User's Manual

January 23, 1996

Software Version: 2.6

Part Number: DOC00004

NOMADIC
TECHNOLOGIES, INC.

Nomadic Technologies
2133 Leghorn Street
Mountain View, CA 94043
TEL 415.988.7200
FAX 415.988.7201
Email: nomad@robots.com

Where can I get help ?

1. By email: send a description of your problem, if possible with the source code of your program, to nomad@robots.com.

2. By phone: call +1 415 988 7200 and ask for Technical Support.

Where can I get software ?

For the convenience of timely software distribution, Nomadic has set up an FTP site for this and future Nomadic software releases. From this FTP, you can down load the most up-to-date software distributed by Nomadic.

In addition, we have setup a directory for you to up-load software that you want to share with other Nomad users.

To down load (or up-load) software from this FTP site, you can simply FTP to ftp.robots.com

```
Name: robots
Password: *****
```

(Password information given on request to Nomadic Customers)

Once you have logged in, cd to the “pub/files” directory. Within this directory, there is a NOMAD-README file and an AGREEMENT file. Please read these two files carefully. In addition, there are seven sub-directories. The NOMAD-README file provides descriptions of the sub-directories (what they contain) and instructions on how to obtain and to extract the software in these sub-directories.

If you have any questions regarding how to obtain the software or how to run the software, please email them to software@robots.com.

To order additional copies of this manual or other manuals, please call +1 415 988 7200 and ask for the Sales Department.

Thank you for purchasing a Nomadic Technologies product. The Nomad and Sensus products are warranted to the original purchaser, to be free from defects in materials and workmanship for a period of one year from the shipping date. During this period Nomadic Technologies, Inc. will repair or replace, at our discretion, any defective components.

This warranty does not apply to any Nomad or Sensus products which have been damaged by accident, abuse, negligence, improper use, power surges, acts of God or have been repaired, altered, or modified in any way by anyone other than Nomadic Technologies. This warranty does not apply to the batteries or antennae.

Nomadic Technologies, Inc. expressly disclaims and excludes all other warranties, express, implied, and statutory, including without limitation, the warranty of merchantability and fitness for a particular purpose.

Nomadic Technologies, Inc. expressly disclaims and excludes all liability for incidental and consequential damages, including lost profits or savings, and the cost of recovering or reprogramming lost data. The Customers maximum entitlement shall in no event exceed the cash value of the covered item(s) at the time of the item(s) breakdown.

If you have any questions or problems with your Nomad or Sensus products contact Nomadic Technologies Customer Service at 415/988-7200 for instructions.

In the event that service is required, after notifying Nomadic Technologies and receiving a RMA, ship your product, together with all accessories, in its original packaging, fully prepaid and insured, to Nomadic Technologies, Inc. Nomadic Technologies, Inc. is not responsible for any damages incurred during shipping. We will notify you of repair costs, if they are not covered by the warranty, before undertaking them and will notify you before return shipping your product. The customer is responsible for all shipping and shipping insurance costs.

Conventions

Here are the typographical conventions used in this manual:

1. `typewriter` characters denote user input at a terminal, as well as code examples, as in:

   ```
   machine:~/Nomad200/host$ Nserver my_world.setup my_robot.setup
   ```

2. ♣ is a remark, note, or tip, as in:

   ♣ *You can have up to 6 robots controlled simultaneously from the same server*

3. **!!!** denotes trouble shooting information, as in:

   !!! The robot does not move

   - Is the emergency stop released ?
   - Are the batteries in place ?

   ---

4. **WARNING** signals an important point, as in:

   WARNING RUNNING THE ROBOT WITH BATTERIES UNDER 11 VOLTS OF CHARGE MAY DAMAGE THE CPU

Contents

List of Figures

Chapter 1

Introduction

This User's manual describes the use of the Nomad mobile robot. It contains:

- A tutorial introduction to the robot use, including start-up procedures, the graphic interface and simulator, elementary programming and trouble shooting,
- A detailed description of the graphic interface and the associated robot simulator,
- An introduction to the programming language,
- A description of the robot advanced features: sensors usage, position attachment, packet communication, configuration files.

In addition to this User's Manual, the Language Reference Manual contains a detailed description of each robot command.

Chapter 2

Getting Started

This chapter is a general introduction to the basic operation of the Nomad 200. You will:

- Install batteries in the robot
- Boot up the robot
- Joystick the robot around
- Start a server
- Connect to the robot
- Move the robot from the server
- Write a simple program, and run it with the simulated and real robots

2.1 Installing the batteries

The Nomad is powered by a set of three batteries that together provide 432 watt-hours of usage. They are located at the very bottom of the robot, behind the lower bumper ring.

- If the joystick is plugged in, remove it before accessing the batteries.
- The bumpers are mounted to a cylindrical piece of metal surrounding the bottom of the robot. This part can slide upwards. Below the lower bumper is a red marker. Slide the cylinder upwards and secure it with the tuck that is located at the red marker.
- Raise the bottom cylinder of the robot.

- Slide the batteries into the battery compartments, making sure that the connections face the outside of the robot. Connect the batteries to the robot using the plug coming out of the compartment. Plugs are numbered: the plug number 18 is for the CPU battery; plugs numbered 17 and 19 are for the motor batteries.

  ♣ *The CPU battery, supplying power to the main processor and all the sensing boards, usually gets exhausted first.*

  A green light to the right of the battery is turned on if the batteries are connected correctly. Otherwise, check the connection and the fuse next to the light.

- Make sure that all batteries are pushed in all the way into their compartments and that the connectors do not stick out too far.

WARNING LOWERING THE CYLINDER ONTO THE CONNECTORS COULD CAUSE SERIOUS DAMAGE TO THE ROBOT

- Carefully lower the cylinder until it rests on the base of the robot.

- Robots of serial number over 53, and upgraded robots have an additional floor for two flat CPU batteries. To install these, remove the skin on each side, slide the batteries into the compartment, and connect them.

A single set of three regular batteries will last one to four hours hours, depending on which sensors are used. With the additional battery floor, the robot is able to run continuously using all its sensors during at least eight hours.

2.2 Battery maintenance

WARNING ALWAYS PAY ATTENTION TO THE CHARGE LEVEL OF YOUR BATTERIES !!!
RUNNING WITH UNDERCHARGED BATTERIES (EVEN ONE UNDERCHARGED BATTERY) MAY CAUSE ERRATIC ROBOT MOTIONS.

Please keep the "Batteries Rules" in mind:

- Keep two sets of batteries (the best is to label them, like "Set A" and "Set B"). Use one set while the other is on charge. Always use and recharge the batteries of a set together, do not mix them: one bad battery can ruin a set.
- If you have an auxiliary battery floor, make a subset of motor batteries (small batteries connected to plugs 17 and 19, lower floor), and a subset of CPU batteries (small battery connected to plug 18 in the lower floor, big batteries in the auxiliary floor). Label each subset batteries, use and charge them together; do not mix them.
- Never put one small battery alone on the charger.
- The charge is complete when the "Fast Charge" LED on the front panel of the charger goes off. If in doubt, check the voltage at each battery when installed and robot ON: the normal level is 12-13V, and it should NEVER go below 10.8V.
- Never leave the batteries connected in the robot when not in use. a long time.
- An alarm will beep whenever the CPU or the motor batteries are too low. If you hear the alarm, turn the robot off, install new batteries and reboot. This should be done within 5 minutes to avoid damaging the batteries.

Consistent use and recharge will yield approximately 200 charge cycles from your batteries. However, these batteries are NOT eternal (who is ?). If you notice that your batteries get exhausted very quickly, get hot when you are charging them, or do not charge, REPLACE THEM.

2.3 Starting the robot

- If the Nomad is not equipped with an internal hard drive, insert the boot disk into the disk drive.
- Turn the robot on by pressing the grey button on the front panel.

  Robots with a PC floor distribution board version 1.2 and above have an alarm that will beep when the power reaches 10.8 V. In that case, you should stop using the robot and recharge the batteries. The battery level can also be accessed from a robot program using `voltCpuGet` and `voltMotorGet` (see the Language Reference Manual). See section 2.9 for indications on how to recharge the batteries.
- The following events indicate a successful bootup:
  - If the robot contains a speech synthesis card a message will sound through the speaker
  - The motor light blinks
  - The CPU light turns on
  - All other lights turn off
- A successfull bootup in DOS will take roughly one minute. Linux takes about three minutes to boot.
- The robot is now operational and can be moved using the joystick.

WARNING Do not boot close to a wall !
After booting, the robot will test motor response by sending extremely small motion commands (so small that the robot won't noticeably move). However, if the encoders have been accidentally disconnected, the robot will move a few inches, with the possibility of hitting something. In this case contact Nomadic Technologies, Inc for assistance.

2.4 Moving the robot around using the joystick

The robot comes with a two-axes joystick depicted in Figure 2.1. It should be plugged into the 15-pin connector on the side of the robot. The joystick is to be used in conjunction with the two motion buttons in the top left corner. Always push the motion buttons BEFORE moving the joystick.

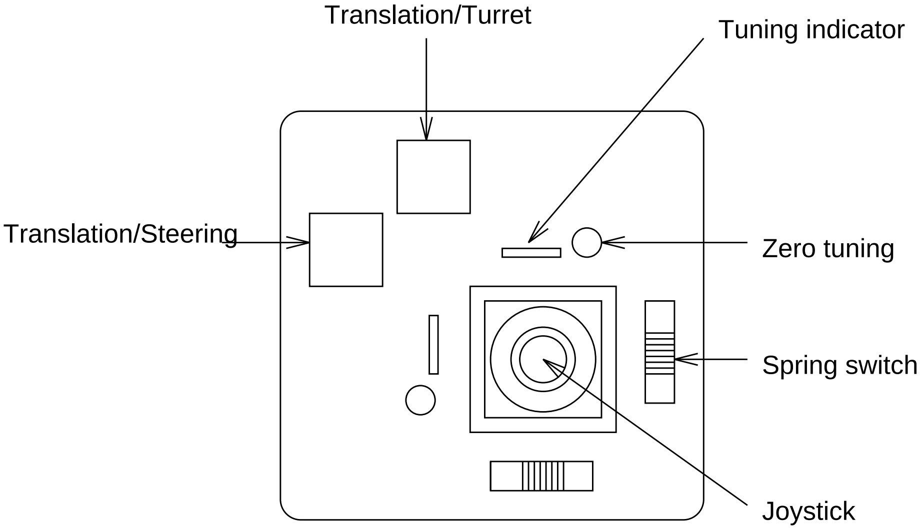

Figure 2.1: The Joystick

- While holding the left button down:
  - Pushing the joystick forward (backward) will move the robot forward (backward). The further you push, the faster the robot will go. Note that forward and backward are related to the current orientation of the wheels
  - Pushing the joystick left (right) will steer the wheels on the left (on the right). The further you push, the faster the robot will steer.
  - Pushing the joystick in a composite direction will both translate and steer, resulting in a somewhat circular motion.
- While holding the top button down:
  - Pushing the joystick forward (backward) will move the robot forward (backward) as with the left button.
  - Pushing the joystick left (right) will rotate the turret on the left (right).
  - Pushing the joystick in a composite direction will have the robot translating while rotating the turret.

♣ *You can move the robot while holding both buttons pressed. This way, the turret and the wheels will turn together at the same speed, giving you a visual indication of the steering.*

♣ *Contrary to popular belief, the bumpers are NOT wired to the motors: the robot will keep moving even if the bumpers are hit. However, when running the robot from a program, you can monitor the state of the bumpers and stop* *the robot if they are hit.*

The spring switches (see Figure 2.1) cause the joystick to go back to the zero (middle) position when released. If they are not engaged, the joystick will be free when released, sending motion commands to the robot. Since this may cause unwanted motions, it is highly recommended to always have the spring switches engaged on both axes.

The zero tuning is used to adjust the joystick. If the robot is still moving with the joystick in the zero position, turn the small knob until the robot is immobilized. The tuning indicator gives the direction of the adjustment.

!!! The robot doesn't move when you joystick it:

- Are the batteries present, connected, and charged (10.8 V minimum)?
- Is the joystick connected?
- Is the Emergency Stop button released?
- Is there a robot process running? If so, the MOTOR green light on the robot control panel should be flashing continuously. If not, restart the robot using the OFF button, then the ON button.

2.5 Starting a Server

The Server is a convenient way to send commands to the robot and to receive sensing data from the robot. It provides an elaborate graphic interface and simulation capabilities. You run the server as a separate process on a workstation. The server process communicates with the robot process through radio ethernet, using the TCP/IP protocol. To start a server on your workstation:

- Make sure that you have, in the same directory, the `Nserver` executable, the two configuration files `world.setup, robot.setup` and the license file `license.data`.
- Type the command:
  `Nserver`

The graphic interface should appear as shown in Figure 2.2. The xterm that you used to run `Nserver` will be used to display messages; it will be referred to as "the xterm" in the sequel. You should now see the following in the xterm:

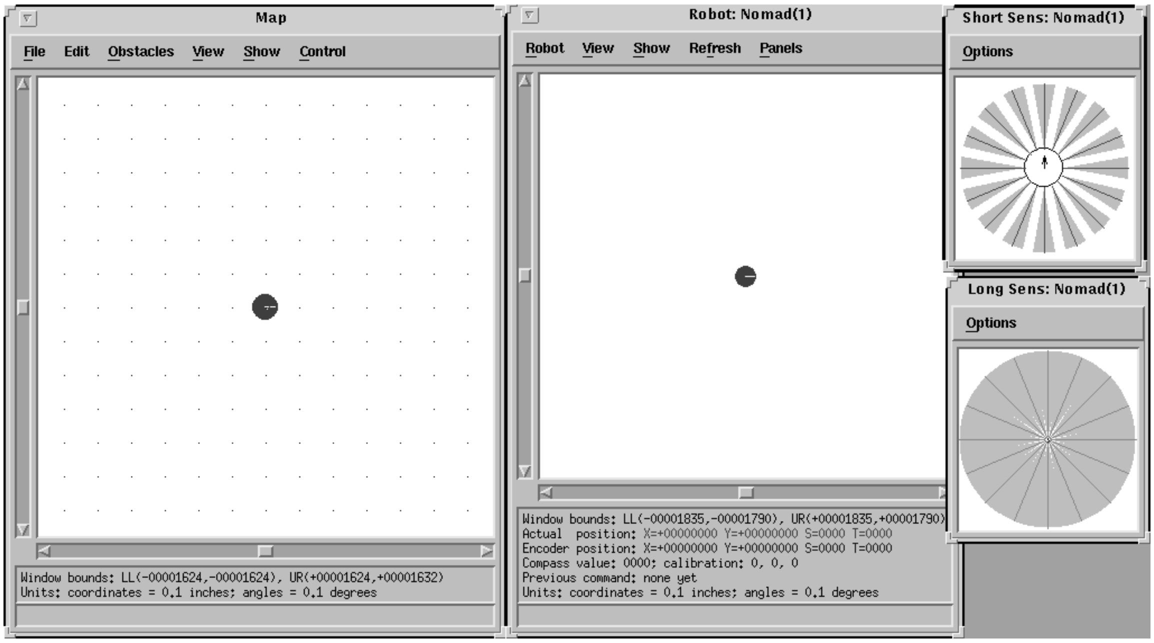

Figure 2.2: The entire graphic interface

```
================================================================
Nomadic Host Software Development Environment (Version 2.6.2)
Last revision date:  12-Aug-1995
Copyright 1992,93,94,95, Nomadic Technologies, Inc.
----------------------------------------------------------------
This software is licensed to:
Software Development
Nomadic Technologies, Inc.
Site License #10000
================================================================
Using server tcp port #7019.
```

!!! The command doesn't work. Possible causes:

- Nserver has not been compiled for your machine
  Check how to get the proper executable in the section "Where can I get help?" at the beginning of this manual. We currently support:
  - Silicon Graphics: MIPS/IRIX5.3
  - SUN: SPARC/SUNOS4.1, SOLARIS2.3
  - Linux 1.2.x systems/X11R5
- You do not have the proper license file for the machine you are running on
  Get the correct license file from Nomadic Technologies, Inc.
- You are not allowed to display on this machine. Check your `DISPLAY` environment variable
- Some configuration files are damaged or missing.
  Restore the original files from the distribution, or edit the files (refer to the section 5.3 in Chapter 5).

2.6 Connecting to the Robot

For safety reasons, the server is by default configured in simulation, which means that actions do not have physical consequences. You are now going to connect to the real robot.

- Make sure that the robot is ON.
- Select the ROBOT item in the robot window (entitled Robot:Nomad(1)) by clicking with the left button.
- Select REAL ROBOT and release the left button: the communication starts

The REAL ROBOT menu item turns into SIMULATED ROBOT: if you select it, the connection to the real robot will be closed, and your actions will be directed to the simulation again. When connected, you can check the information bar of the robot window, which should show the actual position grayed out (the actual position is a simulation concept; see Chapter 3). The xterm should look like the following:

```
Robot <-> Host TCP/IP communication setup
(machine nomad on port 65001)
```

!!! The server doesn't connect to the robot. Possible causes:

- The robot is OFF
- The RANGE/LAN Ethernet bridge is not in operation. Restart the bridge (it takes several minutes to boot).
- The machine name in `robot.setup` does not match the robot name. Edit the file `robot.setup`, paragraph `[connect]`, item `machine`, and type the correct name.
- The robot is too far away. The range limit within one single bridge is about 50 meters (150 feet) under normal conditions.
- This is the first connection to the robot using PC/NFS. Due to a bug in PC/NFS, you have to make several attempts (by repeatedly selecting REAL ROBOT in the ROBOT menu) to correctly initialize the communication.

2.7 Moving the robot

You are now going to initialize the robot, and to move it around using the soft joystick. Initialization, or "zeroing", resets the encoders to the "zero" position.

- Select the PANELS item in the robot window, and choose COMMAND LINE. The window of Figure 2.3 will appear.
- Select the `zr` (zeroing) button: `zr` appears in the command line.
- Click on Execute to start zeroing: the robot will execute zeroing motions.
- Select COMMAND LINE again from PANELS to close the window.

The zeroing process takes about 30 seconds and is exclusive: no other operation can be performed until the end of the zeroing.

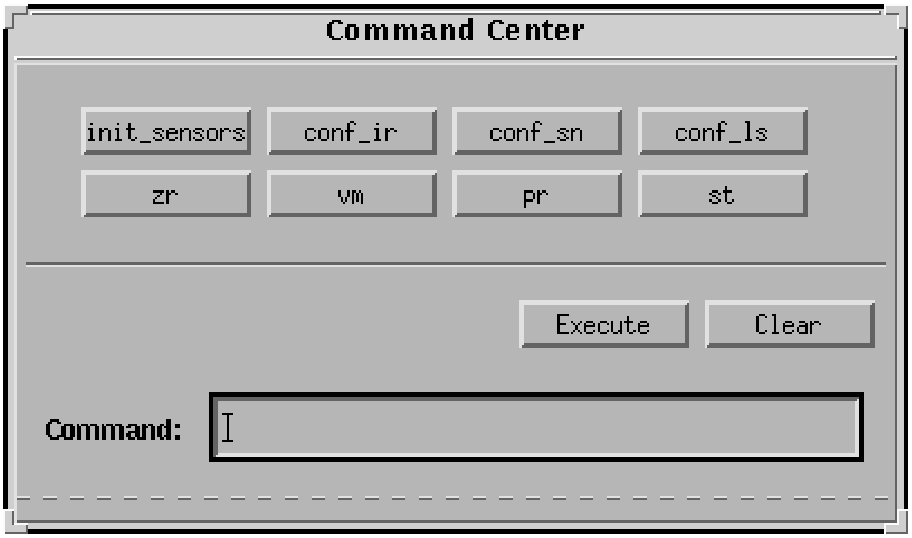

Figure 2.3: The Command Line panel

!!! The robot doesn't initialize. Check the following:

- the robot is ON.
- The Emergency Stop Button is released.
- The Server is connected to the robot (see previous section).

