GLAIR Agents on the iRobot Magellan Pro Robot SNeRG Technical Note 37

Isidore Dinga Madou

Department of Computer Science and Engineering
University at Buffalo, the State University of New York
Buffalo, N.Y. 14260-2000
id@buffalo.edu

December 27, 2004

Abstract

This paper explains the different steps to implement the mobility functions of the Magellan Pro robot and how to communicate with it through sockets using a Lisp client. The working environment is the University at Buffalo Computer Science and Engineering Unix platform and the Magellan Pro robot internal computer. C++ was used to implement the data structure that connects to the robot hardware. This study was conducted under Dr. Stuart Shapiro supervision.

Contents

I. Introduction

Magellan pro is an indoor mobile robot from iRobot Company. “The Magellan Pro is a commercial robot made by the iRobot Corporation. The robot is equipped with an on-board PC running Red Hat Linux. It is 40.6cm in diameter and 25.4cm tall and has 16 SONAR sensors, 16 IR sensors, and 16 bump switches. Initial communication and diagnostics are performed using a tuning knob and LCD screen on the robot itself. It can be controlled using a joystick as well as programs written with its own unique Mobility Robot Integration Software.”[1]

For the past two semesters, Spring and Fall 2004, my work consisted of designing and implementing the mobility functionalities of the iRobot Magellan Pro and building a bridge between the low level, C++, and high level, Lisp implementation.

II. Low level: C++

The Mobility Robot Integration Software can be implemented using Java or C++, I use C++ for all my low level implementation.

1. Getting started

a. login to the Magellan pro environment

From any UB CSE station, one can login into the Magellan robot by typing:

> ssh “userid”@oldred
>password:
> ssh “userid”@irobot
>password:

If the robot is not ON or not connected to the network, the following error message will be displayed: **ssh: connect to host irobot port 22: No route to host**

In case that the robot is ON and you got the above message, contact the cse-consult@cse.buffalo.edu for assistance.

b. loading the Mobility Robot Integration Software

[“userid”@irobot ~]$ source /home/mobility/mobility.csh

c. starting a session

[“userid”@irobot ~]$ name -i
[“userid”@irobot ~]$ startup

d. ending a session

[“userid”@irobot ~]$ name -k
[“userid”@irobot ~]$ jobs –l
[“userid”@irobot ~]$ kill “process number”. Kill all the process related to the robot.

2. Basic Commands

Before communicating with the robot, all the libraries needed from the Mobility Robot Integration Software have to be included in the data structure program that connects the robot to the hardware. All libraries that need to be included are listed in the robot manual [2]. A C++ implementation will start as follow:

```
#include "mobilitycomponents_i.h"
#include "mobilitydata_i.h"
#include "mobilitygeometry_i.h"
#include "mobilityactuator_i.h"
#include "mobilityutil.h"
```

All the robot parameters need to be initialized. Most of the initialization steps are described in the robot manual [2]. This initialization connects the data structures and the robot hardware (sonar, wheel, camera…). The following code is a modified version of a code implemented in Nayak et all [3].

```
/ Look for robot name option so we know which one to run.
robotName = mbyUtility::get_option(argc,argv,"-robot");

if (robotName == NULL)
{
   fprintf(stderr,"Need a robot name to use.\n");
   return -1;
}
// All Mobility servers and clients use CORBA and this initialization
// is required for the C++ language mapping of CORBA.
pHelper = new mbyClientHelper(argc,argv);


// Build a pathname to the component we want to use to get sensor data.
sprintf(pathName,"%s/Sonar/Segment",robotName); // Use robot name arg.

// Locate the component we want.
ptempObj = pHelper->find_object(pathName);

// Request the interface we want from the object we found
try
{
   pSonarSeg     =     MobilityGeometry::SegmentState::_narrow(ptempObj);
//,env);
}
catch (...)
{
   return -1; // We're through if we can't use sensors.
}

// Build pathname to the component we want to use to drive the robot.
sprintf(pathName,"%s/Drive/Command",robotName); // Use robot name arg.

// Locate object within robot.
ptempObj = pHelper->find_object(pathName);

// We'll send two axes of command. Axis[0] == translate, Axis[1] ==
rotate.
```

```
OurCommand.velocity.length(2);


// Request the interface we need from the object we found.
try
{
    pDriveCommand = MobilityActuator::ActuatorState::_duplicate(
            MobilityActuator::ActuatorState::_narrow(ptempObj));
}
catch (...)
{
    return -1;
}
return 1;
```

