to ensure that it is safe to run.

```
#ifndef ARMODEWALLFOLLOW #define ARMODEWALLFOLLOW #include
"Aria.h" #include "ArActionGroupFollow.h"

class ArModeWallFollow : public ArMode { public:
         /// Constructor

         ArModeWallFollow(ArRobot *robot, const char *name, char key, char
key2):
  ArMode(robot, name, key, key2),
          myGroup(robot)
         {
                 myGroup.deactivate();
         }

         /// Destructor
         ~ArModeWallFollow() {}

         void activate(void) {
                 if (!baseActivate())
                         return;
                 myGroup.activateExclusive();
         }

         void deactivate(void) {
                 if (!baseDeactivate())
                         return;
                 myGroup.deactivate();
         }

         void help(void) {
//                ArLog::log(ArLog::Terse, "Attempting to follow the right hand
wall");
//                printf(myGroup.GetStatus());
                 printf("The robot will now attempt to follow the right-hand
wall\n");
         }
         void userTask(void) {
                 printf(myGroup.GetStatus());
                 printf("\n");
         }
protected:
  ArActionGroupFollow myGroup;
}; #endif
```