♣ *When the robot has been zeroed, the wheels and turret are aligned with orientation 0; the "front end" of the robot is opposite to the emergency stop button. This will be true as long as the turret and the wheels are always rotated together.*

After zeroing, you could move the robot by sending motion commands with the command line panel. However, it is usually easier to use the soft joystick:

- Select the PANELS item in the robot window, and choose JOYSTICK. The window of Figure 2.4 will appear. The crosshairs represents the two degrees of freedom of the physical joystick. The central dot is the joystick: the robot moves according to its position.
- Make sure that the robot is free to move and that no equipment (keyboard, monitor,...) is connected to it.
- With the left button, click and drag the dot to the middle of the top vertical line: the robot will move forward. Release the left button to stop.

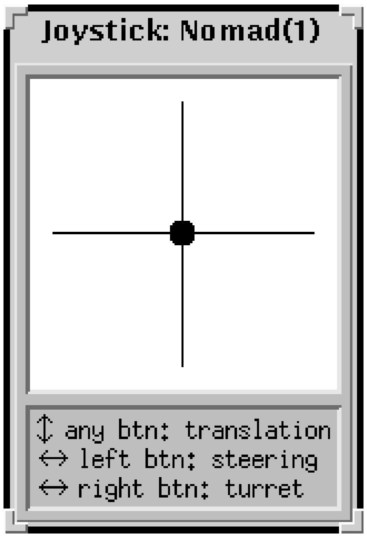

Figure 2.4: The soft joystick

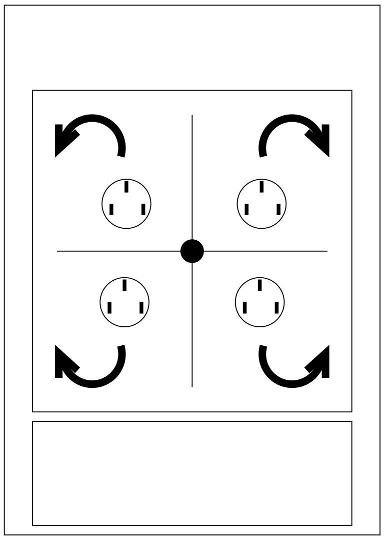

Figure 2.5: Combined motions

- With the left button, click and drag the dot to the middle of the left horizontal line: the robot will steer. Release the left button to stop.
- The further you move the dot from the center of the crosshairs, the faster will the robot go.
- As with the physical joystick, combined motions are possible, as illustrated in Figure 2.5.
- With the right button, you can translate the robot along the vertical axis like with the left button. However, the horizontal axis will rotate the turret.
- To get an idea of the robot motions, you can select SHOW in the robot window. Select then ROBOT TRACE, and choose SOLID. Move the robot: the path is shown. Select NONE to remove it.

♣ *A common mistake is to give an absolute meaning to the soft joystick directions, like "North", "East", etc. The motions are always relative to the current orientation of the wheels. Moving the dot to the top will move forward, which may be "South" if the robot is so oriented.*

♣ *If your mouse has three buttons, the middle button is equivalent to pushing both the left and right together, i.e. the vertical axis will translate, the horizontal axis will steer AND rotate the turret.*

The dedicated Short Sensor and Long Sensor windows display robot-centered sensor information. This is an instantaneous information. It is also possible to accumulate these data on the screen and to get a sensory image of the environment. This allows you to navigate from your workstation.

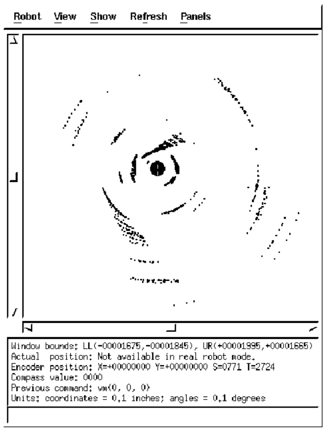

Figure 2.6: Surroundings of the robots as shown by traces

- In the robot window, select SHOW, and choose SONARS; dots representing sonar readings appear (default color is blue) in the robot window, around the robot.

- Select SHOW again, and choose INFRARED; dots representing infrared readings appear (default color is red).

- Using the soft joystick, rotate the turret by moving the dot on the horizontal axis, left or right: sensing data will be accumulated, and the surroundings of the room the robot is in will start to appear like in Figure 2.6

- To stop the accumulating of the data, select SHOW and SONARS, INFRARED again. To erase the accumulated sensory data, select REFRESH in the robot window, and choose ALL SENSOR HISTORY.

!!! Nothing appears, or strange data; check the following:

- Are the sonars and the infrared sensors ON? These sensors are started by default when booting up the robot. You can restart them by sending a `conf_sn` for sonars and `conf_ir` for infrared using the Command Center. See the Chapter 3 for a description of how to fill the fields, and the language reference manual for a description of the meaning of the fields.
- Nothing in the way of the sensors? (typically for infrared, bumpers in high position).
- Are the infrared correctly calibrated? Calibration values for the infrared are provided in the file `robot.setup`, paragraph `[infrared]`, item `calibration`, as a mapping intensity/distance in tenths of inches.
- Each sensor has its specificities. Infrared will be saturated by incandescent light or sun light, and won't see black objects. Sonars won't see smooth objects unless they are presenting a surface normal to the direction of the sound.
- Dots aligned along circles at some distance of the robot are NORMAL; when a sensor doesn't see anything, it returns a constant maximum distance value, whose accumulation produce circles.
- Sensor data is not displayed when the position of the robot is not available. If a program has set the sensor mask so as to exclude position, sensor readings are not printed.

2.8 Introduction to programming

You are now going to run the robot from a program instead of manually sending commands to it. There are two ways to run the Nomad robot from a program, as illustrated Figure 2.7:

- Direct mode: your program communicates directly with the robot daemon (the program constantly running on the robot that accepts commands from outside, executes them, and sends data back).
- Client mode: Your program communicates as a client to the server: the server accepts the commands of your program (exactly as it accepts your

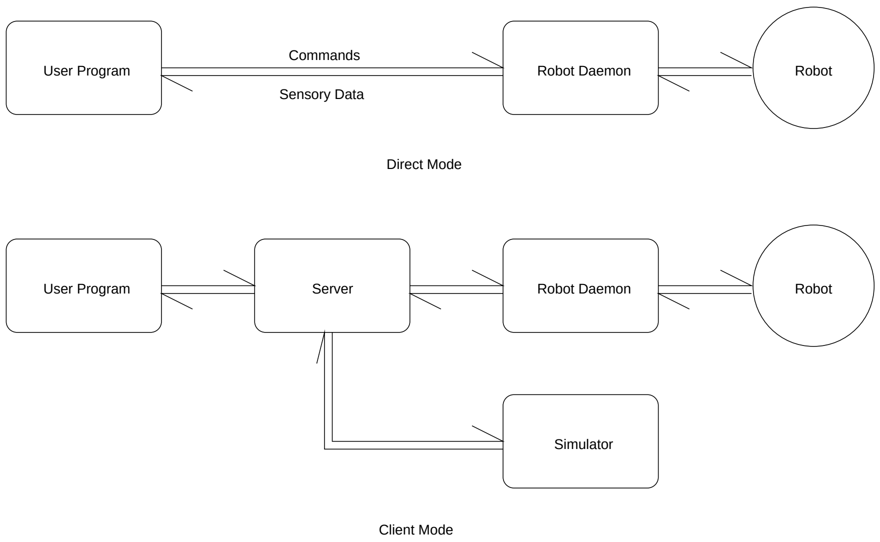

Figure 2.7: Programming in Direct and Client mode

commands from the graphic interface) and transmits them to the robot daemon. The server can also transmit your commands to a simulation module instead of the real robot.

The client mode is the preferred way of testing and debugging a program: you first run your program in client mode using the simulator. When your program works correctly, you run it still in client mode, but using the real robot. You do not have to modify anything in your program: a simple switch in the server redirects your commands from the simulator to the real robot.

The direct mode is used when your program is completely correct, to minimize the communication overhead.

We are going to test the following simple C program:

```
#include N̈client.h¨

void main()
{
connect_robot(1);
zr();
sp(50,0,0);
pr(240,0,0);
while(State[STATE_CONF_X]<1000)
gs();
disconnect_robot(1);
}
```

This program:

- connects to the robot
- initializes it using the command `zr` that you already know
- sets the translational speed to 5 inches/s, the other speeds to zero
- translates the robot by 100 inches (1000 tens of inches)
- gets the robot state during the motion
- disconnects from the robot

The include file `Nclient.h` contains the prototypes of the robot commands: `zr`, `pr`, etc. Let us first compile this program:

- Create a file `myprog.c` containing the above program, and put it in the same directory as `Nclient.h`.
- Compile it: for instance,
  `gcc -o myprog myprog.c Nclient.o`
- Run a server as explained above, but DO NOT connect to the real robot. If a server is already running, disconnect from the real robot by toggling REAL ROBOT in the ROBOT item of the robot window.
- From another xterm, run `myprog`

The program `myprog` will connect to the server, initialize the simulated robot (not that in simulation the zeroing is instantaneous) and move it. You can check the motion progress by looking at the "Encoder Position" in the information bar of the robot window. You can also see which command is in progress in "Previous Command".

♣ *Note that the robot information on the server is updated at each* `gs` *command. If there were no such call in your program, you wouldn't see any motion, because the interface wouldn't be updated. A simple way to update the graphic display is to repeatedly send* `gs()` *from the command line panel.*

♣ *When using* `pr`*, you must be aware that if you request a long motion (for instance* `pr(10000, 0, 0)`*) without sending further commands to the robot, the motion will stop before the motion is complete because of the time-out mechanism. Once again, sending* `gs()` *is a good way to keep in touch with the robot.*

!!!

- The Server displays the following on the xterm:
  `ERROR: bind call failed in server, tcp port #7019 already in use`
  You (or someone else) have a process using the same port. Most likely it is a `Nserver` process; you can kill it, or you can change the communication port by setting the `TCP_SERV_PORT` variable to another value in your program, and accordingly the `serv_port` item in the `[connect]` paragraph of the `world.setup` file.

- Your program says:
  `Not connected to any robot`
  Your program couldn't connect to the server: either you forgot the `connect_robot` command, either you do not have a Server running, either the communication ports of your server and your program do not match.

If your program works correctly in simulation, you can now use it with the real robot.

- MAKE SURE THAT THERE IS NOTHING IN THE WAY OF THE ROBOT, AND THAT NO EQUIPMENT (KEYBOARD, MONITOR,...) IS CONNECTED TO THE ROBOT
- Select REAL ROBOT in the ROBOT menu
- Run `myprog`

2.9 Shutdown and Recharge

After use, you need to shut down the robot and remove the batteries:

- Turn off the robot by pressing the red button on the front panel of the robot. Verify that the robot is off by observing that none of the LEDs on the panel is on.
- Lift up the bottom cylinder and remove the batteries. The batteries should always be removed if the robot is not operational to avoid a complete decharge that would destroy the battery.
- Once the cylinder is secured, disconnect the batteries from the robot by unplugging the battery clips for each battery.
- Then remove the three batteries by pulling them towards you.

2.10 Battery Maintenance Guidelines

This section describes the proper use, charging and storage for the batteries used with your Nomad 200 system. Correct maintenance will ensure proper operation of your Nomad as well as lengthen the life of your batteries.

2.10.1 Battery System Overview

The power system on the Nomad is composed of two separate systems. The first of these is a 24V system that is used to power the Nomad's motors. It is composed of two 12V-12AH batteries stacked in series. These batteries are located in the base of the Nomad and are connected to cables #17 and 19. Secondly, 12 V system is used to power the computers and sensors. For Nomad's equipped with the auxiliary battery floor the 12 volt system is composed of three batteries. The first of these three batteries is a 12V-12AH battery that is located in the base and is connected to cable #18. The remaining two batteries have a capacity of 12V-17AH. They are located in the turret of the Nomad and connect to cables # 36 and 37. These two systems are 100% isolated from each other.

2.10.2 Dividing your batteries into sets

It is important to divide your batteries into "sets". A set is composed of the five batteries (three CPU and two motor) that are used at any one time by the robot. The best way to keep the sets organized is to label them. For example if you have two sets 'A' and 'B' are good labels. Furthermore it is always good to use and charge batteries together. This prevents a discharged battery from being connected to a charged battery which can cause damage to both batteries. This means that the three batteries used for the 12V CPU system should be used together AND charged together and that the two batteries used for the 24V motor system should be used together AND charged together. To ensure this, each set of batteries should be broken into two "subsets" and labeled to prevent them from being mixed together. For example the two motor batteries can be labeled with and 'M' and the three Computer batteries can be labeled with a 'C'. In summary a set of five batteries would have two subsets 'M' and 'C' that should be used and charged together.

2.10.3 Charging

It is best to charge the batteries external to the robot. Each of the subsets mentioned in the previous paragraph should be charged individually using the three battery harness and the charger provided with the Nomad. Since there are only two motor batteries in the motor subset there will be one unused connector when the motor battery subset is charged. The fast charge light on the charger will light when the batteries are first connected to the charger. After a period of time that depends upon how deeply the batteries were discharged (usually 4-8 hours) the fast charge light will turn off and the charger will go into trickle charge mode. At this point the batteries are approximately 90hours for the batteries to reach 100much current to a battery never put one of the small batteries (12AH) on the charger by itself.

2.10.4 Usage

When the robot is loaded and running with a set of charged batteries you should get between 6-8 hours of operating time. This estimate assumes a Nomad 200 that has standard equipment installed in it. Each of the two power systems, motor and CPU has a low voltage alarm system connected to it. Whenever the voltage drops below 10.8 Volts for the CPU system or 21.6 Volts for the Motor system an alarm will sound. When the alarm is heard the batteries should be removed from the robot as soon as possible to avoid over-discharging them. Once the batteries have been deep discharged they are permanently damaged.

2.10.5 Storage

When the Nomad is not in use the batteries should be disconnected from the robot. This is due to the fact that a fuse status LED is connected and will slowly drain the battery if it is left connected. If the batteries will not be used for an extended period of time it is best to charge them every 6 months.

2.10.6 Battery Lifetime

When the batteries are maintained according to the above specifications they should yield approximately 200 charge/discharge cycles. When they begin to discharge very quickly, or will not hold a charge at all it is time to buy a new set.

In summary, observing the following guidelines will allow you to make the best of your batteries:

- Proper maintenance and use of your batteries will ensure the proper operation of your Nomad, and lengthen the life of your batteries.
- There are two independent power systems on the Nomad 200. One is for the motor system and the other is for the CPU/sensor system. These systems are isolated from each other.
- To ensure that the batteries are exercised and recharged together it is best to divide them into sets and to not mix up the sets. Within each set of batteries there are two subsets, one for each of the power systems
- The subsets should be charged together and will require between 8-12 hours to be fully charged
- Low voltage alarms will sound off if the battery voltage goes too low on either system. The batteries should be removed from operation as soon as possible (5 minutes) after the alarm sounds.

- When not in use the batteries should be disconnected from the robot.
- If not used for an extended period of time the batteries should be charged every 6 months.
- When used properly the batteries should yield approximately 200 charge/discharge cycles.

2.11 Where to go next?

You have completed this introduction to the basic operation of the Nomad. You may now:

- get a more detailed description of the use of the graphic interface and the simulator: go to Chapter 3
- get an introduction to programming concepts and command language features: go to Chapter 4
- get some information on advanced topics like the use of position data, sensors characteristics, setup files, etc: go to Chapter 5.

Chapter 3

The Graphic Interface

3.1 Introduction

The Graphic User Interface (GUI) provides a convenient access to the real and simulated robots, and to the representation of the world, as illustrated Figure 3.1. Through the GUI, the user can send command to robots, monitor command execution by seeing the robot actually moving on the screen, visualize instantaneous and cumulated sensor data. The user can also create and modify a simulated environment, and use it to test robot programs.

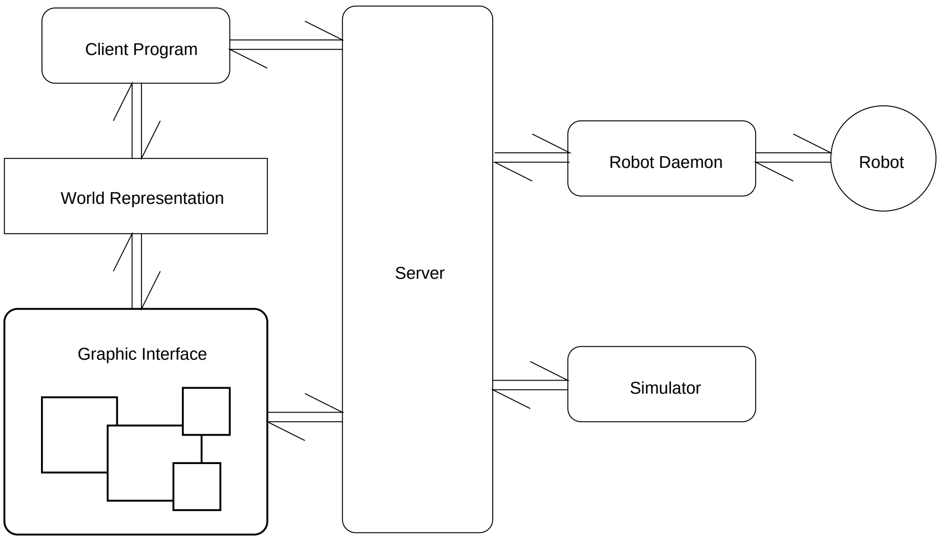

Figure 3.1: The graphic interface in the programming environment

3.1.1 The graphic display

To run the GUI, open an xterm, `cd` to the directory where the `Nserver` is (it should also contain files `world.setup`, `robot.setup` and `license.data`) and type:

> $>$ `Nserver`

You will see (Figure 3.2):

- The Map window
- One robot window
- The Short Sensors and LongSensors windows

The xterm is then used for displaying text messages from the server.

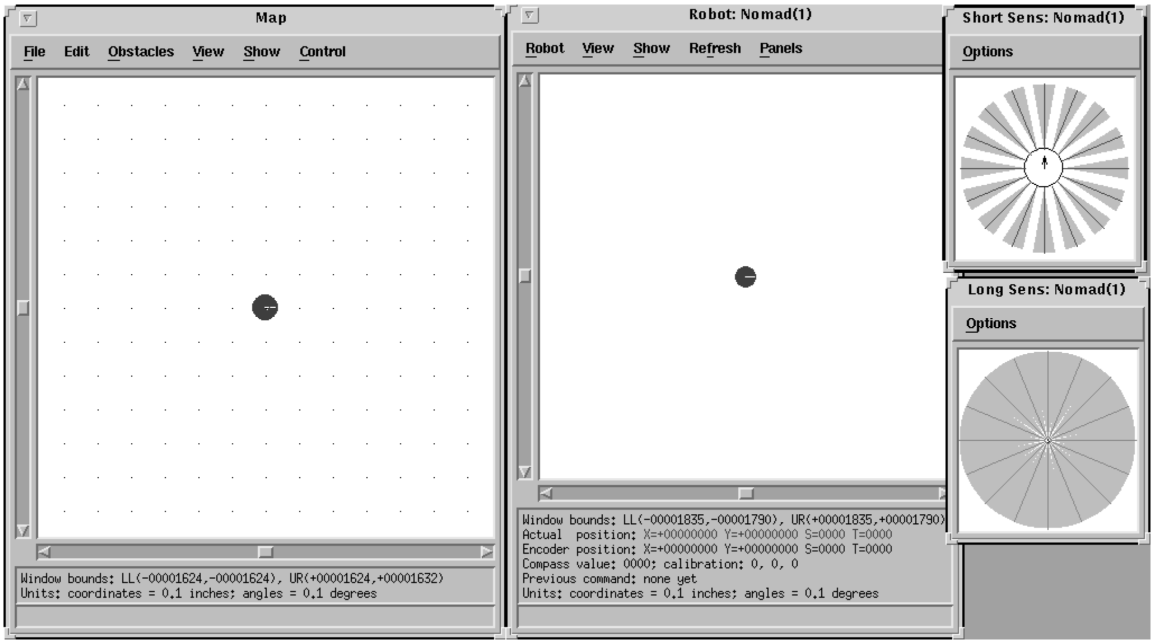

Figure 3.2: The Four Main Windows

The GUI presents 2 different views of the world the robot(s) are in:

- A global view ("God's eye"): the Map window, in which all the robots are represented
- A local view: the Robot window, that displays the world as seen by one robot only

Thus, there is only one Map window, but as many Robot windows as there are robots (real or simulated). The Map window gives access to the world representation, with functions such as: creation of obstacles, edition of obstacles, etc. The Robot window allows the display of sensor history, the path of the robot, as well as the execution of robot commands. Both windows support usual display functionalities like zooming and unzooming, scrolling, centering, etc. Several display parameters are controlled by values set up in the file `world.setup`. See the section 5.3 for an explanation of the display parameters.