I needed to create three methods and use two constants to successfully implement the basic mobility functions: move, stop, delay, MOVE_SPEED, and TURN_SPEED.

```
move(float speed,float angle){
  OurCommand.velocity[0] = speed;
  OurCommand.velocity[1] = angle;
  pDriveCommand->new_sample(OurCommand,0);
}


stop(){
   OurCommand.velocity[0] = 0;
   OurCommand.velocity[1] = 0;
   pDriveCommand->new_sample(OurCommand,0);
}

delay(int t){
  omni_thread::sleep(t);
}
```

a. Move Forward

Move the robot forward and stop.

```
moveForward(){
  move(MOVE_SPEED,0);
  delay(time);
  stop();
}
```

b. Move backward

Move the robot backward and stop.

```
moveBackward(){
  move(-MOVE_SPEED,0);
  delay(time);
  stop();
}
```

c. Turn Left

Turn the robot to the left with a 90 degree angle and stop.

```
turnLeft(){
  move(0,TURN_SPEED);
  delay(time);
  stop();
}
```

d. Turn Right

Turn the robot to the right with a 90 degree angle and stop.

```
turnRight(){
  move(0,-TURN_SPEED);
  delay(time);
  stop();
}
```

3. Reading sonars

I use the sonars to navigate the robot without bumping into objects or walls. They are the senses of the robot for an efficient navigation.

a. Reading one sonar

This function is useful to read specific sonar that we need for a given application.

```
float readSonar(int s1){
  pSegData = pSonarSeg->get_sample(0);
   return  sqrt((pSegData->org[s1].x - pSegData->end[s1].x)*
                (pSegData->org[s1].x - pSegData->end[s1].x)+
                (pSegData->org[s1].y - pSegData->end[s1].y)*
                (pSegData->org[s1].y - pSegData->end[s1].y));
}
```

b. Reading three sonars

To increase the accuracy of approximation, I used a set of three sonars to reflect the value read on each side of the robot. The method readSonars takes three integers, representing the set of sonars we are reading and return the average value of the three.

```
Float readSonar(int s1, int s2, int s3){
return  (readSonnar(s1)+readSonnar(s2)+readSonnar(s3))/3;
}
```

The sonars are numbered going counterclockwise from the front, starting at 0. Therefore, we have 0 to 15 sonars.

c. Reading front sonars

Return the average value of the three front sonars (0, 1, and 15).

```
float readSonarFront(){
 return readSonars(0,1,15);
}
```

d. Reading back sonars

Return the average value of the three back sonars (8, 7, and 9).

```
float readSonarBack(){
 return readSonars(8,7,9);
}
```

e. Reading left sonars

Return the average value of the three left sonars (4, 3, and 5).

```
float readSonarLeft(){
 return readSonars(4,3,5);
}
```

f. Reading right sonars

Return the average value of the three right sonars (12, 11, and 13).

```
float readSonarRight(){
 return readSonars(12,11,13);
}
```

4. Advanced commands

The advanced commands use one or two basic commands to execute an application.

The follow wall methods allow the robot to follow a wall on the right or left until there is an open door or an inside wall and stop.

The step in door function allows the robot to step in an open door and wait for instructions.

More advanced commands can be implemented using the basic commands. A virtual path for a given environment can be coded and used as an advanced command to allow the robot to execute that sequence of basic commands.