3.1.2 The simulator

There is no difference in the graphic display whether the interface is dealing with a real or a simulated robot. By default, the GUI is in simulation mode. The REAL ROBOT option of the ROBOT item in the Robot window is used to toggle between the real and the simulated robot.

The Simulated Environment The simulated robot moves in a two-dimensional world populated with other robots, and convex polygonal obstacles. These obstacles can be added to the current environment from the Map window, or by program using the world manipulation functions. The sensor data displayed on the robot window under simulation comes from modules simulating each of the sensors: these modules compute the sensor's answer according to the current (simulated) robot environment.

Actual/Encoder Robot In the physical world, the robot knows its position by integrating the encoders' information. This is not exact information however, since there is some slippage of the wheels on the floor. The position of the physical robot is different from the position given by the encoders, and the difference keeps growing. In simulation, the actual robot represents this phenomenon. The position of the actual robot is equal to the encoder position, plus some error. The actual robot and the encoder robot are identical just after zeroing (the encoders are reset to zero, as well as the integrated position), and diverge according to the error model afterwards. For example, if a `gs()` command is issued in simulation, the position information returned in the `State` vector will come from the encoder robot, but the sensing information will be computed (simulated) from the actual robot's position.

On the GUI, you can display Actual and Encoder robots in different colors and check for their behavior (when you are in real robot mode, no actual robot is displayed).

The parameters describing the behavior of the simulation modules are in the `robot.setup` file. See the section 5.3 for an explanation of the simulator parameters.

3.1.3 Running the Nserver with arguments

When you run `Nserver`, the program looks for arguments on the command line: the first argument is expected to be a configuration file for the environment, containing among others display parameters for the Map window. All the other arguments are expected to be configuration files for robots. When you run `Nserver` without arguments, the programs looks for a default Map initialization file in the current directory: `world.setup`. Thus, calling

```
>Nserver
```

is equivalent to calling

```
>Nserver world.setup robot.setup
```

if `robot.setup` is the declared robot configuration file in `world.setup` (see below). If you want to automatically create several robots windows corresponding to several real or simulated robot (instead of creating them by hand from the graphic interface), you do so by providing several setup files at the command line level:

```
>Nserver world.setup robot.setup_A robot.setup_A robot.setup_B
```

In the example above, three robots will be created, two using the setup file `robot.setup_A`, and one using the setup file `robot.setup_B`.

If you *always* want to create several robots when you run a server, then you can specify their setup files in the environment configuration file (`world.setup` by default), section `robots`, item `setup_files`. For instance, if you have the following in your `world.setup` file:

```
[robots]
setup_files = robot.setup_A robot_setup_A robot.setup_B
```

then three robots will be automatically created each time you call `Nserver` (without arguments), two using the setup file `robot.setup_A`, and one using the setup file `robot.setup_B`.

♣ *Command line arguments override the specifications given in the environment setup file.*

3.2 The Map Window

The Map window allows an application to interactively define and modify the *map* of the world where a robot moves around. The robot world is an ab-

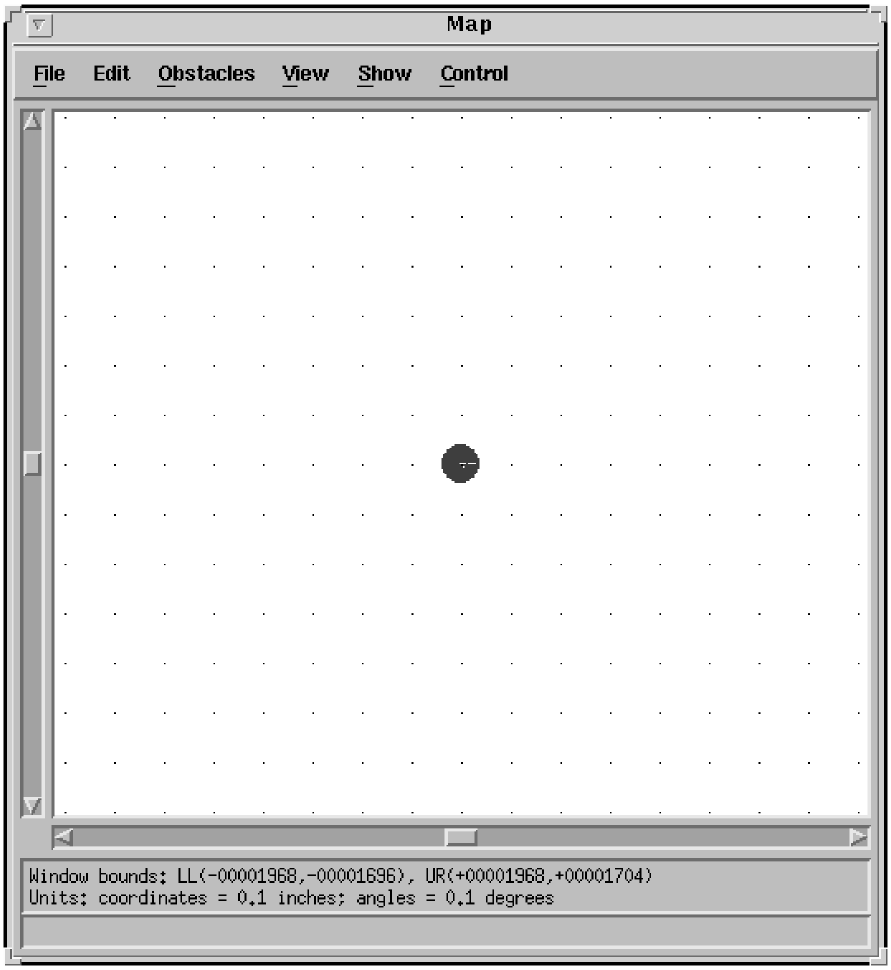

Figure 3.3: The Map Window

stract coordinate system; Its dimensions are set in the `world.setup` file under `[physical]`, item `size`. The two pairs of coordinates at the bottom of the window reflect the current positions of the window's lower-left corner and the upper-right corner in the world, respectively. Look for units information also at the bottom of the window.

Initially, the (0,0) point of the coordinate system is the center of the Map window.

This window has six menus: **File**, **Edit**, **Obstacles**, **View**, **Show**, **Control**.

3.2.1 The File Menu

The robot map is treated as a disk file. The environment is describe one obstacle per line, each line beginning with the number of vertices, followed by the x-y coordinates of the vertices enumerated counterclockwise.

Choose the menu to select one of the following five operations:

New Map will create a blank map with no obstacles. All the robots will remain unchanged.

Open Map will prompt you to open a map file, which was previously saved by a **Save Map** or a **Save Map As** operation. After selecting the map file, click OK. The map name, along with its path, will appear as the name of the window in the top bar.

Save Map will save the current map to the same file that is currently open. If this is the first time to save a file, it will prompt you to select the path and give a file name.

Save Map As allows you to save the current map in a different file. It will prompt you to select the path and give a file name.

Quit will quit the whole current server application. All windows will be closed.

3.2.2 The Edit Menu

The **Edit** menu allows you to edit the obstacles in the map. Before performing the following operations, you have to select an obstacle by clicking on the obstacle, in **Grab Obstacle** mode from the **Obstacles** menu.

Copy will store the selected obstacle in the internal clipboard.

Cut will delete the selected obstacle and store it in the internal clipboard.

Paste will paste on the map the obstacle currently in the internal clipboard. It requires a previous Copy or Cut operation. The pasted obstacle will overlap the one it was copied from, and can be moved.

Clear will delete the selected obstacle, without storing it in the internal clipboard.

3.2.3 The Obstacles Menu

The **Obstacles** menu allows you to add, delete, and select obstacles in the map. It has the following modes:

Add Rectangles is to create rectangular obstacles. To add a rectangular obstacle, click on a position where you want one corner of the rectangle to be, hold and drag the mouse, and release on a second position where you want the diagonal corner of the rectangle to be. You can repeatedly add rectangles when in this mode.

Add Polygons is to create polygonal obstacles. To add a polygonal obstacle, click on a position to be the starting vertex of the polygon, move and click on a position to be the second corner of the polygon, and continue *counterclockwise* until finally click on a position close enough to the starting vertex. You can repeatedly add polygons when in this mode.

♣ *Currently, only convex polygons are allowed. However, overlapping obstacles are allowed, so that you can create non-convex shapes by overlapping convex polygons.*

Grab Obstacle allows you to select an obstacle by clicking on it. A selected obstacle has handles at its vertices. It can be moved by click and drag, and reshaped by moving the handles.

Delete Obstacles will delete obstacles by clicking on them. You can repeatedly delete obstacles in this mode.

♣ *New Map option in the File menu will delete all the obstacles at once.*

3.2.4 The View Menu

The **View** menu performs the following operations:

Graphics On displays the robots' motions in the Map and Robot windows during simulation. If this option is off, the graphic overhead will be avoided. The default for this option can be set on or off in the `world.setup` file, under `[graphics]`, item `graphics`.

Zoom In enlarges the map and centers it on the click-on point. You will see a map twice as large.

Zoom Out reduces the map and centers it on the click-on point. You will see a map half as large.

Center designates a point in the map as the center of the Map window. After selecting Center in the View menu, click on a point on the map and the point will be the center of the Map window.

Clip allows you to selectively zoom in a region of the Map window. After selecting this option, click and drag to form a rectangle to select the desired region to zoom in.

Slide allows you to move the whole map in the window. After selecting this option, position the mouse in the map and move the map by dragging the mouse.

3.2.5 The Show Menu

The **Show** menu performs the following operations:

Map is to switch on and off the display of the obstacles.

Actual Robot is to switch on and off the display of the robot at its actual position (which only has meaning in simulation mode) in the Map window.

Encoder Robot is to switch on and off the display of the robot at its encoder position in the Map window.

3.2.6 The Control Menu

The **Control** menu performs the following operations:

Create Robot creates another robot in the current simulation. It will prompt you for a configuration file.

The robot id of the created robot will be given sequentially. A new Robot window will show up with its attached Sensors windows. The maximum number of robots currently allowed is 6.

♣ *This is a "manual" creation of a robot. Automatic creation can be achieved using the* `setup_files` *option in the environment setup file*

♣ *Removing a robot is done through the DESTROY ROBOT item of the ROBOT menu in the corresponding robot window*

Speedup Simulation speeds up the simulation process by making the simulation process twice as fast.

Slowdown Simulation slows down the simulation process by making the simulation process half as fast.

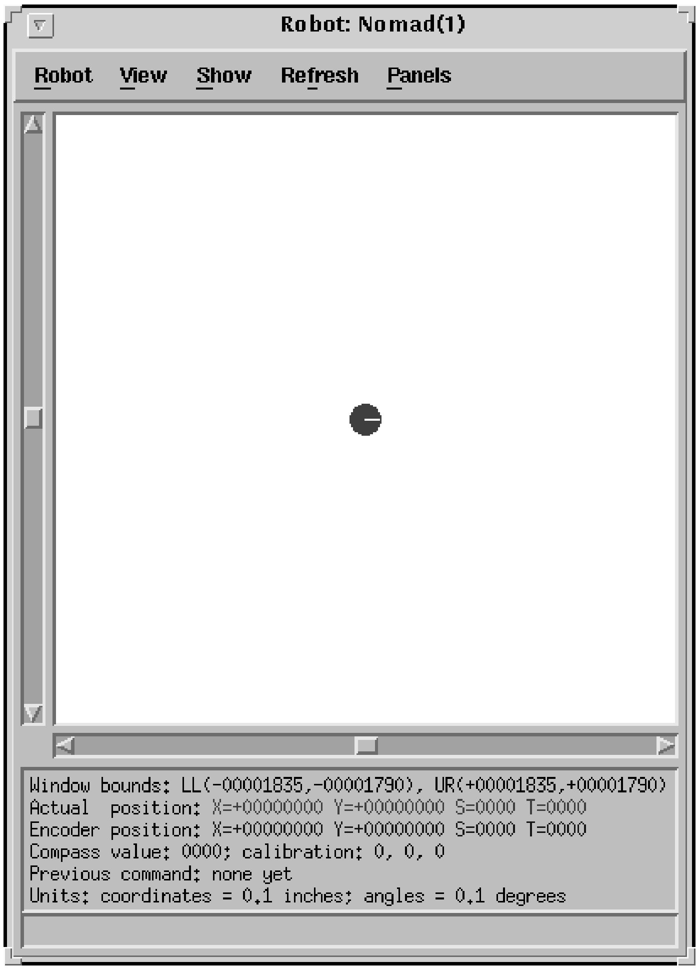

Figure 3.4: The Robot Window

3.3 The Robot Window

The Robot window allows interactive control of a robot. This window has five menus: **Robot**, **View**, **Show**, **Refresh**, **Panels**.

At the bottom of the window are information about the current robot position, compass value, and the last command issued. In position information, X and Y are the coordinates, S is the steering direction in degrees, T is the turret direction in degrees. Degrees range from 0 to 360, with 0 as the horizontal right.

3.3.1 The Robot Menu

The **Robot** menu has the following options:

Real Robot switches between the real robot and the simulator. When you are switching to the real robot, you must make sure that the appropriate communication (radio modem or radio ethernet) is set up between the robot and the host computer.

WARNING AFTER SWITCHING TO THE REAL ROBOT, EVERY ACTION (COMMANDS, JOYSTICK MOTIONS) PERFORMED IN THE GUI WILL BE DIRECTLY EXECUTED BY THE ROBOT: MAKE SURE THAT THE ROBOT'S ENVIRONMENT IS SAFE.

Place Robot allows you to place the robot in a given configuration. Synchronization bars and handles appear as shown in Figure 3.5. Only the robot that are currently displayed in the window according to the SHOW menu will be affected by place robot operation. If only the Encoder robot is shown, then the Encoder position and orientation will be reset. Let's assume that this is the case:

- Clicking Left with the left button on a (free) place will set the new position of the robot to this place
- Click-and-Dragging Left the farthest handle will set the turret angle. Dragging the closest handle will set the steering angle. You can monitor the value of the angle in the Position display at the bottom of the window
- Clicking Left on one of the synchronization bars will align both turret and steering to the angle of that bar
- Click-and-Dragging Right on one of the synchronization bars will move both the steering and the turret, keeping their relative angle

To finish, click in the gray label: bars and handles disappear.

♣ *When connected to a real robot, the only information modified is the encoder information: placing the robot will only have an effect on where*

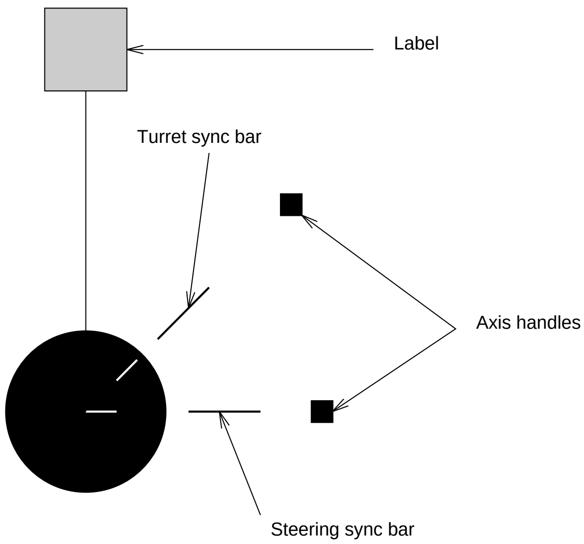

Figure 3.5: Place robot

the robot thinks it is, but won't incur any motion. Only in simulation is the actual robot modified.

There are some specific operations when the two robots, Encoder and Actual, are displayed at the same time; you can monitor the effects of your actions by having both encoder and actual robot displayed in the Map window. Note however that clicking left will always set the position of both Encoder and Actual robots, even if only one is displayed.

- When Encoder and Actual robot are identical, the handle and synchronization bars are grayed. Gray handles and bars move both robots at the same time. To separate them, hide one of the robots in the SHOW menu.
- Clicking Left on some position will reset Encoder robot AND Actual robot x-y position. Dragging Left will move only one robot at a time.

Destroy Robot destroys the robot. All the three windows Robot, ShortSensors, and LongSensors associated with the robot will disappear. You can create a new robot from the Map window Control Menu.

3.3.2 The View Menu

The **View** menu has the following options:

Zoom In enlarges the map and centers it on the click-on point. You will see a map twice as large.

Zoom Out reduces the map and centers it on the click-on point. You will see a map half as large.

Center designates a point in the robot world as the center of the Robot window. After selecting this option, click on a point on the map and the point will be the center of the Robot window.

Clip allows you to selectively zoom in a region of the Robot window. After selecting this option, click and drag to form a rectangle to select the desired region to zoom in.

Slide allows you to move the whole robot world in the Robot window. After selecting this option, position the mouse in the Robot window and move the robot world by dragging the mouse.

3.3.3 The Show Menu

The **Show** menu has the following options:

Robot Trace sets various ways to display a trace of the robot as it moves. Currently available are **Solid** and **Outline**. Select **None** to remove the trace. Trace accumulated so far can be removed using the REFRESH options.

Map switches on and off the display of the obstacles of the map.

♣ *Superimposing the obstacles to the traces of the sensors gives an idea of how well sensors capture the environment*

Actual Robot switches on and off the display of the robot at its actual position (which only has meaning in simulation mode) in the Robot window.

Encoder Robot switches on and off the display of the robot at its encoder position in the Robot window.

Bumper switch on and off the display of the bumper when the robot runs into an obstacle.

Infrared switches on and off the display in the Robot window of what the infrared sensors see.

Sonar switches on and off the display in the Robot window of what the sonar sensors see.

Laser switches on and off the display in the Robot window of what the laser sensor sees.

♣ *Displaying pixel mode laser data makes no sense! See* `conf_ls`

3.3.4 The Refresh Menu

The **Refresh** menu allows you to get rid of the graphic robot traces and sensor history accumulated so far. The display of traces and sensor history will continue on a blank background (use the SHOW menu options to stop these). The **Refresh** menu has the following options:

All clears everything mentioned in the rest of the menu — traces and histories.

All Traces clears traces of the actual robot and the encoder robot. See the **Show** menu for setting robot traces.

Actual Robot Trace clears the trace of the actual robot.

Encoder Robot Trace clears the trace of the encoder robot.

All Sensor History clears all the histories (successive sensor marks) mentioned in the remaining of the menu. See the **Show** menu for setting sensor display.

Bumper History clears the bumper-related marks in the Robot window.

Infrared History clears the infrared-related marks in the Robot window.

Sonar History clears the sonar-related marks in the Robot window.

Laser History clears the laser-related marks in the Robot window.

Client Graphics clears the drawings made by a client program, using instructions such as `draw_line` or `draw_robot`.