a. Follow Wall Right

```
/*Follow a wall at the right of the robot*/
void followRightWall(){
  if(!quit){  // quit is a Boolean variable declared in the main
program. It set to true if the initialization of parameters fails.

    float front= readSonarFront();// assign the front sonars value to
front
    float left= readSonarLeft();// assign the left sonars value to left
    float right= readSonar(12);// assign the most right sonar value to
 right
    float sleft=left;//starting value of the left sonar
    float sright=right;//starting value of the right sonars
    printf("left= %f, front=%f,right=%f \n",left,front,right);

/*start the follow wall on the right
SC_W is a constant declared in the main program. It is a threshold
value, representing a distance between the robot and a wall. SC_O is
the range within which the robot can go off from the starting position
on the left or right */
    while((front-SC_W>0)&&(right<4)&&(left-SC_W>0)){ /* as long as the
value of the front is greater that the threshold value, the right sonar
doesn't return and open range and the left sonar is greater than the
threshold value*/

      if(right<sright-SC_O){ /* check if the robot has moved further
right from the starting position, if yes, the robot is turned left to
go back to the starting position on the left*/
      turnLeft();
      printf("from right= %f, start=%f\n",right,sright);
      }
      if(right>sright+SC_O){ /* check if the robot has moved further
left from the starting position, if yes, the robot is turned right to
go back to the starting position on the right*/

      turnRight();
      printf("from left right= %f, start at =%f\n",right,sright);
      }
      // the two checks above keep the robot moving on a more or less
straight line.

      delay(0.001);
      moveForward();// move the robot forward
      delay(0.01);
      front= readSonarFront(); // get the new front sonars reading
      left= readSonarLeft();// get the new left sonar reading
      right= readSonar(12);// get the new most Right sonar reading
    }
    printf("exit with font= %f, right=%f\n",front-SC_W,right);
    stop();// when one of the loop condition is true, the robot is
stopped
    }
}
```

b. Follow Wall Left

```
/*Follow a wall at the left of the robot*/
Void followLeftWall(){
  if(!quit){  // quit is a Boolean variable declared in the main
program. It set to true if the initialization of parameters fails.
    float front= readSonarFront();// assign the front sonars value to
front
    float left= readSonar(4);// assign the most left sonar value to
front
    float right= readSonarRight();// assign the right sonars value to
front
     float sleft=left;//starting value of the left sonar
    float sright=right;//starting value of the right sonars

    printf("left= %f, front=%f,right=%f \n",left,front,right);
/*start the follow wall on the left
SC_W is a constant declared in the main program. It is a threshold
value, representing a distance between the robot and a wall. SC_O is
the range within witch the robot can go off from the starting position
on the left or right */

    while((front-SC_W>0)&&(left<4)&&(right-SC_W>0)){ /* as long as the
value of the front is greater that the threshold value, the left sonar
doesn't return and open range and the right sonar is greater than the
threshold value*/

      if(left<sleft-SC_O){ /* check if the robot has moved further left
from the starting position, if yes, the robot is turned right to go
back to the starting position on the right*/

      turnRight();
      printf("from left= %f, start at =%f\n",left,sleft);
      }

      if(left>sleft+SC_O){  /* check if the robot has moved further
right from the starting position, if yes, the robot is turned left to
go back to the starting postion on the left*/

      turnLeft();
      printf("from right= %f, start=%f\n",left,sleft);
      }
      // the two checks above keep the robot moving on a more or less
straight line.
      delay(0.001);
      moveForward();//move the robot forward
      delay(0.01);
      front= readSonarFront(); // get the new front sonars reading
      left= readSonarLeft();// get the new left sonar reading
      right= readSonar(12);// get the new most Right sonar reading
    }
    printf("exit with font= %f, left=%f\n",front-SC_W,left);
    stop();// when one of the loop condition is true, the robot is
stopped
   }
}
```

c. Step in a door

```
/*Step into a door*/
void stepDoor(){
  if(!quit){// check if all the parameters are initialized correctly
    float front= readSonar(0); // assign the most front sonar value to
front
    float  left= readSonar(4); // assign the most left sonar value to
front

    float right= readSonar(12); // assign the most right sonar value to
front

    float sleft=left; //starting value of the left sonar
    float sright=right; //starting value of the right sonar
    int n=0;
    bool t=true;
    printf("left= %f, front=%f,right=%f \n",left,front,right);
    while(t && (front-SC_W>0)){

      // check the door frame, the current value read front left and
right have to be less than the starting position to be between the door
frame.
      if((left<sleft)&&(right<sright)){
      n=1;
      sleft=left;
      sright=right;
      }
      //check one open side of the door on the right or left. At least
one of the side is an open, greater that the value read in the door
frane.
      if((n==1)&&((left>sleft)||(right>sright))){
      t=false;
      printf("from right= %f, start=%f\n",right,sright);
      }
      // the following two checks keep the robot moving on a more or
less straight line.

      if(left<sleft-SC_W){
      turnRight();
      delay(0.001);
      }
      if(right<sright-SC_W){
      turnLeft();
      delay(0.001);
      }
      moveForward();// move the robot forward
      delay(0.01);
      front= readSonar(0); // get the new most Right sonar reading
      left= readSonar(4); // get the new most Right sonar reading
      right= readSonar(12); // get the new most Right sonar reading
   }
   printf("exit with left= %f, front=%f, right=%f\n",left,front,right);
   delay(15);//let the robot move forward further
   stop();// stop the robot
 }
}
```

5. Socket Connection

To be able to communicate with the robot in a higher level, I needed to build a bridge, between the low and high levels. I modified the socket connection described in MSocket class [4] and by Santore [5]. The classes msocket.h and msocket.cpp (appendix A) explain the MSocket class. The implementation was made in a way that two clients could interact with the robot using different port numbers. Although only one socket is needed, implementing two socket connections leaves the door open for multiple treads application. At least two different clients could interact with the robot.

a. Starting the server

To start the server, the following command line needs to be type in the folder containing the server application.

[“userid”@irobot ~]$ ./server 12 –robot Robot0 –port1 4500 –port2 4322

If the port numbers are available, the server will wait for the clients to connect before proceeding (for details refer to connection method, Appendix B).

```
"Server waiting for connections..."
```

b. Accepting the connection

Once a Lisp client is connected, the server accept the connection and the client can write in the socket and the server will read the command and execute it (appendix B).

```
"Client connection accepted..."
```

c. Reading through the socket

Depending on the Lisp implementation, the command can be typed or programmed by the operator. Therefore, primitive actions need to be defined at the Lisp level. At the server level, the reading steps are explained in Appendix B.

d. Executing the command

Once the connection is secure between the server and the client, all valid inputs from the client are executed by calling the execute method (Appendix C).

e. Closing the connection

When the command read through the socket is “quit”, the server closes the connection with the client.

III. High level: Lisp implementation

This implementation wouldn't be made possible without the contribution of Trupti D. Nayak who provided the Lisp implementation template. To test the correctness of our implementation, we built four primitive actions to interact with the robot: forward, backward, turnleft, and turnright. We modified the code described by Nayak et all [3]. Each primitive writes the labeled command to the socket using a defined primitive action "writeToSocket" once the connection has been established with "OpenConnection."

```
;;; Lisp file to start up socket connections
;;; between C++ on Magellan Pro and Lisp on another computer

;;; Trupti Devdas Nayak
;;; Started Nov 12th, 2004

(defvar *robotMachine* "irobot.cse.buffalo.edu"
  "Name of the robot machine")

(defun promptedReadWithDefault (prompt default)
  "Prompts user for input, if no input takes default value"
  (format t "~&~a [~a]:  " prompt default)
  (let ((inp (read-line)))
    (if (string= inp "")
        default
      (read-from-string inp))))

;;; Lisp Function to start up a socket connection to robotMachine
(defun OpenConnection (socketPort)
  (setf *robotMachine*
        (promptedReadWithDefault "What machine to connect to?"
"irobot.cse.buffalo.edu"))
  (socket:make-socket :remote-host *robotMachine* :remote-port socketPort))

;;; Lisp Function to write command-string to socket
(defun writeToSocket (command socketStream)
  (format nil "~&~a~%" command)
  (write-line command socketStream)
  (force-output socketStream)
  (format t "***** Moving ~a *****" command)
  (sleep 5))

;;; ProcessInput is the monitoring process which monitors input to the socket
(defun ProcessInput (example-socket)
  (let ((InString "")
        (socket-output-lock (mp:make-process-lock)))
    (loop do
      (mp:wait-for-input-available example-socket)
      (setf InString (string-trim '(#\null)(read-line example-socket)))
      (if (equalp InString "Quit")
          (close example-socket)))))
```

```
(defun setup ()
  (setf socketStream (OpenConnection 4500)))

(defun demo ()
  (writeToSocket "forward" socketStream)
  (writeToSocket "right" socketStream)
  (writeToSocket "right" socketStream)
  (writeToSocket "forward" socketStream)
  (writeToSocket "left" socketStream)
  (writeToSocket "left" socketStream)
  (writeToSocket "backward" socketStream))

(defun quit ()
  (writeToSocket "quit" socketStream))
```

IV. Difficulties

Designing and developing the mobility function of the Magellan Pro robot and building the bridge between the high and low level were not easy tasks. The main difficulties encountered were:

- The robot availability for testing. Most of the time, the robot was down for technical problem. Given the fact that the iRobot company doesn't make the Magellan Pro robot any more, we have very limited technical support from the company. Also, network problems and hard drive failure paralyzed the implementation process.
- Learning about sockets and the network communication sequence was quit a challenge. Fortunately, good literatures [4] and [5] were available and I could modified pervious code to complete my application
- I wasn't able to build the Lisp interface without the precious contribution of Trupti Nayak. Although I had the documentation on Lisp, I was not able to build the client by myself.