3.3.5 The Panels Menu

The **Panels** menu has the following three robot monitoring/control methods:

Recorder displays a Record&Playback center for recording robot motion and sensing sequences. This option also allows you to play back recorded sequences. To use the recorder:

- Select the items to record by Left-clicking (the Recorder uses only the left mouse button)
- Click on the record button (red dot) to start recording. A record is made and time-stamped each time the `State` vector gets updated (almost all robot commands update the State vector: See section 4)
- Click on the stop button (black square) to stop recording
- Click on the left double-arrow to rewind (the single arrow rewinds one recording at a time)
- Click on the play button (green triangle) to play the recording. You can move forward and fast-forward, backward and fast-backward in the recordings.

Records can be saved to and loaded from a text file using LOAD and SAVE options. To dismiss the Recorder panel, select **Recorder** again.

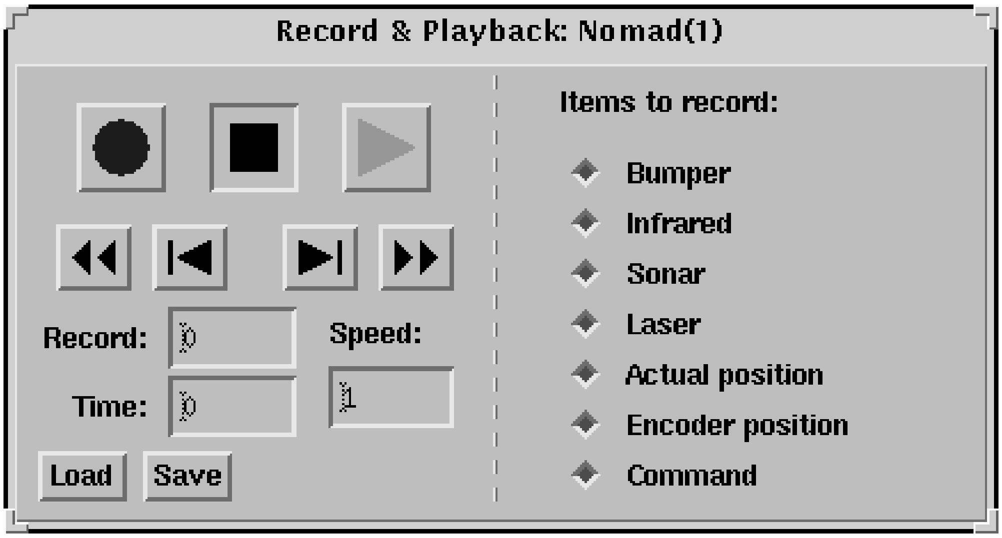

Figure 3.6: The Recorder Panel

Command Line allows you to type in a robot command from a console. All of the robot commands mentioned in the Language Reference Manual manual are supported.

♣ *However, you cannot set the individual values of the* `SMask` *vector from this panel (there is no robot command to do this as well); you can only reset it with* `init_sensors` *to one everywhere (all available sensor data gets returned). The only way of setting individually the* `SMask` *vector values is through a client program.*

The top-half panel contains pre-defined calls to standard robot instructions like `zr` or `pr`. When you click on one of these, the bottom-half panel drops and displays a number of text fields corresponding to the command (zero for `zr` that takes no argument, sixteen for `conf_ir`) (see Figure 3.8).

Other commands can be entered by typing their names in the Command line. Typing a space after the command name will drop the corresponding number of editing fields. Type a space after each value entered to jump to the next editing field.

The command will be actually sent to the robot after clicking on EXECUTE. Click on CLEAR to cancel, and select COMMAND LINE again

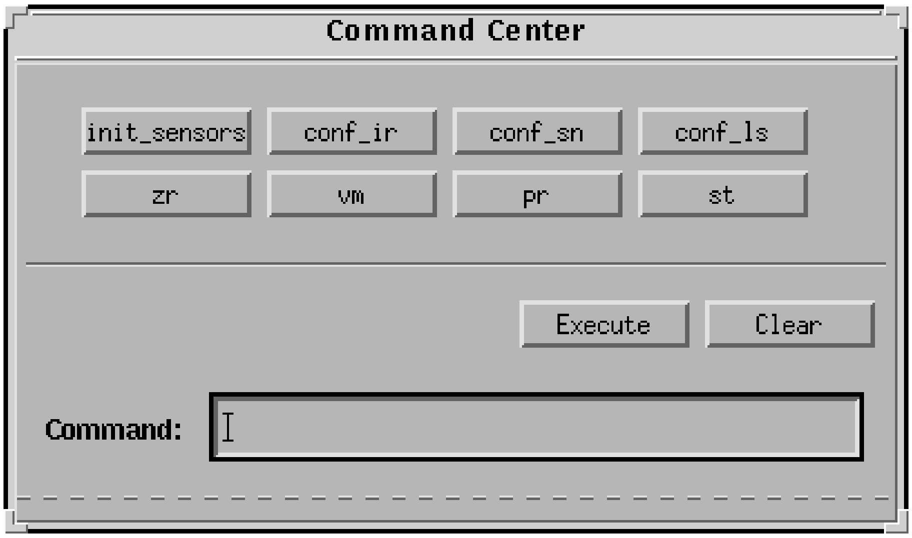

Figure 3.7: The Command Line Panel

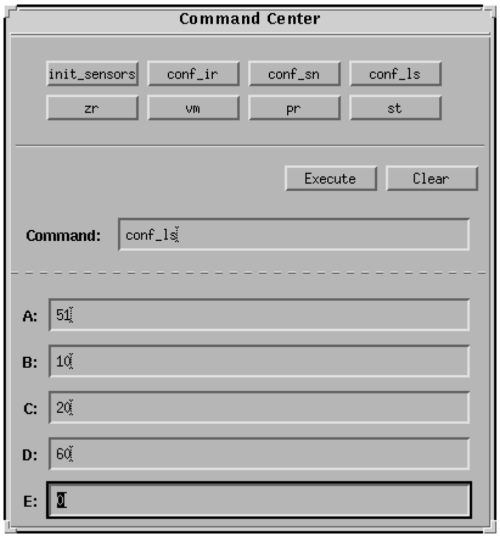

Figure 3.8: Entering parameters in the command center

to dismiss the command panel.

The COMMAND LINE does not support the user-packet commands, the position attachment commands, and the `get_rpx` command.

Joystick allows you to move the robot or the simulated robot around by moving the mouse in the Joystick window (Figure 3.9) which will appear once this option is selected.

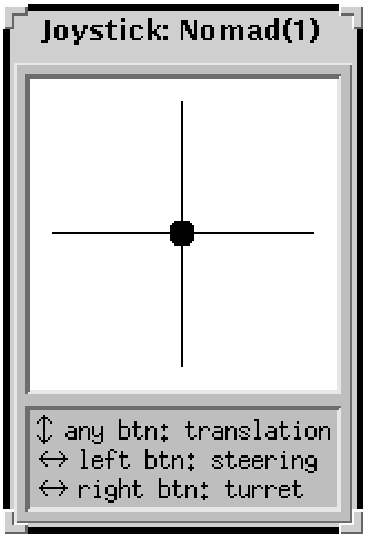

Figure 3.9: The soft joystick

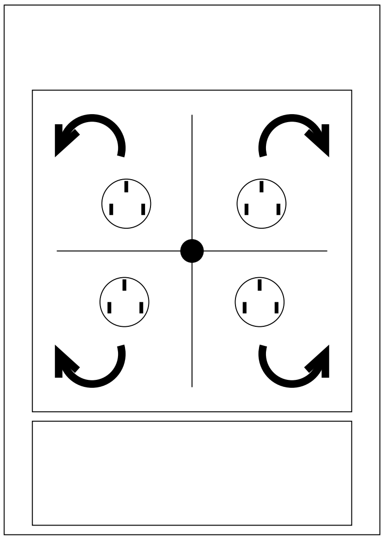

Figure 3.10: Combined motions

- The crosshairs represent the two degrees of freedom of the physical joystick. The central dot is the joystick: the robot moves according to its position.
- Left clicking and dragging the dot to the top or the bottom will move the robot forward orbackward by sending repeated `vm` commands. The speed value depends on how far from the center you move the joystick dot (the farther, the faster).
- Release the button to stop the motion.
- Left clicking and dragging the joystick dot to the left or the right will steer the robot to the left or the right. The farther you go, the faster the steering. Releasing the button will stop the motion.
- As with the physical joystick, combined motions are possible, as illustrated in Figure 3.10.
- With the right button, you can translate the robot along the vertical axis like with the left button. However, the horizontal axis will rotate the turret.

♣ *A common mistake is to give an absolute meaning to the soft joystick directions, like "North", "East", etc. The motions are always relative to the current orientation of the wheels. Moving the dot to the top will move forward, which may be "South" if the robot is so oriented.*

♣ *If your mouse has three buttons, the middle button is equivalent to pushing both the left and right together, i.e. the vertical axis will translate, the horizontal axis will steer AND rotate the turret.*

To dismiss the Joystick panel, select **Joystick** again.

3.4 The ShortSensors Window

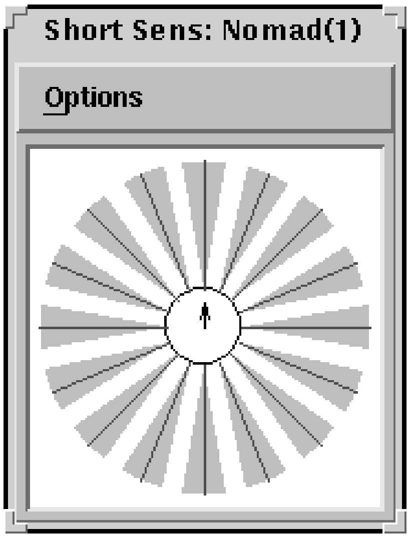

Figure 3.11: The ShortSensors Window

The **ShortSensors** Window has only one menu: **Options**.

3.4.1 The Options Menu

The **Options** menu has the following options:

Show Bumper allows you to switch on and off the display of the bumpers when the robot runs into an obstacle.

Show Infrared Rays allows you to switch on and off the display of the infrared sensor data as radius lines.

Show Infrared Cones allows you to switch on and off the display of the infrared sensor data as cones, which is closer to the physical reality of infrared sensors but longer to draw.

IR Cones Are Arcs allows you to switch on and off the display of the infrared sensor data (IR) as cones with arcs at the outside ends. Arcs are a better representation, but take longer to draw.

IR Cones Are Filled allows you to switch on and off the display of the infrared sensor data as cones shaded or outlined. Shaded cones are a better representation, but take longer to draw.

Show Infrared Connections allows you to switch on and off the display of the infrared distance points, connected by line segments.

Global View allows you to switch between global view and local view of the robot. In local view, the robot's forward direction is always aligned with the upward vertical direction of the window. In global view, the robot will rotate with respect to the existing environment, as the turret of the robot rotates.

3.5 The LongSensors Window

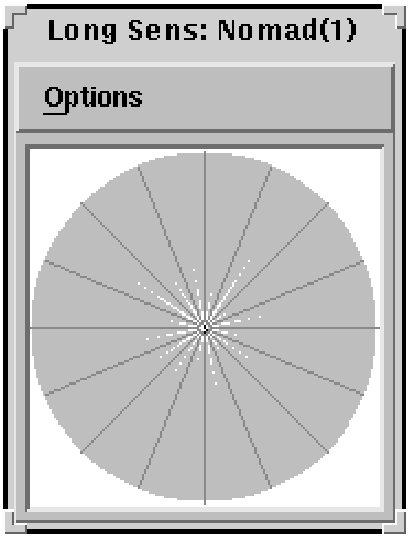

Figure 3.12: The LongSensors Window

The **LongSensors** Window has only one menu: **Options**.

3.5.1 The Options Menu

The **Options** menu has the following options:

Show Sonar Rays allows you to switch on and off the display of the sonar sensor data as radius lines.

Show Sonar Cones allows you to switch on and off the display of the sonar sensor data as cones, which is closer to the physical reality of sonar sensors but longer to draw.

Sonar Cones Are Arcs allows you to switch on and off the display of the sonar sensor data as cones with arcs at the outside ends. Arcs are a better representation, but take longer to draw.

Sonar Cones Are Filled allows you to switch on and off the display of the sonar sensor data as cones shaded or outlined. Shaded cones are a better representation, but take longer to draw.

Show Sonar Connections allows you to switch on and off the display of the sonar distance points connected using line segments.

Show Laser allows you to switch on and off the display of the laser data as lines or dots depending how the laser is configured. Use `conf_ls` in the **Command Line** panel to configure the laser; `conf_ls 51 10 20 60 0` for example. If the laser has not been configured by `conf_ls`, nothing will show up in the LongSensors window even if **Show Laser** is switched on.

Show Robot Proximity allows you to switch on and off the display of dots when another robot is nearby. This display will occur only if a client program sends a `get_rpx` command.

Global View allows you to switch between global view and local view of the robot. In local view, the robot's forward direction is always aligned with the upward vertical direction of the window; all the sensors are fixed. In global view, the robot will rotate with respect to the environment shown in the Map window, as the turret of the robot rotates.

Chapter 4

Programming the Nomad

In this chapter the concepts and techniques required to write application programs for the Nomad robot are presented. It is recommended to read this chapter completely before starting to develop software.

To exploit all features of the robot and the programming environment the programmer has to be familiar with the basic architecture of the system and the entailed programming concepts, which are presented in the following section. The subsequent section gives an overview of the different classes of commands that can be used in application programms. How to use these commands is demonstrated in the last section of this chapter with example programs.

4.1 Programming Concepts

4.1.1 Programming Modes

An application program can communicate with the Nomad in two different ways (see Figure 4.1):

- Client mode: the application communicates with a server, which in turn communicates with the robot daemon *or* the simulator.
- Direct mode: the application communicates directly with the robot daemon.

Which of these two modes should I use? While developing an application program it is usually advisable to use the client mode. When using the client mode it is possible to switch between real robot and simulated robot. This is convenient during debugging because the robot cannot be damaged due to a programming error.

Once a program is running stably the direct mode can be used: the program can run independently of the server.

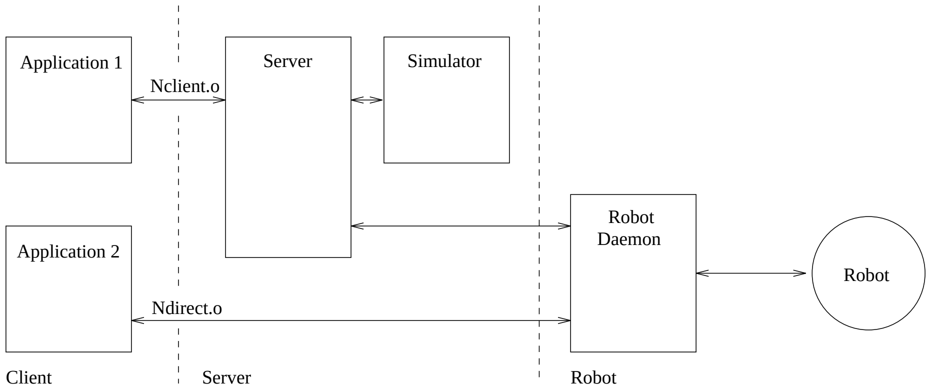

Figure 4.1: Programming in Direct and Client mode

How can I switch between these modes? To switch between the two modes the program simply has to be relinked. The object file `Nclient.o` is used for the client mode and the object file `Ndirect.o` is used for the direct mode. In general the source code does not need to be changed.

Are the modes compatible? The two modes are fully compatible. However, commands that are specific to the server/simulator will have no effect in direct mode. Nevertheless, the procedure call will succeed and return an appropriate value. When writing applications that are supposed to run in client mode as well as in direct mode the programmer has to take these differences into account; programs must not rely on functionality provided by the server. In the file `Nclient.h` functions that have no effect in direct mode are marked. Also, section 4.3.4 discusses these commands in detail.

How can I specify to which robot/server I want to connect? How to establish a connection with a specific robot or server is explained below in section 4.3.1 on communication commands.

4.1.2 Simulated versus Real Robot

From the programming point of view, there is no difference between the simulated robot and the real robot: you do not have to change your program to switch between them. However, since simulation is always an idealization of data, especially regarding sensing, you should be ready to experience some variations between simulated and real data.

When running in client mode, you can switch between real and simulated robot using the REAL ROBOT item of the ROBOT menu in the ROBOT window. You can also switch from a program using the commands `real_robot` and `simulated_robot`. Again, these commands will have no effect in direct mode. Direct mode always uses the real robot.

If the application relies on realistic sensory data the direct mode should be used, since the simulation cannot capture all physical aspects of the real world.

4.1.3 C and Lisp

To control the robot from within a C program, you include the `Nclient.h` header file and link your program with the `Nclient.o` or `Ndirect.o` library. You can also use the robot from C++ by declaring all the commands appearing in `Nclient.h` as `external C`.

To control the robot from within a Lisp program (Lucid Common Lisp), you must first load `Nclient.lsp` into your Lisp environment. `Nclient.lsp` in turn loads `Nclient.o`.

4.2 The Global Vectors

Information about the current state of the robot, its configuration and the readings of the sensors can be obtained by an application program through a global array, called the `State` vector. This structure is updated after the execution of a robot command (see section 4.3 for a detailed description of the commands). Which of the fields in that array are updated is determined by the global vector `Smask`, which stands for state mask. Another global array, the `Laser` vector, contains the sensory reading of the laser. All three of these structures are described in detail below. Table 4.1 provides a quick reference to `State` and `Smask`.

The name fields are values that are defined in `Nclient.h`; they should be used in application programs rather than the index into the array. This increases readability and is invariant to changes in the state vector.

4.2.1 The State Vector

In this section the fields of the state vector are explained in more detail. See also Chapter 5.1.

`STATE_SIM_SPEED` Simulation speed. The value is a factor to the speed of realtime; its unit is $\frac{1}{10}$. Therefore, 10 corresponds to realtime, with a setting of 5 the simulation of one second will take two seconds (half the speed) and with a setting of 20 it will take $\frac{1}{2}$ second (twice the speed).

`STATE_IR_0 - 15` The readings of the sixteen infrared sensors. The infrared sensors are numbered counter-clockwise consecutively beginning with the front of the robot (opposite the emergency stop button). The readings correspond to distances in inches according to the con-

	Name	State Vector	Name	Smask Vector
0	STATE_SIM_SPPED	speed of simulation	SMASK_POS_DATA	position attachment
1	STATE_IR_0	infrared data #0	SMASK_IR_0	infrared mask #0
2	STATE_IR_1	infrared data #1	SMASK_IR_1	infrared mask #1
3	STATE_IR_2	infrared data #2	SMASK_IR_2	infrared mask #2
...	...	...	...	...
16	STATE_IR_15	infrared data #15	SMASK_IR_15	infrared mask #15
17	STATE_SONAR_0	sonar data #0	SMASK_SONAR_0	sonar mask #0
18	STATE_SONAR_1	sonar data #1	SMASK_SONAR_1	sonar mask #1
19	STATE_SONAR_2	sonar data #2	SMASK_SONAR_2	sonar mask #2
...	...	...	...	...
32	STATE_SONAR_15	sonar data #15	SMASK_SONAR_15	sonar mask #15
33	STATE_BUMPER	bumper data	SMASK_BUMPER	bumper mask
34	STATE_CONF_X	x position	SMASK_CONF_X	x mask
35	STATE_CONF_Y	y position	SMASK_CONF_Y	y mask
36	STATE_CONF_STEER	steering angle	SMASK_CONF_STEER	steering mask
37	STATE_CONF_TURRET	turret angle	SMASK_CONF_TURRET	turret mask
38	STATE_VEL_TRANS	translational velocity	SMASK_VEL_TRANS	transl. vel. mask
39	STATE_VEL_STEER	steering velocity	SMASK_VEL_STEER	steering vel. mask
40	STATE_VEL_TURRET	turret velocity	SMASK_VEL_TURRET	turret vel. mask
41	STATE_MOTOR_STATUS	motor status	—	—
42	STATE_LASER	laser mode	SMASK_LASER	laser mask
43	STATE_COMPASS	compass data	SMASK_COMPASS	compass mask
44	STATE_ERROR	error number	—	—

Table 4.1: A quick reference to the State and Smask arrays.

version table in `infcal.cfg` on the robot that relates measured infrared intensities and distances.