V. Summary

To sum up, the basic mobility function of the Magellan Pro robot are built and we can interact with the robot through sockets using a Lisp client. However, there are still a lot to do with the robot to be able to use all of its functionalities. One of the most important features of the robot is unexploited up to today, the camera. It will be nice to have the camera working. We will then be able to control the robot remotely by using its new vision capability.

Thank you to Dr. Stuart Shapiro for giving me the opportunity to work on the Magallan Pro robot and for his guidance.

References

[1] http://www.cs.rpi.edu/courses/spring02/robotic/handouts/lab.pdf

[2] http://www.cse.buffalo.edu/shapiro/Courses/CSE716/MobilityManRev4.pdf

[3] Trupti Devdas Nayak, Michael Kandefer, and Lunarso Sutanto, Reinventing the Reinvented Shakey in SNePS, SNeRG Technical Note 36, Department of Computer Science and Engineering, University at Buffalo, The State Universtiy of New York, Buffalo, NY, April 6, 2004.

[4] MSocket Class.
http://www.code.ghter.com/linux2.html

[5] John F. Santore, Multiprocessing, Semaphores and Networking with ACL, SNeRG Technical Note 33, Department of Computer Science and Engineering and SNePS Research Group, University at Buffalo, The State Universtiy of New York, Buffalo, NY, 2002.

Appendixes

These are program files used for the sockets communication.
The Msocket class codes are available in the UB cse network under the /projects/robot/Magellan/ directory.
Connection and execute are modified codes that I wrote for my implementation. The original code is available on /project/robot/Magellan/cserver.cpp, written by Michael Kandefer.
All the source codes that I use for this application can be found in the Magellan Pro robot hard drive under /id/Fall04/SnepsLog/ directory.

Appendix A

msocket.h{

```
#ifndef _MSOCKT_H_DEFINE

#define _MSOCKET_H_DEFINE

// Needed by MSocket class:

#include <ctype.h> // for isdigit

#include <sys/types.h> // for connect(), bind(), accept(),

#include <sys/socket.h> // for connect(), listen(), bind() accept(),

#include <netinet/in.h> // for ntons(), htonl(), etc...

#include <arpa/inet.h> // for inet_addre()

#include <unistd.h> // for close(), read(), write()

#include <netdb.h> // for gethostbyname

#include <sys/time.h> // for timeval struct, and select()

#include <string.h> // for memset()

#include <map> // for STL map class (Reference counting)

// The MSocket class

class MSocket

{

public:

MSocket( int nFamily=AF_INET, int nType=SOCK_STREAM, int nProtocol=0 );

MSocket( const MSocket& msSrc ); // Copy Constructor

~MSocket();

operator int() { return( m_nSocket ); }
```

```
int Close();

// Socket operations (Client)

int Connect( struct sockaddr * sapSrvrAdrs, int nLenAddr ) const;

int Connect( const char* szHostAddr, short sPort ) const;

// Socket operation (Server)

int Bind( struct sockaddr * sapSrvrAdrs, int nLenAddr ) const;

int Bind( int nPort, int nFamily=AF_INET, int nAddr=INADDR_ANY) const;

int Listen( int nBackLog=5 ) const;

int Server( int nPort ) const; // Helper Function

int Accept( MSocket& msrRet, struct sockaddr* sapClientAddr=0, int*
npLen=0 ) const;

// Atributes for normal- or exception-mode error reporting

static bool m_bThrowException;

static bool SetExceptions( bool bMode );

// Operators

MSocket& operator=( MSocket& msrSrc );

MSocket& operator=( int Socket );

// I/O functions

int SetTimeOut( int MicroSeconds );

int ReadLine( char *cpDest, int nSize, char Term='\n' ) const;

int WriteLine( const char* cpSrc, char Term='\n' ) const;

// Helper functions

void ThrowErrMsg( const char* cpMsg, int nCode=0 ) const;

void ThrowErrMsg( const char* cpMsg, const char* cpInfo ) const {
ThrowErrMsg( cpMsg, (int) cpInfo); }

static bool WasConTermErr( const char* ErrMsg );

protected:

int m_nSocket, m_nTimeOut;

int TimedOut() const;

// Support for reference counting, using STL map class
```

```
static map<int,int> RefCount;

void IncRef()

{ if( RefCount.find( m_nSocket )!=RefCount.end() )
RefCount[m_nSocket]++; else RefCount[m_nSocket]=1; }

void DecRef()

{ if( RefCount.find( m_nSocket )!=RefCount.end() ) RefCount[m_nSocket]-
-; }

bool GetRef()

{ return( RefCount.find(m_nSocket)!=RefCount.end()? RefCount[m_nSocket]
: 0 ); }

};

// Some prototypes for over-ridden operators, to make class look like a
cout/cin object

MSocket& operator<<( MSocket& msrOut, const char* cpStr );

MSocket& operator>>( MSocket& msrIn, char* cpStr );

#endif
}
```

msock.cpp

```
{
#include <iostream>

#include <stdio.h> // For sprintf

#include "msocket.h"

// Common error strings, to save some space

static char ErrNotInit[] = "Socket not initialized";

static char ErrConTerm[] = "Connection terminated";

// m_szErrMsg is not part of the class, because of the const functions
throwing exceptions

// might modify it. Size is probably overkill. Unfortunately, my g++
compiler doesn't

// support the sstream class.

static char m_szErrMsg[256];

// Statics for the class
```

```
bool MSocket::m_bThrowException = true;

map<int,int> MSocket::RefCount;

// Start of the MSocket class functions

MSocket::MSocket( int nFamily, int nType, int nProtocol )

{

m_nTimeOut=0;

// Try to intialize the MSocket, using the socket system call

m_nSocket = ::socket( nFamily, nType, nProtocol );

if( m_nSocket < 0 && m_bThrowException ) // If an error, and we should
throw it

ThrowErrMsg( "socket() call failed: %d", m_nSocket );

if( m_nSocket >= 0 )

IncRef();

}

MSocket::MSocket( const MSocket& msrSrc ) // Copy Constructor

{

m_nTimeOut=0;

m_nSocket = msrSrc.m_nSocket;

IncRef();

}

MSocket::~MSocket() //Destructor

{

if( m_nSocket >= 0 )

Close();

}

int MSocket::Close()

{

int Ret = m_nSocket;

if( Ret < 0 && m_bThrowException )
```

```
ThrowErrMsg( "Close called on a closed socket" );

if( Ret > 0 )

{

DecRef(); // Check reference count, if we are the last

if( !GetRef() ) // using the socket, then close it.

close( m_nSocket );

m_nSocket = -1;

}

return( Ret );

}

int MSocket::Connect( struct sockaddr * sapSrvrAdrs, int nLenAddr )
const

{

if( m_nSocket < 0 )

{

if( m_bThrowException ) // If not setup correctly

ThrowErrMsg( ErrNotInit );

return(-1);

}

// Call the connect system function to actually do connect.

int Ret = ::connect( m_nSocket, (struct sockaddr*)sapSrvrAdrs, nLenAddr
);

if( Ret < 0 && m_bThrowException ) // If there was an error, and we
should throw it.

ThrowErrMsg( "connect() failed: %d", Ret );

return( Ret );

}

int MSocket::Connect( const char* cpHostAddr, short sPort ) const

{

// Setup and init the sock_addr_in struct, needed by real connect
function
```

```
struct sockaddr_in SrvrAdrs;

memset( &SrvrAdrs, 0, sizeof( SrvrAdrs ) );

// If szHostAddr looks like '127.0.0.1' then it's easy, just

// call inet_addr to convert the string into a netword address ID
integer

if( isdigit(cpHostAddr[0]) )

SrvrAdrs.sin_addr.s_addr = inet_addr( cpHostAddr );

else

{

// Otherwise, it may be a host name. Get it's IP address and use that.

// For example, szHostAddr may be www.acme.com

struct hostent * pHostEnt = gethostbyname( cpHostAddr );

if( pHostEnt == NULL )

{

if( m_bThrowException )

ThrowErrMsg( "Unable to determine IP for %s", cpHostAddr );

return( -1 );

}

SrvrAdrs.sin_addr.s_addr = *(long*)pHostEnt->h_addr_list[0];

}

SrvrAdrs.sin_family = AF_INET;

// Call htons, to convert our local pc short to format compatible with
the 'net'

SrvrAdrs.sin_port = htons( sPort );

// finally, call the other version of MSocket::Connect, with the struct
we set up

return( Connect( (struct sockaddr*) &SrvrAdrs , (int)sizeof( SrvrAdrs )
) );

}

int MSocket::Bind( struct sockaddr* sapMyAddr, int nAddrLen ) const
```

```
{
if( m_nSocket < 0 ) // Some called Bind with a closed socket?
{
if( m_bThrowException ) // Should we throw exception on an error?
ThrowErrMsg( ErrNotInit );
return(-1);
}
// Call the bind system function to actually do connect.
int Ret = ::bind( m_nSocket, sapMyAddr, nAddrLen );
if( Ret < 0 && m_bThrowException ) // If there was an error, and we
should throw it.
ThrowErrMsg( "bind() failed: %d", Ret );
return( Ret );
}
int MSocket::Bind( int nPort, int nFamily, int nAddr ) const
{
struct sockaddr_in SrvrAdrs;
memset( &SrvrAdrs, 0, sizeof( SrvrAdrs ) );
SrvrAdrs.sin_family = nFamily;
SrvrAdrs.sin_addr.s_addr = htonl( nAddr );