`STATE_SONAR_0 - 15` The readings of the sixteen sonars. The sonars are numbered like the infrared sensors. The readings correspond to distances in inches (see also section 5).

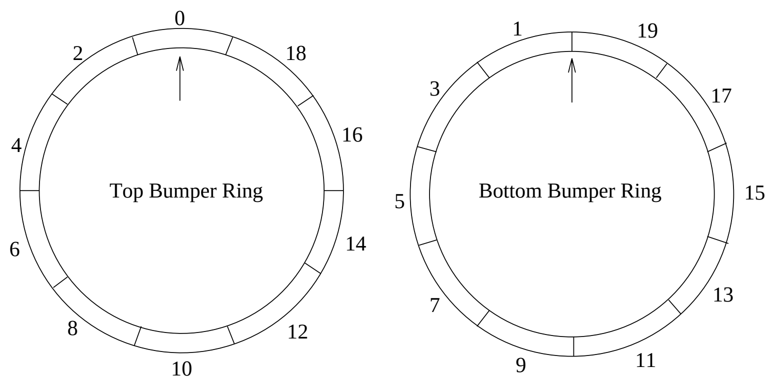

Figure 4.2: The arrangement of the bumper sensors.

`STATE_BUMPER` The readings of the bumpers. There is a total of twenty individial bumper sensors on the robot that are arranged in two rings of ten sensors each, one upper ring and one lower ring. Refer to Figure 4.2 for an illustration of their arrangement. Bumper sensor number n is represented by the $(n+1)$th bit in this value of the state vector. The 0th bit is the least significant one. A bit is set to one when the corresponding bumper is hit.

`STATE_CONF_X` The integrated x-coordinate of the robot in $\frac{1}{10}$s of inches with respect to the start position. This value is reset by the commands **zr** and **dp** (see Section 4.3.2).

`STATE_CONF_Y` The integrated y-coordinate of the robot in $\frac{1}{10}$s of inches with respect to the start position. This value is reset by the commands **zr** and **dp** (see Section 4.3.2).

`STATE_CONF_STEER` The orientation of the steering in $\frac{1}{10}$s of degrees with respect to the start orientation, in the range $[0, 3600[$. This value is reset by the commands **zr** and **da** (see Section 4.3.2).

`STATE_CONF_TURRET` The orientation of the turret in $\frac{1}{10}$s of degrees with respect to the start orientation, in the range $[0, 3600[$. This value is reset by the commands **zr** and **da** (see Section 4.3.2).

`STATE_VEL_TRANS` The translational velocity of the robot in $\frac{1}{10}$s of inches per second.

`STATE_VEL_STEER` The steering velocity of the robot in $\frac{1}{10}$s of degrees per second.

`STATE_VEL_TURRET` The turret velocity of the robot in $\frac{1}{10}$s of degrees per second.

`STATE_MOTOR_STATUS` The status of the motors. Only the lowest three bits are relevant: the translational motor is represented by the first or least significant bit, the steering motor is represented by the second bit and the turret motor by the third bit. A bit is set if the corresponding motor is active.

`STATE_LASER` The mode of the laser. See the command **conf_ls** in the Language User Manual for details.

`STATE_COMPASS` The reading of the compass.

`STATE_ERROR` Error number.

The state vector is updated by all robot motion and sensing commands (see Section 4.3). Note, however, that the family of commands that starts with **get_** only updates a part of the state vector. For example, **get_sn** (get sonar) only updates the sonar readings. The command **conf_cp**, which configures the compass, is an exception in that it does not affect the state vector at all. If the user wants to force an update of the state vector, the command `gs()` (get state) can be issued (see Section 4.3).

4.2.2 The Smask Vector

As mentioned above, the state vector is updated by basically every robot command. For applications in which the bandwidth of the communication between application and robot is a crucial issue, this update can be reduced to a subset of the state vector. If some information should not be transmitted from the robot to the application the corresponding value in `Smask` is simply set to zero. All values of `Smask` are initialized to one.

For example, if only the sonars zero through three are used the `Smask` values `SMASK_SONAR_4` to `SMASK_SONAR_15` can be set to zero. Now the values of sonars 4 through 15 in the state vector will not be updated any more.

After the user has made modifications to the `Smask` vector, they have to be transmitted to the robot, otherwise the data that the robot sends will be packaged and unpackaged with different `Smask` vectors. The command `ct` (see Section 4.3.3) sends the current `Smask` vector to the robot and should be issued in order for modifications to become active.

4.2.3 The Laser Vector

The sensory data returned by the laser is stored in the `Laser` vector. The `Laser` vector is updated when the state vector is updated and the value of `Smask[SMASK_LASER]` is one, or when the command `get_ls()` is issued (see Section 4.3.3).

For the operation of the laser three modes are differentiated: *pixel mode, point mode, and line mode*. Note that pixel mode can only be used in conjunction with the real robot; the simulator does not support this mode.

In *pixel mode* the data returned corresponds to either the peak pixel, the rising pixel, the falling pixel, or the magnitude of the signal per scan line of the camera. See the command `conf_ls` in the Language User Manual for details on these different options.

In *point mode* the returned data are pairs that represent points. These points are expressed in a two-dimensional horizontal coordinate system attached to the robot. The x-axis is aligned with the direction of the laser.

In *line mode* the returned data are pairs of points that specify a line segment. Four consecutive values x_1, y_1, x_2, y_2 define a line from (x_1, y_1) to (x_2, y_2). These segments are expressed in the same coordinate systems as the points in point mode.

The first element of the `Laser` vector (`Laser[0]`) always contains the number of data returned. In case of *line mode*, for example, this nuber indicates the count of line segments stored in the subsequent elements of the `Laser` vector (`Laser[1] - Laser[964]`).

4.3 Commands

This section introduces the robot language. It is intended as a general overview. For a detailed description of the commands please refer to the Language User Manual.
Programming the Nomad requires the following steps:

- Establish communication with a robot
- Initialize the robot and its sensors
- Repeat until done:
  - Send motion and sensing commands to the robot
  - Get motion and sensing data from the robot
- Disconnect from robot

This illustrates the use of the three basic classes of robot commands:

1. Communication commands to establish a connection to a robot.
2. Motion commands to move the robot and to obtain its current configuration.
3. Sensing commands to configure the sensors and to receive the sensory data.

In the remainder of this section one subsection is dedicated to each of these groups of commands. A common property of almost all of the commands is that they update the global vector `State` according to the `Smask` vector (see Section 4.2.1 and section 4.2.2). The value returned by the functions themselves is `TRUE` if the command was successfully *transmitted* to the robot, and state information came back correctly. It is *not* an indication that the command was successfully completed. Usually, the user's program will have to monitor the state of the robot (for instance, the robot integrated coordinates, or the robot speeds) to make sure that a command has been successfully executed.

The reason why monitoring of commands is required to determine success is that commands are executed asynchronously: the function itself will return immediately, and while the intended action starts on the robot, the program will move on to the next instruction. For instance, if you send `pr(1000,0,0)`, a command that tells the robot to move forward by 100 inches, the command will return immediately (and probably even before you see the robot actually moving). If your program's next line is `pr(-1000,0,0)`, this will cause the robot to stop the previous motion and start this next one, requesting a move into the opposite direction. The only exceptions are the commands `zr` and `ws`, that initialize the robot's encoders and wait for the robot to stop, respectively. These functions will only return after the execution of the command has been completed.

4.3.1 Communication Commands

The application program can connect to the server or directly to the robot. This is determined by the object file the application is linked with. In case of `Nclient.o` the connection will be established with the server and in case of `Ndirect.o` with the robot.

If the user decides to use the server two variables can be used to specify the connection: `SERVER_MACHINE_NAME` is a string that should contain the name of the machine the server is running on or its IP-address. If that name is `computer`, for example, you have to issue the C-command `strcpy (SERVER_MACHINE_NAME, "computer");` before connecting to the server. Furthermore, the variable `SERV_TCP_PORT` determines to which socket the application program will connect to. For the server and the application to be able to communicate they both have to connect to the same socket. The socket number the server connects to is specified in the file `world.setup`. Assign the number given there to the variable `SERV_TCP_PORT` before connecting to the server.

When connecting directly to the robot the variable `ROBOT_MACHINE_NAME` has to contain the name or the IP address of the robot. This variable can be assigned by using the C-command `strcpy ( SERVER_MACHINE_NAME, "128.12.24.8" );`, assuming that the IP-address of the robot is `128.12.24.8`.

For the case of communication failure, the user can specify a timeout for the robot. This is done with the command `conf_tm`. After the specified time has elapsed the robot will stop; if the application programs crashes this hopefully prevents the robot from crashing into a wall.

▷ `int create_robot ( long robot_id )`
Causes the server to create a new robot with the ID `robot_id`. A robot window and the sensor windows will be created. After the creation of the the robot the application program can connect to it. The return value is the argument upon success, otherwise zero. This command has no effect when connecting directly to the robot using `Ndirect.o`.

▷ `int connect_robot ( long robot_id )`
Establishes a socket connection between application program and server (if using `Nclient.o`) or between application program and robot (if using `Ndirect.o`). Only after this command has been issued successfully commands can be send to the robot. The return value is the argument upon success, otherwise zero.

▷ `int disconnect_robot ( long robot_id )`
Disconnects the application program from the server (if using Nclient.o) of from the robot (if using `Ndirect.o`). The return value is `TRUE` upon success, otherwise `FALSE`.

▷ `int tk ( char *talk_string )` (talk)
If the speech synthesis card DoubleTalk is present this command will send the string `talk_string` to it. With this command you can make the robot speak. This is particularly helpful for debugging.

▷ `unsigned int conf_tm ( int timeout )` (configure timeout)
Sets the timeout of the robot to `timeout` seconds. If the robot does not receive any commands within this period the robot is stopped.

The packets that are send back and forth between the application/server and the robot are of the format listed in table 4.2. Every time a command is issued or values are returned from the robot such a packet is sent.

Name	Description	Size
begin_pkg	always 1	1 byte
length	bytes between begin_pkg and end_pkg	2 bytes
type	ID of the packet	1 byte
data	the data to be transmitted	n bytes
chksum	sum of previous bytes mod 256	1 byte
end_pkg	always 92	1 byte

Table 4.2: The format of the data packages.

Another group of commands is related to packets with content defined and provided by the application program. For details please refer to Section 5.4.

4.3.2 Motion Commands

To fully understand the robot motion commands it is helpful to have some knowledge about the drive system of the robot, which consists of three independent axes. The first axis is the translational axis; it drives the three wheels of the robot that are mechanically coupled to reduce slip. The steering axis changes the direction of the robot's wheels. The wheels remain parallel to each other at any time. This allows the robot to turn in place. The translational axis and the steering axis move the robot in the plane in a non-holonomic fashion.

In addition to the two axes that are responsible for the motion of the base there is a third axis that moves the turret with respect to the base; we refer to it as the turret axis. Using the turret axis enables the user to ((for example) navigate the robot base while maintaining a constant heading with the turret.

All three axes can be controlled in two different control modes: velocity and position control. In velocity control the goal is to maintain a given velocity, whereas position control attains and maintains a given position relative to the current position of the robot.

For all motion commands velocities are specified in $\frac{1}{10}$s of inches per second for the translational axis and in $\frac{1}{10}$s of degrees per second for the steering and the turret axis. Positions are specified relaitve to the current position. Positive velocities and positions for rotations are considered to be counterclockwise; they are relative to the current position of the steering and turret axes. Note that moving the steering axis at constant speed during translation will cause the robot to move on a circular arc.

In the following description of the motion commands `t`, `s`, and `r` are used to indicate that the arguments of the functions refer to the translational, steering, and turret axes, respectively. All commands return `TRUE` upon successful transmission to the robot, `FALSE` otherwise.

▷ `int ac ( int t_ac, int s_ac, int r_ac )` (acceleration)
This command sets the accelerations for the three axis. The unit is the unit of velocity devided by seconds. The acceleration must be lower than:

- 300 $\frac{0.1\mathrm{in}}{s^2}$ for the translational axis,
- 300 $\frac{0.1\mathrm{deg}}{s^2}$ for the steering axis, and
- 300 $\frac{0.1\mathrm{deg}}{s^2}$ for the turret axis.

When doing only very small motions with position control the accelerations should be set to small values; otherwise the robot will accelerate very abruptly, move further than the desired position, accelerate abruptly into the oposite direction to compensate for the error, and so on.

▷ `int sp ( int t_sp, int s_sp, int r_sp )` (speed)
This command sets the maximum speeds for the three axis. The maximum velocities must be lower than

- 200 $\frac{0.1\mathrm{in}}{s}$ for the translational axis,
- 450 $\frac{0.1\mathrm{deg}}{s}$ for the steering axis, and
- 450 $\frac{0.1\mathrm{deg}}{s}$ for the turret axis.

▷ `int vm ( int t_vm, int s_vm, int r_vm )` (velocity move)
This command controlls the three axes of the robot in velocity mode. The arguments to this function are the desired velocities for the three axis; they can be negative but their absolute value has to stay below the maximal value given above (see `sp`). A velocity of zero will maintain the current position.

When a `vm` command is issued the robot will move its axes at the requested velocity and will continue moving unless another motion command is issued, or a timeout occurs (see `conf_tm`).

Note: After a `vm` command the user has to issue the commands `st` (stop, see below) and `ws` (wait for stop, see below), before the robot can be moved with `pr`.

▷ `int pr ( int t_pr, int s_pr, int r_pr )` (position relative)
This command controls the three axes of the robot in position mode. The arguments are the desired position relative to the current one. Note that `pr ( 100, 200, 200 )` will cause all axis to move at the same time. As a result, the motion of the robot will not be a straight line, but a circular arc.

However, if the rotation of the steering is completed before the translation is completed (perhaps due to different speeds or accelerations), the last part of the trajectory will be a straight line. In either case, the length of the trajectory will be 10 inches. Note also that the function call will return immediately. To wait for the motion to be completed the user should issue the command `ws` after the `pr`. A velocity of zero will maintain the current position.
Note: If the command preceeding a `pr` is a `vm`, the commands `st`(stop, see below) and `ws`(wait for stop, see below) have to be issued prior to `pr`.

▷ `int mv ( int t_mode, int t_mv, int s_mode, int s_mv, int r_mode, int r_mv )` (move)
This command allows the user to drive all three axis independently from each other in velocity or position control mode. There are two arguments for each axis:

1. The first argument determines the control mode by specifying
   - `MV_VM` for velocity control,
   - `MV_PR` for position control, and
   - `MV_IGNORE` if the motion for that axis should remain unaltered,

   and
2. The second argument is a velocity if velocity control is the specified mode or a position if position mode was requested.

For example, `pr ( MV_VM, 100, MV_PR, 300, MV_IGNORE, MV_IGNORE)` will cause the robot to start translating at 10 inches per second, the steering will be rotated 30 degrees counterclockwise and the motion that was executed on the turret axis will remain unchanged.

▷ `int st ( void )` (stop)
The robot will stop after this command has been sent. However, since the function call returns immediately, the robot can still be decelerating after the termination of the command. It is recommended to issue a `ws` (see below) after a `st`.

▷ `int ws ( unsigned char t_ws, unsigned char s_ws, unsigned char r_ws, unsigned char timeout )` (wait for stop)
This allows the user to wait for some or all of the axes to be stopped. The first three arguments are `TRUE` if the command should return after the stopping of that particular axis; the argument should be `FALSE` if the status of an axis does not need to be monitored. If not all of the specified axes have stopped before `timeout` seconds have elapsed the function returns.

▷ `int lp ( void )` (limp)
After the robot has stopped it will still try to maintain its position. That means that any attempt to push it will cause the motors to drive the robot

into the opposite direction. This command cause the robot to go in *limp* mode. After executing it, position will no longer be maintained and the robot can be pushed around. This control mode is sometimes referred to as *floating*.

▷ `int zr ( void )` (zero)
Since the turret of the robot can be rotated independently of the base, they can be misaligned. This command causes steering and turret to have the same heading. Therefore, it should be called at the beginning of an application program as part of the initialization. The function call `zr` will only return after the process of zeroing is completed.

▷ `int dp ( int x, int y )` (define position)
Sets the robot position to `(x,y)`.

▷ `int da ( int th, int tu )` (define angles)
The orientation of the steering and the turret is set to `th` and `tu` $\frac{1}{10}$s of degrees, respectively.

▷ `int get_rc ( void )` (get robot configuration)
Issuing this command updates the following entries in the state vector: `STATE_CONF_X, STATE_CONF_Y, STATE_CONF_STEER, STATE_CONF_TURRET`.

▷ `int get_rv ( void )` (get robot velocities)
Issuing this command updates the following entries in the state vector: `STATE_VEL_TRANS, STATE_VEL_STEER, STATE_VEL_TURRET`.

4.3.3 Sensing Commands

For each of the sensors there exists a command to configure it and another one to obtain the readings. In general the sensory readings will be obtained with the `gs` command (see below) that stores readings in the `State` vector using the `Smask` vector. However, dedicated functions exist to save bandwidth.

▷ `int ct ( void )` (transmit change in Smask)
`ct` transmits all changes to the `Smask` vector to the robot and thereby causes them to take effect.

▷ `int gs ( void )` (get state)
Updates the `State` vector according to the `Smask` vector.

▷ `int conf_ir ( int history, int order[16] )` (configure infrareds)
This command will configure the infrared sensors; please refer to the Language User Manual for details on the configuration of the infrared sensors.

▷ `int conf_sn ( int rate, int order[16] )` (configure sonars)
This command will configure the sonars; please refer to the Language User Manual for details on the configuration of the sonars.

▷ `int conf_cp ( int mode )` (configure compass)
This command will configure the compass; please refer to the Language User Manual for details on the configuration of the compass.

▷ `int conf_ls ( unsigned int mode, unsigned int threshold, unsigned int width, unsigned int num_data, unsigned int processing )`
(configure laser)
This command will configure the laser; please refer to the Language User Manual for details on the configuration of the laser.

▷ `int conf_sg ( unsigned int threshold, unsigned int min_points, unsigned int gap )` (configure segmentation data)
This command will configure the segmentation data returned by the laser; please refer to the Language User Manual for details on the configuration of the segmentation data.

▷ `int conf_tm ( unsigned int timeout )` (configure timeout)
Sets the timeout of the robot to `timeout` seconds. If the robot does not receive any commands within this period, the motion will be stopped.

▷ `int get_ir ( void )` (get infrared)
This command will update the infrared data in the `State` vector.

▷ `int get_sn ( void )` (get sonar)
This command will update the infrared data in the `State` vector.

▷ `int get_bp ( void )` (gett bumper)
This command will update the infrared data in the `State` vector.

▷ `int get_cp ( void )` (get compass)
This command will update the infrared data in the `State` vector.

▷ `int get_ls ( void )` (get laser)
This command will update the laser in the `Laser` vector.

▷ `int get_sg ( void )` (get segmentation data)
This command will update the segmentation data in the `Laser` vector.

Note: The remainder of this section refers to commands that are only available in version 2.6.0 or higher.

If the voltage of any of the batteries drops below 10.8V the batteries could be seriously damaged. There are two commands that allow the application program to monitor the voltage.

▷ `float voltCpuGet ( void )`
This returns a float value that corresponds to the voltage at the battery that powers the onboard computer.

▷ `float voltMotorGet ( void )`
This returns a float value that corresponds to the voltage at the battery that powers the motors of the robot driving system.

Since sensor readings are not instantaneous and the robot's position or the environment can change significantly between acquisition and processing of the sensory data, the sensor reading might be inaccurate when processed. There is another group of commands that allow the user to deal with this source of error; they are explained in detail in Section 5.2.

4.3.4 Server Commands

The commands introduced in this section require the application to be connected to a server. Called from programs that connect directly to the robot (linking with `Ndirect.o`) they will have no effect. For a more elaborate description of these commands please refer to the Language User Manual.

Commands related to the server:

▷ `server_is_running ( void )` Returns `TRUE` if a server is connected to the same socket the application is connected to.

▷ `quit_server ( void )` Called from an application that is connected to a server, this command will cause the server to exit.

Commands related to the world representation: The application program can modify the world data base of the server. The polygons that are passed as arguments to the following functions have to be convex and the points are given in counterclockwise order.

▷ `add_obstacle ( long obs[2*MAX_VERTICES+1] )` Add an obstacle to the world representation of the server.

▷ `delete_obstacle ( long obs[2*MAX_VERTICES+1] )` Delete an obstacle from the world representation of the server.

▷ `move_obstacle ( long obs[2*MAX_VERTICES+1], long dx, long dy )`
Translate an obstacle of the server's world representation by `(dx, dy)`.

Commands related to drawing in the map window:

▷ `draw_robot ( long x, long y, int th, int tu, int mode )` This command draws a robot at the given configuration into the ROBOT window of the server. For a description of the arguments and `mode` refer to the Language User Manual.

▷ `draw_line ( long x_1, long y_1, long x_2, long y_2, int mode )` This command draws a line into the ROBOT window of the server. For a description of the arguments and `mode` refer to the Language User Manual.

▷ `draw_arc ( long x_0, long y_0, long w, long h, int th1, int th2, int mode )` This command draws a circular arc into the map window of the server. For a description of the arguments and `mode` refer to the Language User Manual.

▷ `place_robot ( int x, int y, int th, int tu )` With this command the client program can request the user to input a robot position via the graphic interface.

Commands to switch between real and simulated robot:

▷ `real_robot ( void )` Called from an application program, this command causes the server to send the commands (from the application program) to the real robot instead of sending them to the simulated robot.

▷ `simulated_robot ( void )` This command has the inverse effect of `real_robot`.

4.4 Some Examples

In this section, the development of a small application program is presented. Step by step, we are going to increase its capabilities to demonstrate how to program the robot. The first program in Figure 4.3 just connects to the robot, initializes it, moves 10 inches forward, and finally disconnects again. It will be the starting point for further refinement. For information on how to compile programs and how to run them in connection with the server please refer to Section 2.5 in Chapter 2; you will have to adapt the first two lines of the procedure `main` to adjust the communication parameters to your specific environment. This program should be used in conjunction with a server. The user can then choose if the real or the simulted robot will be used.

If your program connects to the real robot you should see the robot move ten inches forward. Nothing will happen if you use the simulated robot. The reason for this is that the simulator only updates the screen when the state vector is updated. The simulator basically displays the robot as represented by the state vector. If you manually issue a `gs` command in the command line window you will be able to see the result of this little program.

Now we will extend this program to move the robot in a square. While it is moving it will monitor the bumpers and if one of the bumpers is hit the program will abort. The turret will maintain a constant orientation.

To make the robot move in a square we will issue eight motion commands: first a translation, then a steering rotation, repeated for the four sides of the square. However, `pr` does not wait for the current motion to be completed, so the user must be careful not to overwrite a commanded motion before it is completed. We will implement a function that monitors the velocity of a given axis. This will allow the program to determine when a motion has been completed. In order to do so we could use the command `ws`, but we want

```
/*
 *  Example 1 Version 1
 *
 *  This program will connect to the robot, initialize it,
 *  move in a square monitoring the bumpers, and disconnect.
 *
 */

#include "Nclient.h"

#define ROBOT_ID   1        /* the ID of the robot */

void main ( void )
{
    int i;

    /* setup connection parameters */
    SERV_TCP_PORT = 7019;
    strcpy ( SERVER_MACHINE_NAME, "computer");

    /* establish connection */
    if ( !connect_robot ( ROBOT_ID ) )
       exit ( 0 );

    /* initialize robot */
    conf_tm ( 2 );
    zr ();
    ac ( 200, 300, 300 );
    sp ( 100, 200, 200 );

    /* move robot 10 inches */
    pr ( 100, 0, 0 );

    /* disconnect */
    disconnect_robot ( ROBOT_ID );
}
```

Figure 4.3: The first example.

our program to issue the command `gs` while waiting so that the simulator gets updated and we are able to follow the motion on the screen.

```
void wait_for_stop ( int axis )
{
    /* wait for motion to begin */
    while ( State [ axis ] == 0 )
        gs();
    /* wait for motion to end   */
    while ( State [ axis ] != 0 )
        gs();
}
```

The above procedure requires as an argument the index to the `State` vector that contains the velocity of the axis to be monitored. It first waits for the motion to be started by remaining in the first `while`-loop until the velocity differs from zero. Then it waits for the motion to be completed; this is recognized by the velocity becoming zero again. Inside both `while`-loops the command `gs()` is executed to update the `State` vector, and therefore the simulator screen.

We will now extend the procedure `wait_for_stop` to monitor the bumper while waiting for the motion to be completed. This can be realized using the bumper reading in the state vector.

```
int wait_for_stop_or_bumper ( int axis )
{
    /* wait for motion to begin */
    while ( ( State [ axis ] == 0 ) && ( State [ STATE_BUMPER ] == 0 ) )
        gs();
    if ( State [ STATE_BUMPER ] != 0 )
        return ( FALSE );

    /* wait for motion to end   */
    while ( ( State [ axis ] != 0 ) && ( State [ STATE_BUMPER ] == 0 ) )
        gs();
    if ( State [ STATE_BUMPER ] != 0 )
        return ( FALSE );
    return ( TRUE );
}
```

Now the `while`-loop is also terminated when the bumper data assumes a value different from zero, which indicates a bumper hit. The new procecure `wait_for_stop_or_bumper` returns `TRUE` if the motion was completed without a bumper hit and it returns `FALSE` if a bumper was hit. This allows us to write the second version of our example program, listed in Figure 4.4. In that listing the code for the procedure `wait_for_stop_or_bumper` has been left out for brevity.

```
/*
 *  Example 1 Version 2
 *
 *  This program will connect to the robot, initialize it,
 *  move in a square monitoring the bumpers, and disconnect.
 */

#include "Nclient.h"

#define ROBOT_ID   1        /* the ID of the robot */

int wait_for_stop_or_bumper ( int axis )
{  . . . }

void main ( void )
{
    int i;

    /* setup connection parameters */
    SERV_TCP_PORT = 7019;
    strcpy ( SERVER_MACHINE_NAME, "zulu");

    /* establish connection */
    if ( !connect_robot ( ROBOT_ID ) )
       exit ( 0 );

    /* initialize robot */
    conf_tm ( 2 );
    zr ();
    ac ( 200, 300, 300 );
    sp ( 100, 200, 200 );

    /* move in a square */
    for ( i = 0; i < 4; i ++ )
      {
         /* move straight */
         pr ( 200, 0, 0 );
         if ( !wait_for_stop_or_bumper ( STATE_VEL_TRANS ) )
            break;
         /* turn 90 degrees */
         pr ( 0 , 900, 0 );
         if ( !wait_for_stop_or_bumper ( STATE_VEL_STEER ) )
            break;
      }
    /* stop robot */
    st();
    /* disconnect */
    disconnect_robot ( ROBOT_ID );
}
```

Figure 4.4: The second version of the first example

Having gained some familiarity with the robot, we are now going to develop a *wanderer* program. This program will cause the robot to wander around in an unknown environment. It will move forward until the front sonar detect an obstacle, then it will choose a new direction for motion based on the reading of the sonar to the rear and on the side. First, we will develop a few procedures that will be helpful for the wanderer program.

```
/* returns the minimum sonar reading of the front 5 sonars */
int get_min_dist ( void )
{
   int i;
   int min_value = 255;    /* max. sonar reading */

   /* use sonars 0,1,and 2 and remember minimum value */
   for ( i = 0; i < 3; i++ )
      if ( State [ STATE_SONAR_0 + i ] < min_value )
          min_value = State [ STATE_SONAR_0 + i ];

   /* use sonars 14 and 15 and remember minimum value */
   for ( i = 0; i < 2; i++ )
      if ( State [ STATE_SONAR_14 + i ] < min_value )
          min_value = State [ STATE_SONAR_14 + i ];

   return ( min_value );
}
```

The function `get_min_sonar` uses sonars 1, 2, 3, 15, and 16. It determines the shortest sonar reading and remembers it in `min_value`. Once all five sonar are examined, the smallest value is returned. This will help us in determining if an obstacle is in our way. Once we have determined that this is the case we want to choose another direction to wander. That can be done with the following procedure:

```
/* returns the angle that is the freest direction */
int get_best_dir ( void )
{
   int i;
   int max_value = 0;      /* min. sonar reading        */
   int best_dir;           /* to remember the direction */

   /* use sonars 3 through 13 */
   for ( i = STATE_SONAR_3; i <= STATE_SONAR_13 ; i++ )
      if ( State [ i ] > max_value )
        {
          /* remember value and direction */
          max_value = State [ i ];
          best_dir  = i - STATE_SONAR_0;
```

```
      }

   /* do we want to rotate left or right ? */
   if ( best_dir > 8 )
      best_dir = best_dir - 16;

   /* return angle to rotate in 1/10 of degrees if enough space */
   if ( max_value > 30 )
       return ( best_dir * 225 );
   else
       return ( -1 );
}
```

This function returns the angle in $\frac{1}{10}$s of degrees that we should rotate in order to be facing into the direction with the most free space. In the `for`-loop we find the largest sonar reading among the sonars 4 through 14; we remember that value in the variable `max_value`. The variable `best_dir` stores the number of the sonar that had this value as its reading. In order to do so we have to subtract `STATE_SONAR_0`, otherwise we would store the index of that sonar in the state vector.

After these values have been determined, we can compute the angle that we have to rotate the robot by to face into the freest direction. However, since it is faster to rotate clockwise by 10 degrees that it is to rotate counterclockwise by 350 degrees we change `best_dir` to the appropriate negative number if necessary.

Since there should be a way to indicate that there is no free space at all, the function will return -1 if the largest sonar reading does not exceed 30. Otherwise the angle is returned. As two neighboring sonar have an angle of 22.5 degrees, we simply have to multiply by 225 to obtain the value in $\frac{1}{10}$s of degrees.

Using these two functions and the `wait_for_stop` routine from the previous example the *wanderer* looks as follows (again, for brevity the bodies of the procedures are omitted):

```
/*
 *  Example 2  The wanderer
 *
 *  This program will let the robot wander around.
 */

#include "Nclient.h"

#define ROBOT_ID 1

/* returns the minimum sonar reading of the front 5 sonars */
int get_min_dist ( void )
{ . . . }
```

```
/* returns the angle that is the freest direction */
int get_best_dir ( void )
{ . . . }

/* wait for a certain axis to stop */
void wait_for_stop ( int axis )
{ . . . }

void main ( void )
{
    int i;
    int angle;

    /* setup connection parameters */
    SERV_TCP_PORT = 7019;
    strcpy ( SERVER_MACHINE_NAME, "computer");

    /* establish connection */
    if ( !connect_robot ( ROBOT_ID ) )
       exit ( 0 );

    /* initialize robot */
    conf_tm ( 2 );
    zr ();
    ac ( 200, 300, 300 );
    sp ( 100, 200, 200 );

    /* as long as there is free space */
    while ( ( angle = get_best_dir () ) != -1 )
      {
          /* turn into that direction */
          pr ( 0, angle, angle );
          wait_for_stop ( STATE_VEL_STEER );

          /* start wandering and monitor sonars */
          vm ( 100, 0, 0 );
          while ( get_min_dist () > 30 )
             gs();

          /* stop the robot and wait for it to be stopped */
          st();
          ws ( TRUE, TRUE, TRUE, 3 );
      }
    /* disconnect */
    disconnect_robot ( ROBOT_ID );
}
```

This little program tries to keep a safe distance from obstacles. You might want to try to improve it.

Chapter 5

Advanced Features

5.1 Using the Sensors

5.1.1 The Infrared Sensors

The Sensus 300 is a ring of 16 reflective intensity based range detectors. It is used for obstacle detection in the range 0-36 inches. The 16 sensors can provide 360 degrees coverage.

Hardware

Each sensor is composed of two SIEMENS SFH 434-3 GaAIAs infrared emitters and a SIEMENS SFH 2030F Silicon Pin Photodiode enclosed in a delrin housing. The two emitters and the receiver are mounted horizontally, the diode between the two emitters. Range to the object(s) is determined by the intensity of the light from the emitter reflected back to the detector. The intensity of the light is digitally measured using a Voltage Controller Oscillator, a device which emits a square wave signal with a frequency proportional to its input voltage. The higher the input voltage, the faster the oscillation.

Software

The read cycle contains two readings: the first reading is done without emitting; its purpose is to measure the ambient light level. The second reading is done while the emitter is on. The difference between the reading with the emitter on and the emitter off yields a value which is dependent only on the light from the emitter. However, as the experimentation will show, the signal/noise ratio is still affected by the ambient lighting level, and more specifically by the amount of infrared in the ambient light. The measurement time actually varies with the level of the reflected light (the intensity is determined by the amount of time necessary to get a fixed number of pulses from the VCO). A weak input will take more time than a strong input. The order of magnitude for standard conditions (normal office lighting, white wall at a distance of 16 inches) is 0.0025 seconds per measurement cycle.

Factors influencing the measurements

The factors that influence the measurement obtained from the sensors are the following:

- Geometry

  The farther the object, the weaker the reflected light coming back to the receiver will be: this is the underlying principle of this sensor. Additionally, the angle of the reflecting object also plays an important role. Let us assume that the object is planar and infinite i.e. it intercepts all the light emitted by the sensor. Practically speaking, this means that the plane is big enough to contain the entire infrared light spot. When illuminated, part of the incoming energy is reflected, and part of it is diffracted. If the illuminated plane is normal, or close to normal to the receiver's axis, most of the reflected light will be directed back to it. On the contrary, if the illuminated plane is close to parallel to the emitter's axis, then only the diffracted light will hit the sensor back. Distance and Angle are then the two most important characteristics that affect the readings.

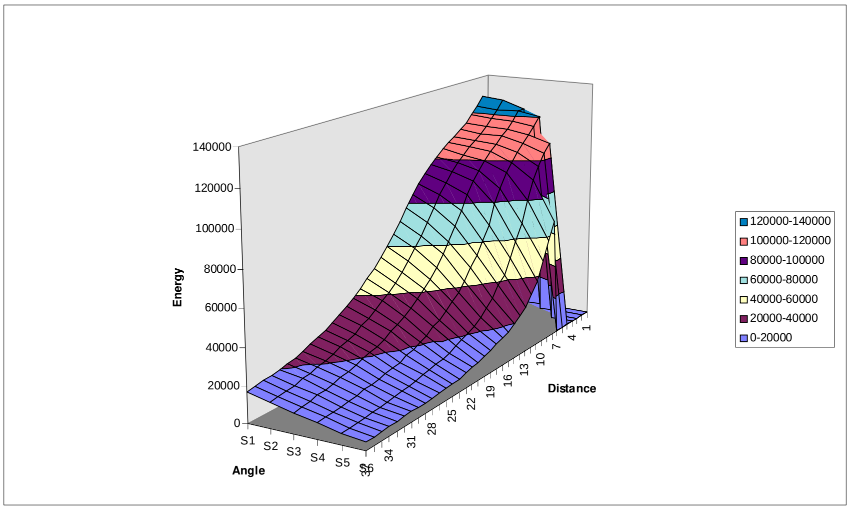

Figure 5.1: Energy vs Angle and Distance: white dull surface in dark

  Figure 5.1 shows the raw energy level measured at distances from 0 to 36 inches, for angles from 0 to 75 degrees. Measurements were done in the dark, with a white dull target (regular copy paper).

  If we abandon the infinite plane hypothesis, then the geometry of the object itself has an influence. First, still assuming a planar object, it can be only partially illuminated. The reading will then be dependant on the illuminated area. Second, if the object is not planar, then the light reflected back to the sensor will be a complex combination of the reflected/diffracted energy coming from all the illuminated parts of the object.

- Lighting

  The sensor computes the difference between the light received with emitter off, then on. If the ambient light has a lot of energy in the infrared range, then the difference will be small (weak contrast), and the sensor will be less sensitive than in a low-infrared environment.

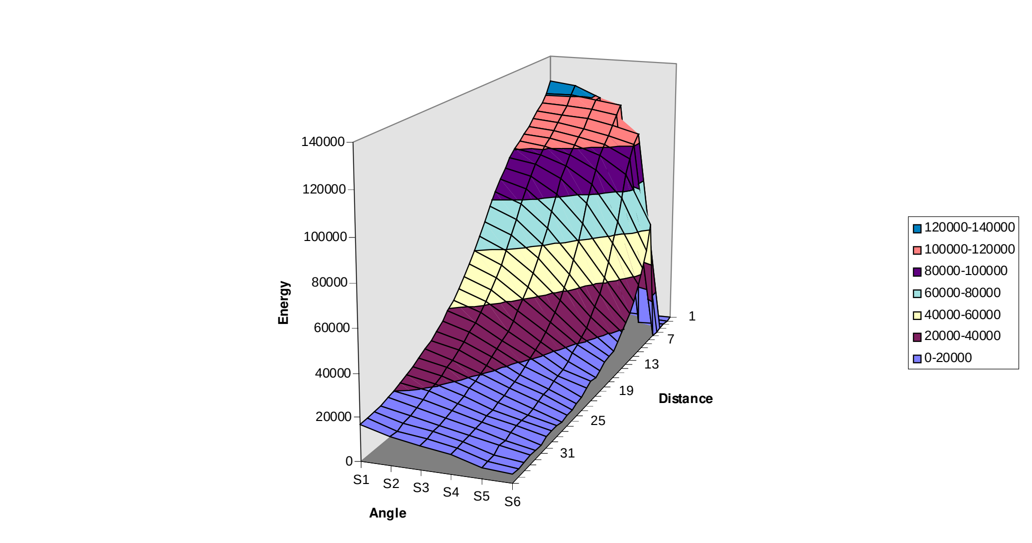

Figure 5.2: Energy vs Angle and Distance: white dull surface in fluorescent light

The data of Figure 5.2 were recorded in fluorescent light; the data of Figure 5.3 were recorded in incandescent light. Compare this with Figure 5.1: since fluorescent light contains very little infrared, the energy values are not much affected. They are affected under incandescent light (sunlight gives similar results).

- Color and surface.

  The color (or more precisely the reflectivity in infrared range) of the objects to have an influence on the readings. The surface has also to be taken into account: Glossy surfaces will show a peak in intensity for the angles close to the normal to the receiver's axis, because they behave like mirrors. Surfaces like concrete or fabric will absorb more. The chart of Figure 5.4 shows the energy measured at 0 degrees (the object surface normal to the

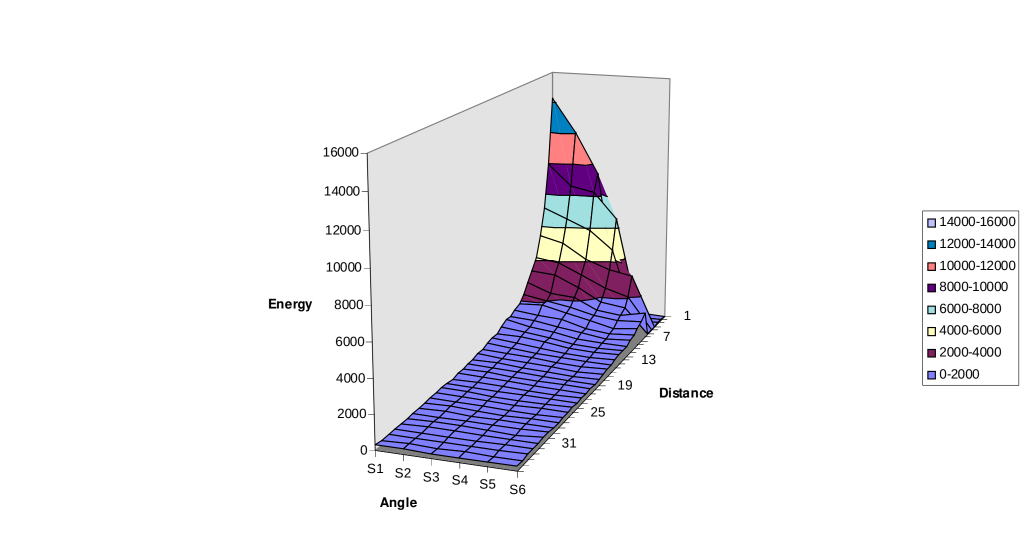

Figure 5.3: Energy vs Angle and Distance: white dull surface in incandescent light

sensor's axis), for a black metallic cabinet (black and glossy), a piece of black fabric (black and dull), a white board (white and glossy) and a sheet of paper (white and dull) [1].