SrvrAdrs.sin_port = htons( nPort );
// Here, we call our 'other' Bind member function
return( Bind( (struct sockaddr*)&SrvrAdrs, (int)sizeof( SrvrAdrs ) ) );
}
int MSocket::Listen( int nBackLog ) const
{
if( m_nSocket < 0 )
{
if( m_bThrowException )
```

```
ThrowErrMsg( ErrNotInit );

return( -1 );

}

// Call the system 'listen' function

int Ret = ::listen( m_nSocket, nBackLog );

if( Ret < 0 && m_bThrowException )

ThrowErrMsg( "listen() failed: %d", Ret );

return( Ret );

}

int MSocket::Server( int nPort ) const // Just a convenient function

{

int Ret;

Ret = Bind( nPort );

if( Ret >= 0 )

Ret = Listen();

return( Ret );

}

int MSocket::Accept( MSocket& msrRet, struct sockaddr* sapClientAddr,
int* npLen ) const

{

int MyLen;

struct sockaddr_in Client;

if( sapClientAddr == 0 ) // To make things easier on our caller, we can
create the struct

{

sapClientAddr = (sockaddr*)& Client;

npLen = &MyLen;

}

if( m_nTimeOut && TimedOut() )

return( -1 );
```

```
// Call the system 'accept' function
int nRet = ::accept( m_nSocket, sapClientAddr, (unsigned int*)npLen );
if( nRet < 0 && m_bThrowException )
ThrowErrMsg( "accept failed: %d", nRet );
msrRet = nRet;
return( nRet );
}
MSocket& MSocket::operator=( MSocket& msrSrc )
{
if( &msrSrc != this ) // Make sure caller didn't to X = X;
*this = msrSrc.m_nSocket;
return( *this );
}
MSocket& MSocket::operator=( int nSocket )
{
if( nSocket < 0 && m_bThrowException )
throw "operator= called with bad socket";
if( m_nSocket >= 0 )
Close();
m_nSocket = nSocket;
IncRef();
return( *this );
}
int MSocket::TimedOut() const
{
fd_set fdSet;
struct timeval TimeToWait;
TimeToWait.tv_sec = m_nTimeOut / 10000;
```

```
TimeToWait.tv_usec = m_nTimeOut % 10000;
FD_ZERO( &fdSet );
FD_SET( m_nSocket, &fdSet );
int Ret = select( m_nSocket+1, &fdSet, &fdSet, &fdSet, &TimeToWait );
if( Ret <=0 && m_bThrowException )
ThrowErrMsg( Ret==0?"Time out":"Select error: %d", Ret );
return( Ret==1?0:1 );
}
bool MSocket::SetExceptions( bool bMode )
{
bool Ret;
Ret = m_bThrowException;
m_bThrowException = bMode;
return( Ret );
}
int MSocket::SetTimeOut( int nMicroSeconds )
{
int Ret = m_nTimeOut;
m_nTimeOut = nMicroSeconds;
return( Ret );
}

int MSocket::ReadLine( char *cpDest, int nSize, char Term ) const
{
int ReadIn=0, Stat;
if( m_nSocket < 0 )
{
if( m_bThrowException )
ThrowErrMsg( ErrNotInit );
```

```
return( -1 );
}
while( ReadIn < nSize-2 )
{
if( m_nTimeOut && TimedOut() )
return( -1 );
Stat = read( m_nSocket, cpDest+ReadIn, 1 );
if( Stat <= 0 )
{
if( m_bThrowException )
ThrowErrMsg( Stat==0?ErrConTerm:"ReadLine error: %d", Stat );
return( Stat );
}
if( cpDest[ReadIn]==Term )
break;
ReadIn++;
}
if( Term != '\0' )
ReadIn++;;
cpDest[ReadIn]='\0';
return 1;
}
int MSocket::WriteLine( const char *cpSrc, char Term ) const
{
int Written=0, Len;
Len = strlen(cpSrc);
if( Term=='\0' ) // If terminator is '\0', include that in data out
Len++;
```

```
if( m_nSocket < 0 )
{
if( m_bThrowException )
ThrowErrMsg( ErrNotInit );
return( -1 );
}
while( Len>0 )
{
Written=write( m_nSocket, cpSrc, Len );
if( Written <=0 )
{
if( m_bThrowException )
ThrowErrMsg( Written==0?ErrConTerm:"Writing socket: %d", Written );
return( Written );
}
cpSrc += Written;
Len -= Written;
}
return( 0 );
}
void MSocket::ThrowErrMsg( const char* cpMsg, int nCode=0 ) const
{
sprintf( m_szErrMsg, cpMsg, nCode );
throw m_szErrMsg ;
}
bool MSocket::WasConTermErr( const char* ErrMsg )
{
return( strcmp(ErrMsg,ErrConTerm)==0);
}
```

```
// A cout 'look alike' function, for simplicty. ostream was not really
used.

MSocket& operator<<( MSocket& msrOut, const char* cpStr )

{

msrOut.WriteLine( cpStr );

return( msrOut );

}

// A cin 'look alike' function, for simplicity. istream is not really
used.

MSocket& operator>>( MSocket& msrIn, char* cpStr )

{

msrIn.ReadLine( cpStr, 0x0FFFF ); // No good number here, really

return( msrIn );

}
}
```