The infrared calibration values stored in the file `infcal.cfg` on the robot (`/usr/local/share/robotd`); this file looks like this:

```
7500     /* values below this will be translate to a 15 */
15000    /* values below this will be translate to a 14 */
22500    /* values below this will be translate to a 13 */
30000    /* values below this will be translate to a 12 */
37500    /* values below this will be translate to a 11 */
45000    /* values below this will be translate to a 10 */
52500     /* values below this will be translate to a 9 */
60000     /* values below this will be translate to a 8 */
67500     /* values below this will be translate to a 7 */
75000     /* values below this will be translate to a 6 */
82500     /* values below this will be translate to a 5 */
90000     /* values below this will be translate to a 4 */
97500     /* values below this will be translate to a 3 */
```

[1] Objects sometimes behave differently in infrared and visible light. For instance, we have observed that two pieces of fabric identically dark in normal light behave very differently in infrared: one staying black, and the other becoming brightly white. Rigorously, one cannot tell the infrared reflectivity of a material from its white light color.

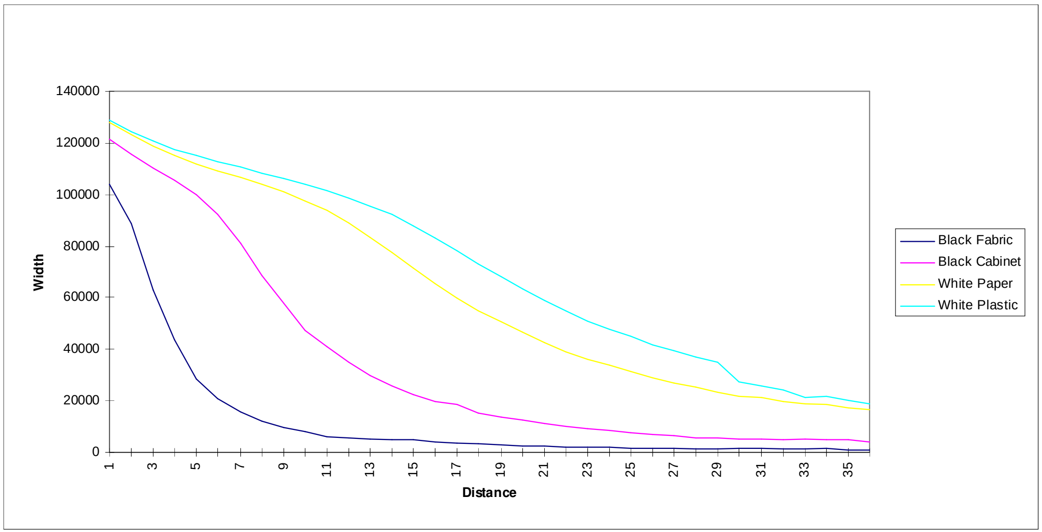

Figure 5.4: Energy vs Distance at Angle 0, Black/White, Dull/Glossy

```
105000    /* values below this will be translate to a 2 */
112500    /* values below this will be translate to a 1 */
120000    /* values below this will be translate to a 0 */
```

The values in this file have been determined as an average of the observed values for office-type environment under fluorescent light. You may try to determine your own values: you can get the raw infrared intensity values using the user packet communications as explained in section 5.4.

The language commands related to infrared sensors are `conf_ir`, `get_ir`; the infrared data is stored in the `State` vector, at indexes ranging from `STATE_IR_0` to STATE_IR_15. Refer to chapter 4, and to the Language Reference Manual.

5.1.2 The Sonar Sensors

Sensor Description

The Sensus 200 is a ring of 16 Polaroid 6500 sonar ranging modules. The Polaroid 6500 is an acoustic range finding device that has been widely used in the mobile robotics community. It can measure distances from 6 inches to 35 feet, with a typical absolute accuracy of ±1 percent over the entire range.

The standard Polaroid system in long-range mode (the one currently used on the Nomad robot) emits 56 cycles of a 49.4 kHz square wave through a transducer. A blanking period follows, during which the internal circuitry is reset and stabilized. The transducer then acts as a receiver, feeding the detected echoes in a time variable gain amplifier. The gain factor of this device increases with time to compensate for spreading loss and the attenuation of sound in air. The output of the amplifier then goes to a thresholding circuit. As soon as the threshold is exceeded, the time elapsed since the beginning of the transmission of the pulse is measured, and converted into distance through an appropriate calibration factor.

Sensor Physics

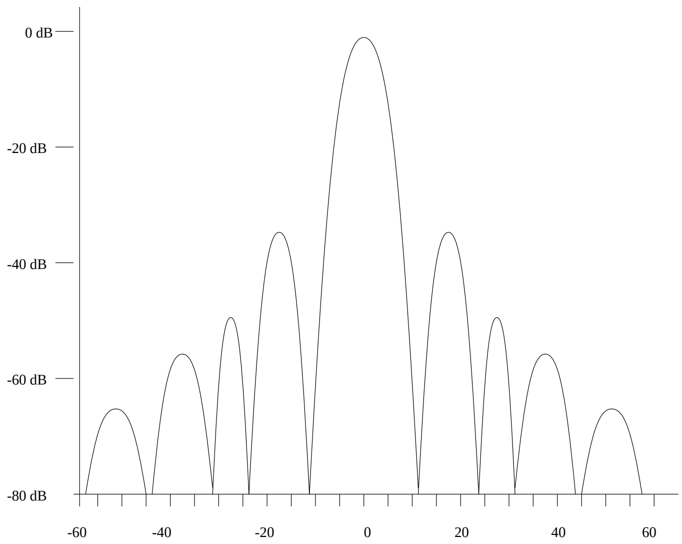

Figure 5.5: Modelized sound radiation for the Polaroid Transducer (from Leonard)

Sound characteristics The transducer does not emits energy homogenously in all directions, but instead form lobes of decreasing intensity, as illustrated in the chart of Figure 5.5[2].

According to the results of extensive experiments conducted by various researchers (Lang, Kuc, Leonard), the radiation pattern of Polaroid Transducers is not symmetric, and varies from one transducer to another. These effects are more significant for the side lobes.

The attenuation of ultrasound in air increases with frequency, and depends on temperature and humidity. Typical values for 50 Hz are an attenuation of

[2] Most of the information relative to sonars in this section come from Leonard J., Durrant-Whyte H., Directed Sonar Sensing for Mobile Robot Navigation, Kluwer Academic Publishers, 1992

0.6 to 1.8 dB/m for variations in temperature from 17 to 28 degrees C and variation in relative humidity from 15 to 70 percent. The speed of sound in air is expressed as $c = 331.4\sqrt{T/273}m/sec$, T being the ambient temperature in degrees K.

Electronics characteristics The following three sources of errors in range measurements are related to the specifics of the sensor circuitry (Leonard):

- Transmitted Pulse duration
  All the timing is based on the assumption that the start of the transmitted pulse is the part of the returned echo that actually exceeds the detector threshold. If this is not the case, the error can be as much as 8 inches.

- Time Variable gain amplifier
  The ideal exponential curve that would exactly cancel beam spread and attenuation losses is approximated by a 12-step only piecewise constant function. Even if this function was exactly given, it should be different according to temperature and humidity conditions. Since the returned energy is function of the incident angle (as shown in the radiation pattern), the visibility angle changes with range.

- Capacitive charge-up in the threshold circuit.
  For strong reflected signals, 3 cycles are enough to charge up to the threshold: the calibration usually accomodates that delay. For weaker signals, charge-ups can take place over a considerably longer time, resulting in erroneously elongated range values.

One major source of uncertainty in distance estimation coming from these characteristics is the existence of weak returns, as opposed to strong returns:

- Strong returns possess enough energy to exceed the threshold promptly, giving very accurate measurements,

- Weak returns cause time-delay range errors: the threshold is reached only after a long charge-up and changing gain in the amplifier. The threshold is only exceeded by the random combination of a slow charging-up period and jumps in the non-linear time variable gain amplifier.

Target characteristics

Targets can be divided in two groups:

- Reflecting objects, of dimensions larger than the wavelength (6.95 mm at 20 degrees C),

- Diffracting objects, of dimensions smaller than the wavelength.

Objects whose overall size is smaller than the wavelength are usually rare (one can think of wire fences for instance), but rough surfaces like concrete, or textured walls present small asperities which actually behave as diffractors, as the experimental data will show. Smooth surfaces like metallic desks, painted walls, doors, etc, are reflectors. Those are the most common in indoor environments, which supports the commonly heard assertion that most indoor surfaces act as mirrors with respect to sound waves. Slightly rounded convex edges with radius of the order of the wavelength produce weak specular echoes: some ornamental (carved) table legs and cardboard boxes often have this character.

One consequence of the reflective properties of surfaces is the multiple echo effect: the sound wave bounces around, and eventually reaches the receiver after several reflexions: because of attenuation, the energy level of the incoming wave is very likely to be quite low, one possible origin of what is called a "weak return".

Typical Sonar Data

A good representation of sonar data is the sonar scan: a dot representing the measured distance is drawn on the sonar axis. Repeated measurements over a given angular range, for instance one every degree over the whole circle, give a sonar scan. Sonar scans are the basic data that can be used for map building or navigation purposes. Since the sonar are mounted horizontally on the robot turret, which can itself be oriented to any angle, one of them can be used directly to make measurements, using the robot to set it at desired angle and distance with respect to the targets.

- Smooth flat surface

  The scan of Figure 5.6 is typical of the response of a flat, smooth surface. The central feature is an arc of circle, centered with respect to the surface's normal. This pattern is caused by the main lobe: although the sonar is rotating, as long as there is in the active cone of the main lobe a patch of surface normal to the direction of the wave, it gets reflected back. Since we are in the main lobe, where energy is high, the thresholding circuit is promptly triggered, and the measurement is very precise: we get an arc of circle of constant radius.

  When the normal to the surface leaves the main lobe, the energy is reflected away from the source. However, the secondary lobe comes into effect, with the same property: if the normal to the surface is within the secondary lobe, the sound gets reflected. Since the secondary lobe carries less energy than the main lobe, we have a time-delay range error because the thresholding circuit takes more time to be triggered. We also get an arc of circle, but of slightly greater radius. Note that the secondary lobe dots may be accidentally aligned with the actual wall, which in the past

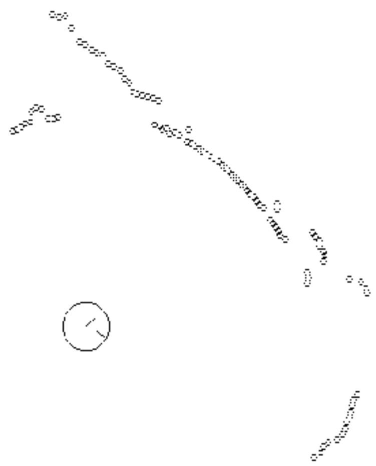

Figure 5.6: Sonar Scan on a flat and smooth surface

have induced people to try to match lines to sonar readings, but this is purely coincidental: physics of the sonar dictates that circles should be fit to sonar scans.

The sonar rotating further, the secondary lobe also leaves the normal to the surface: nothing can be seen any more (there are actually other lobes to follow, until the whole half circle, but they are too weak to give an answer), and the plots are drawn to infinity (equal to the maximum detectable distance). One interesting phenomenon can be observed at the junction of the main and secondary lobe: provided that the scan is detailed enough, one consistently gets readings that are extremely erroneous, more than twice the actual distance. These are caused by some low energy coming at the junction of the two lobes. Following the acoustic model, this should be zero, but the experimental curves provided by the maker show that there is actually some remaining energy coming out when switching from one lobe to another. Being a very weak return, this gives a very erroneous reading, sometimes even missing.

- Rough flat surface

  The scan of Figure 5.7 is typical of a rough surface. It can be understood if we consider that a rough surface is made of numerous small asperities, of dimension close to the sound wavelength. Thus, the primary lobe always picks a small patch normal to its axis, and return the corresponding distance. This is the only case where the sonar scans bears a strong

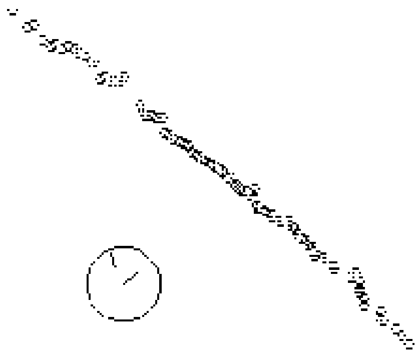

Figure 5.7: Sonar scan on a flat and rough surface

resemblance with the actual map of the room.

- Corner

  The scan of Figure 5.8 is typical of a smooth corner. The central pattern is an arc of circle centered about the corner. This arc comes from the reflection of sound on the corner, which has dimensions close to the wavelength and therefore acts as a diffractor. We also have two neighbohring arcs that we can attribute to the secondary lobes. The time-delay is bigger than in the smooth surface case, probably because a diffracting edge returns less energy than a flat surface at the same distance. The leftmost and rightmost arcs come from specular reflection on the corner's sides. One interesting point is that the corner's side reflexions could also give secondary lobe arcs, if the sonar was a bit farther from the corner. In the present case, it happens to be that these secondary lobe arcs actually come from the corner, as we verified by unfolding it (then the arcs disappeared).

  Note that when the sonar is close to the corner, it is difficult to predict where the returned echo is actually coming from, because there is a competition between a weak, but specular echo (the secondary lobe on the corner side), and a strong, but distant and diffracted echo (the main lobe on the corner). Additionally, there may be multiple echo phenomena that can create additional error, depending whether the first echo to trigger the threshold circuit comes from a single-hit echo, or from multiple reflections on the corner sides.

Figure 5.8: Sonar scan on a corner

The language commands related to sonar sensors are `conf_sn`, `get_sn`; the infrared data is stored in the `State` vector, at indexes ranging from `STATE_SONAR_0` to `STATE_SONAR_15`. Refer to chapter 4, and to the Language Reference Manual.

5.1.3 The Laser Range Finder

For this sensor, please refer to the Sensus 500 User's Manual that comes with the Laser Range Finder.

5.2 Position Attachment

The Nomad 200, when fully equipped, includes five different types of sensors, infrared sensors, sonar, a bumper, a compass, and a laser range finder. Since sensor readings are not instantaneous and the robot's position or the environment can change significantly between acquisition and processing of the sensory data, the sensor reading might be inaccurate when processed. By attaching to each sensor reading a time stamp and the position the robot was at when the reading was acquired, the user is able to account for this source of error.

5.2.1 The Type `PosData`

To allow for easy access to the position information a new type of variables is introduced. In the file `Nclient.h` the type `PosData` is defined as follows:

```
typedef struct _PosData
```

```
{
    /* the configuration of the robot at the time of the reading */
    ConfigData config;

    /* the time of the sensing in milliseconds (Intellisys 100 time) */
    TimeData   timeStamp;

} PosData;
```

The configuration of the robot is stored in a variable of type`ConfigData` which is also defined in `Nclient.h`:

```
typedef struct _ConfigData
{
    /* the configuration of the robot */
    long          configX;
    long          configY;
    long          configSteer;
    long          configTurret;

    /* the velocities of the robot*/
    long          velTrans;
    long          velSteer;
    long          velTurret;

    /* time of integration in milliseconds (Intellisys 100 time) */
    TimeData  timeStamp;

} ConfigData;
```

The robot's configuration consists of its position in the world coordinate frame, the steering orientation, and the orientation of the turret. Furthermore, the translational velocity and the angular steering and turret velocities are included in the configuration. Finally, a time stamp is attached to the configuration. This time stamp denotes the time at which the computation of the configuration data was performed. This time is referred to as *integration time*.

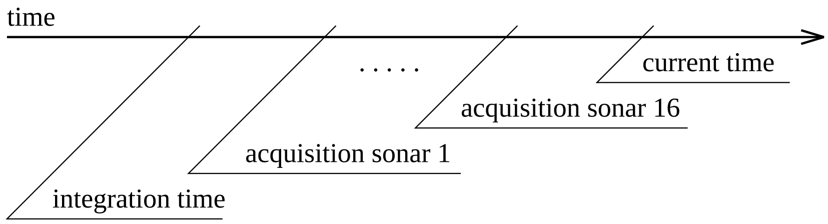

Figure 5.9: Sequence of events

The position attachment itself consists of a configuration and a time stamp denoting the time of the sensor acquisition. This time stamp will sometimes differ from the time stamp of the configuration, because the configuration was computed prior to the sensor acquisition. This time of acquisition of the sensory data is referred to as *acquisition time*.

Both time stamps are in milliseconds of an internal clock of the Intellisys 100 board, which is the board that handles the sensors. This clock starts at zero when the power is turned on and is incremented every 16.4 milliseconds, therefore the resolution of the time stamps is roughly 16 milliseconds. The clock will roll over after 49 days.

The user can call `int posTimeGet ( void )` to obtain the latest information on the Intellisys 100 time. This time is set to the current Intellisys 100 time during the execution of a command that updates the state vector, like for example `gs()`, `get_ir()`, `get_sn()`, `get_bp`, and `get_cp()`. To obtain an accurate time value the command preceeding `posTimeGet` should be one that updates the state vector.

5.2.2 Requesting the Attachment of Position Data

With a call to `int posDataRequest ( int posRequest )` the attachment of position information to sensory readings can be requested. In `Nclient.h` the following values are defined:

```
#define POS_NONE        ( 0 << 0 )
#define POS_INFRARED    ( 1 << 0 )
#define POS_SONAR       ( 1 << 1 )
#define POS_BUMPER      ( 1 << 2 )
#define POS_LASER       ( 1 << 3 )
#define POS_COMPASS     ( 1 << 4 )
```

The logical disjunct of these values is passed as an argument to `posDataRequest`. For example, position attachment for sonars and bumper would be requested by calling `posDataRequest ( POS_SONAR | POS_BUMPER )`.

Once this attachment is requested it will be updated each time the corresponding sensors are read. This feature can be turned off by calling `posDataRequest ( POS_NONE )`, or it can be turned off selectively for the bumper, for example, by calling `posDataRequest ( POS_SONAR )`.

The function `int posDataCheck ( void )` returns the argument previously passed to `posDataRequest`. The user can inquire the status of the position attachment with the macros defined in `Nclient.h`:

```
#define POS_INFRARED_P  ( ( (posDataCheck()) & POS_INFRARED ) ? 1 : 0 )
#define POS_SONAR_P     ( ( (posDataCheck()) & POS_SONAR    ) ? 1 : 0 )
#define POS_BUMPER_P    ( ( (posDataCheck()) & POS_BUMPER   ) ? 1 : 0 )
#define POS_LASER_P     ( ( (posDataCheck()) & POS_LASER    ) ? 1 : 0 )
#define POS_COMPASS_P   ( ( (posDataCheck()) & POS_COMPASS  ) ? 1 : 0 )
```

For example:

```
if ( POS_INFRARED_P || POS_SONAR_P )
    printf("Position attachment for IR or sonar turned on.\n");
```

The `printf` statement is only executed if the positition attachment is requested for either the infrared sensors or the sonars, or both.