Appendix B

/*The connection method initializes the server for connection, checks the port for client connection and then reads the socket for any command written until the command read is “quit”.*/

```
Int connection(){
// variables to handle sonar watching
  pthread_t sonarScan;
  int rc;
  bool started = false;

// Main loop that checks for socket commands

try
{

   S.Server(thePort);

   cout << "Server waiting for connections..." << endl;

   while( 1 )
     {

       S.Accept(C);

       cout << "Client connection accepted" << endl;

       if( fork() == 0 ) // ID is 0 for child, non-0 for parent
       { // New child handles the socket here

         char Buffer[128];
         int  duration;
         char *command;
         char *temp;

         S.Close(); // Client doesn't need server socket anymore

         while( 1 )
           {
             // Read from client
             C >> Buffer;

             // Fill rest of array with null characters
             for(int i = strlen(Buffer)-1; i < 128; i++)
             Buffer[i] = '\0';

             cout << "The input is " << Buffer << endl;

             // Parse out the command
             command = strtok(Buffer, " ");
             cout << "The command is " << command << endl;
             if(strcmp(command, "quit")==0)
             {
            cout << "In quit." << endl;
            quit = true;
            omni_thread::sleep(5);
```

```
            break;
             }
             else
             {
               execute(command);
             }
           }

         break; // Terminates client, gets out of loop

       }
       else
         C.Close(); // Server doesn't need client socket anymore
     }
}
catch( char* Msg )
  {

    if( MSocket::WasConTermErr( Msg ) )
      cout << "Ok, no biggie, connection terminated" << endl;
    else
      cout << "Error: " << Msg << endl;
  }

 return(0);
}
```

Appendix C

/* The execute method runs the command read through the socket. If the command is not defined, the robot does nothing. It can be extended depending on the user need. We can add advanced commands on the list of commands.*/

```
Void execute(char *command){
// Take appropriate action based on the action passed in
  if(strcmp(command, "forward")==0)
    {
      moveForward();
    }
  else if(strcmp(command, "backward")==0)
    {

      moveBackward();
    }
  else if(strcmp(command, "right")==0)
    {
      turnRight();
    }
  else if(strcmp(command, "left")==0)
    {
      turnLeft();
    }
}
```