5.2.3 Getting the Position Data

For each sensor type there is a dedicated function to obtain the position information. When calling these functions a pointer to a `PosData` variable is passed. If the attachment of position information was requested for that particular sensor, upon return of the function this variable will be filled with the newest position information for the latest sensor reading. Since there are sixteen infrared sensors and sixteen sonars, for these sensors a number has to be specified also. When passing this numbers the defined values from `Nclient.h` should be used:

```
#define POS_IR_0            0
#define POS_IR_1            1
#define POS_IR_2            2
:
#define POS_IR_15          15

#define POS_SONAR_0         0
:
```

Furthermore, for the infrared sensors and the sonars there exist functions that require as an argument an array of sixteen variables of type `PosData`. This allows the user to obtain the position attachment for infrareds / sonars with one function call.

In summary, for obtaining the position attachment for sensory readings the following functions can be used:

```
/* for the infrared/sonars with two arguments */
int posInfraredGet     ( PosData *posData, int infraredNumber );
int posSonarGet        ( PosData *posData , int sonarNumber)

/* for the bumper, compass, and laser with one argument */
int posBumperGet       ( PosData *posData );
int posCompassGet      ( PosData *posData );
int posLaserGet        ( PosData *posData );

/* to read all infrareds/sonars at once */
int posInfraredRingGet ( PosData posDataArray[INFRAREDS] );
int posSonarRingGet    ( PosData posDataArray[SONARS] );
```

5.2.4 A Simple Example

Here is an example program that prints out the position attachment for all the sonar readings and for the bumper.

```
/*
 *  This is an example program to demonstrate the use of the
 *  position attachment for sensory data. Save it under the
 *  name 'posDemo.c'. To compile it type
 *
 *       gcc -o posDemo posDemo.c Nclient.o
 *  OR   gcc -o posDemo posDemo.c Ndirect.o
 *
 *  The files Nclient.h and Nclient.o have to be in the same
 *  directory as posDemo.c. The robot and the server have to
 *  be running. For this demonstration to be meaningful the
 *  robot has to be moving. Use the joystick to move the robot
 *  during the execution of this program. Before compiling
 *  change the string 'your_machine_name' and the number 7019
 *  to the name of your host machine and the port number the
 *  server is using if you are using the server; also change
 *  the string 'your_robot_name' to the name of your robot
 *  computer, if you are directly communicating to the robot.
 *
 */

#include "Nclient.h"

/* to convert from tenth of inches to meters */
#define  TENTH_IN_TO_CM   0.254
#define  ROBOT_ID         1
#define  SONAR_INDEX      5         /* pick one from the 16 */

void main ( void )
{
    int i;
    PosData posBumper;
    PosData posSonarArray[16];
    float   ageSensoryReading;
    float   ageIntegrationAtAcquisition;
    float   distMovedSinceAcquisition;

    /* adopt to your system configuration */

    /* when using Nclient.o */
    strcpy ( SERVER_MACHINE_NAME, "your_machine_name" );
    SERV_TCP_PORT = 7019;        /* your port number */
```

```
/* when using Ndirect.o */
strcpy ( ROBOT_MACHINE_NAME, "your_robot_name" );

/* connect to robot */
connect_robot ( ROBOT_ID );

/* request position attachment */
posDataRequest ( POS_BUMPER | POS_SONAR );

/*
 * get the bumper and the sonar readings
 * the position information will be attached to these
 * values and can then be accessed below
 */
get_bp();
get_sn();

/* read the position attachment */
posBumperGet    ( &posBumper    );
posSonarRingGet ( posSonarArray );

printf("+----------------------------------------+\n");
printf("|  Position Data Attachment Test Program  |\n");
printf("+----------------------------------------+\n\n");

/* print out the position of the robot for bumper */
printf ( "Robot's last integration before last bumper hit: (%d,%d)\n\n",
         posBumper.config.configX, posBumper.config.configY );

/* print out the position of the robot for sonar */
for ( i = 0; i < SONARS; i+=2 )
    printf ( "Sonar # %2d : (%4d,%4d)        Sonar # %2d : (%4d,%4d)\n",
             i+1, posSonarArray[i  ].config.configX,
                  posSonarArray[i  ].config.configY,
             i+2, posSonarArray[i+1].config.configX,
                  posSonarArray[i+1].config.configY );

/* update the Intellisys time */
gs();

/* how old is the sensory reading? */
ageSensoryReading           = ( posTimeGet() -
                                posSonarArray[SONAR_INDEX].timeStamp
                              );
printf ( "\nSonar #%d was read %2.2f ms ago.\n",
```

```
            SONAR_INDEX, ageSensoryReading );

    /* how old was the integration at acquisition? */
    ageIntegrationAtAcquisition = ( posSonarArray[SONAR_INDEX].timeStamp,
                                    posSonarArray[SONAR_INDEX].config.timeStamp );
    printf ( "The integration was computed %2.2f ms before acquisition.\n",
            ageIntegrationAtAcquisition );

    /* how far did we move since acquisition? (assuming constant velocity) */
    distMovedSinceAcquisition     = ( ageSensoryReading / 1000.0 ) *
                                      posSonarArray[SONAR_INDEX].config.velTrans *
                                     TENTH_IN_TO_CM;
    printf ( "The robot moved %4.2f cm since acquisition of sonar #%d.\n",
            distMovedSinceAcquisition, SONAR_INDEX );

    /* disconnect from robot */
    disconnect_robot ( ROBOT_ID );
}
```

5.3 The Setup Files

The setup files are a convenient way to set a number of configuration parameters for the applications using the Nomad robot. The file `world.setup` mostly sets graphic parameters for the MAP window of the graphic interface. It also sets the communication port (Unix socket) that will be used by the client programs to connect to the server, and the default configuration file(s) for the robot(s). The `robot.setup` file sets graphic parameters for the ROBOT window, and simulation parameters. It also sets the connection parameters for the robot.

You can have only one setup file for the MAP window; its default name is `world.setup`: this name is used when you start a `Nserver` without arguments on the command line. If you do use arguments, the first one should be the name of the MAP window setup file (any name), and the following arguments should be the name(s) of ROBOT setup files. The server will automatically create as many robots as there are ROBOT setup files. There is also an option in the MAP setup file that allows you to specify ROBOT setup files as well. Any command line specification will override the setup file specification.

If you want to change any of the parameters, you simply edit the file and modify the values. Upon restart of the `Nserver`, your new values will take effect. The syntax of both files is the same: after the Copyright banner, the file is subdivided in sections with a section title in brackets:

```
[graphics]
```

In each section appear items with their values, separated by the equal sign:

```
grid_origin = +0+0
```

Comments start with a semicolumn and end at the end of the line. Line continuation (if the data is too long to fit on one line) is possible by putting a backslash at the end of each line except the last one.

5.3.1 The `world.setup` file

`[physical]`

- `size` encodes the physical size of the world for simulation and graphic representation purposes. The world is rectangular; its size is expressed in tenth of inches. World units are also tenths of inches. After the size comes the origin. An origin of (0,0) will be at the left bottom corner of the rectangle. Shifting the world by -Width/2, -Height/2 will set the origin to the middle of the rectangle, which is the default. The default value of `87360x87360-43680-43680` is a square world of 1/64th sq. mile, with origin at the center.

`[graphics]`

- `graphics` is `off` if the graphic interface (Map window, robot window) is to be disabled by default, for instance to speed up simulations. This can also be done by toggling the Graphics ON switch in the VIEW menu of the MAP window.

- `world_geom` is the pixel size and position of the MAP window, expressed in X fashion, upon creation of the graphic interface. The position is the position of the left top corner of the MAP window relatively to the left top corner of the screen. These can be modified using the window manager.

- `world_zoom` is the default zooming factor; 100% is 1 pixel per world unit (1/10th inch). 50% is 0.5 pixel per world unit. The default 12.5% is 0.125 pixels per world unit, which is 1 pixel for 8 tenths of an inch. Zooming can also be achieved using the zooming options of the VIEW menu.

- `world_center` is the world position of the center of the MAP window upon creation of the graphic interface. It is expressed in world coordinates (10th of inches) in the world coordinate system. This position can also be changed in a variety of ways: the horizontal and vertical slides of the MAP window, and all the options: ZOOM IN, ZOOM OUT, CENTER, CLIP, SLIDE, of the VIEW menu.

- `grid_origin`, `grid_incr` relate to a rectangular grid laid on the MAP window. This grid is a visual aid for the user: it provides the world dimensions, and can be used (for instance) when creating obstacles. The origin and increment size of the grid are expressed in world coordinates, relative to the world origin.

- `grid_dot_color` sets the colors for the grid dots.
- `grid_shad_color` sets the colors for the shadow dots. These shadow dots are offset by 1 pixel right and down relative to the regular grid dots. They allow the grid to be seen even in the presence of obstacles.
- `default_map` is used as a short cut if you use frequently the same map and want to load it automatically upon creation of the server. It is set to a path to the map you want to use, like `./maps/my_office.map`. The default value is `none`.

`[menu toggle]`

- `display` is set to `solid` if you want the robot to be represented as a solid circle in the MAP window, to `outline` if you want it to be represented as a hollow circle.
- `encoder` and `actual` enable the display of the encoder and real robots (see Section 3.1.2 in chapter 3). The default value is `on`. This display can be dynamically changed using the SHOW menu of the MAP window.
- `grid` enables the display of the grid; its default value is `on`.

`[connect]`

- `serv_port` is the number of the port that client programs may use to connect to the server. This number has to match the number set by the client program (variable `SERV_TCP_PORT`).

`[robots]`

- `setup_files` sets the names of the robot setup files. This is used if you usually work with several robots, and want to avoid creating them by hand each time you run an `Nserver`. Each name is separated from the next by a white space. The server will create as many robots as there are setup files. The setup files specification can be overriden by command line arguments.

  ♣ *If you want to create three identical robots, put the name of your setup file three times on this line.*

5.3.2 The `robot.setup` file

`[simulation]`

- `sim_speed` is the ratio simulated time/clock time. If set to `2.0`, the simulation will run twice as fast as the real time. The simulation can also be accelerated/decelerated using the CONTROL menu of the MAP window.

- `timeout` is the default timeout value for the simulated robot. It is equivalent to the default value given in the file `timeout.cfg` on the real robot (`/usr/local/share/robotd`). See also the command `conf_tm`.

- `speed:t_v`, `speed:s_v` and `speed:r_v` set the default translation, steering and rotation speeds for the simulated robot. They are equivalent to the default values given in the file `motor.cfg` on the real robot (`/usr/local/share/robotd`). See also the command `sp`.

`[infrared]`

- `calibration` sets the correspondance between the distance computed by the internal infrared model and the index between 0-15 that will be returned as a result of the functions getting infrared data. Distances are in tenths of inches from the sensor. The default setting assumes a linear response between 0 and 32 inches.

- `firing_order` sets which infrared sensors are active. It is a list of 16 binary values. If the n_{th} is to be active, then the n_{th} element of the list should be 1, else it should be 0. This array DOES NOT set a firing order as the one found in the `order[16]` parameter of the `conf_ir` instruction, because the simulation does not takes the firing delays into account yet. All infrared measurements are assumed instantaneous in this simplified version. By default, all infrared from 0 to 15 are active.

- `dependency` sets the history dependency of the digital filter used to even out the readings. It is a number between 0 and 100; the readings are computed as:

  $$\frac{(1-dependency)*new+dependency*history}{100}$$

  history being the average of all the readings obtained so far. This is equivalent to the setting of this value in the file `infrared.cfg` on the real robot (`/usr/local/share/robotd`). See also the `conf_ir` command.

- `halfcone` sets the simulated beam width for the infrared sensors. It is given in tenths of degrees. If the value is zero, a simple ray model of the infrared is used, in which the actual distance to the object hit by the beam is simply modified by the error coefficient described below. If the value is not zero, a cone is used to simulate the infrared beam, and the incidence angle of the beam to object surfaces is considered. Using the simple model increases the speed of the simulation.

- `incident` is an error factor used with the conical model of the beam. It is expressed as a percentage of the real value: the resulting distance is computed as $value * (1 + \Theta * incident)$, Θ being the angle of the incidence angle to the normal to the surface.

- `error` is an absolute random error factor. It is expressed as a percentage of the real value: the resulting distance is computed as $value * (1 + random[-1,1] * error)$, so if the item `error` is set to 0.2, the simulated values will be randomly distributed between 80% and 120% of the real values. In the conical model, the incident factor will be applied to the result of this computation.

`[sonar]`

- `firing_rate` and `firing_order` have the same meaning as the corresponding parameters of the function `conf_sn`: the interval between two firings in slices of 0.004 secs, and the firing order. However, since dynamic simulation of the sonars is not implemented, only the list `firing_order` is actually used to detect which are the active sonars. As with the parameter `order[]` of the function `config_sn`, a marker `255` can be used to stop the list.
- `dist_min` and `dist_max` set the range of the simulated sonars in tenths of inches.
- `halfcone` is half the angular range of the main lobe of the sonar, in tenths of degrees. If this value is set to 0, a simple ray model of the sonar is used, in which the actual distance to the object hit by the beam is simply modified by the error coefficient described below. If the value is not zero, a cone is used to simulate the sound beam, and the incidence angle of the beam to surfaces is considered. Using the simple model increases the speed of the simulation.
- `critical` is the maximum angular difference in tenths of degree between the sonar axis and the normal to the surface for the sensor to return a value. If the difference is larger, the simulated sonar is assumed not to have got any echo back, and the maximum distance will be returned.
- `overlap` sets the minimal apparent size of a surface to be detected when using the conical model. It is expressed as a ratio ($0.5 = 50\%$) to the angular range of the main lobe. If the angular sector connecting the two endpoints of a segment within the sound cone is larger than $overlap * (2 * halfcone)$, then this segment is considered visible.
- `error` is an absolute random error factor. It is expressed as a percentage of the real value: the resulting distance is computed as $value * (1 + random[-1,1] * error)$, so if the item `error` is set to 0.2, the simulated values will be randomly distributed between 80% and 120% of the geometric values.

`[laser]`

- `dist_min` and `dist_max` set the range of the simulated laser in tenths of inches.

- `halfcone` is half the angular range of the laser beam in tenths of degree.
- `critical` is the maximum angular difference in tenths of degree between the laser axis and the normal to the surface for the sensor to return a value. If the difference is larger, the surface will not generate any data.
- `mode` sets the laser in `point` or `line` mode. However, since the optics of the camera are not simulated, only the distance is available in point mode. See also the `conf_ls()` command in the Language Reference Manual.
- `num_points` sets the number of points returned in point mode. See also the `conf_ls()` command in the Language Reference Manual.
- `error` is an absolute random error factor. It is expressed as a percentage of the real value: the resulting distance is computed as $value * (1 + random[-1, 1] * error)$, so if the item `error` is set to 0.2, the simulated values will be randomly distributed between 80% and 120% of the geometric values.
- `calibration` and `offset` set the coefficients of the conversion pixel→distance. They are obtained as a result of the calibration of the laser using the `calibration` utility provided with the Laser Range Finder.

  These numbers should be the same as the coefficients in the file `/usr/local/share/robotd/laser.cal` on the robot. Note the backslash as a continuation between two lines.

  ♣ *These values are NOT simulation values: they are used by the laser programs to convert laser pixel data to range data. If the contents of the file* `laser.cal` *on the robot are differents from this item, you will experience anomalies when switching from point mode/distance to line mode, since the computations for line mode are done on-board (using* `laser.cal`*) whereas the computations for point mode are done in the server (using the values of* `robot.setup`*).*

`[other]`

- `bumper_mode` is `on` if the simulated robot has bumpers, `off` if not.
- `compass_mode` represents the state of the compass when starting the simulated robot.

`[mask]`

The real robot sends its sensor mask upon connection to the server. This mask is in the robot file `/usr/local/share/robotd/mask.cfg`. For the simulator, the smask values are defined here (as they would be set by the simulated robot). Each item contains a list of binary values determining whether the corresponding data should be updated. See also the Section 4.2.2 in chapter 4.

[motion]

- bounce sets the distance that the simulated robot should go back after hitting a wall. The value is the time in seconds during which the bouncing motion occurs. The distance bounced is the current speed multiplied by this value. Note that a simplified control model is used: the wheels are supposed to be still spinning forward during the bouncing motion, so that the encoder robot will move on (and will probably cross the obstacle) while the actual robot will be pushed off the obstacle.

- pos_*_bias, zero_*_bias and neg_*_bias set the error model for the motion of the three axes: Translation, Steering, Rotation. They are percentage errors on robot velocities. For all three axes, the same model is used:

  - systematic bias: zero_*_bias
  - directional bias: pos_*_bias and neg_*_bias
  - directional random error interval: pos_*_int and neg_*_int;

  Directional bias means a bias related to a particular direction of motion, positive or negative. If commanded_* is the commanded value (translation, steering or rotation), then the simulated value will be:

```
if (commanded_* > 0)

simulated_*=commanded_*(1+zero_*_bias+pos_*_bias)(1+random[-1,1]*pos_*_int)

else

simulated_*=commanded_*(1+zero_*_bias+neg_*_bias)(1+random[-1,1]*neg_*_int)
```

  The provision for a difference in negative and positive is to take into account gravity when the robot has to move on slanted surfaces. It is actually something that has been implemented for future releases when the client program will be able to set simulation parameters. For now, the difference can be ignored.

[graphics]

- wind_name is the name of the ROBOT window for the associated robot.

- robot_geom is the pixel size and position of the ROBOT window, expressed in X fashion, upon creation of the graphic interface. The position is the position of the left top corner of the ROBOT window relatively to the left top corner of the screen. It can be modified using the window manager.

- `robot_zoom` is the default zooming factor; 100% is 1 pixel per world unit (1/10th inch). 50% is 0.5 pixel per world unit. The default 12.5% is 0.125 pixels per world unit, which is 1 pixel for 8 tenths of an inch. Zooming can also be achieved using the zooming options of the VIEW menu.
- `robot_center` is the world position of the center of the ROBOT window upon creation of the graphic interface. It is expressed in world coordinates (1/10th of inches) in the world coordinate system. This position can also be changed in a variety of ways: the horizontal and vertical slides of the ROBOT window, and all the options: ZOOM IN, ZOOM OUT, CENTER, CLIP, SLIDE, of the VIEW menu.
- `ssen_geom`, `lsen_geom`, `joy_geom` set the pixel size and position of the SHORT SENSORS, LONG SENSORS and JOYSTICK windows respectively. These can be modified using the window manager.
- `com_coor` and `rec_coor` set the pixel position of the COMMAND LINE window and the RECORDER window. These can be modified using the window manager.
- `*_color` set the colors of the different graphic entities displayed in the ROBOT, SHORT SENSORS and LONG SENSORS windows, including actual and encoder robot.

`[menu toggle]`

- `bumper`, `infrared_ray`, ..., `long_local` set the default values of the toggle switches that govern graphic displays in the SHORT SENSORS and LONG SENSORS windows. See Chapter 3 for a description.
- `robwin_bumper`, ..., `robwin_laser` set the default values of the toggle switches that govern graphic display of sensor data in the ROBOT window. See also the SHOW menu for this window.
- `display` is set to `solid` if you want the robot to be represented as a solid circle in the ROBOT window, to `outline` if you want it to be represented as a hollow circle.
- `trace` is set to `outline` or `solid` if you want the robot to show its path as it moves, to `none` else. See also the SHOW menu for this window.
- `encoder` and `actual` enable the display of the encoder and real robots (see Section 3.1.2 in chapter 3). The default value is `on`. This display can be dynamically changed using the SHOW menu of the ROBOT window.

`[recorder]`

- `infraredp`, .., `commandp` set the default values of the item that will be recorded when starting the recorder. See also Section 3.3.5.
- `tape_len` sets the maximum length (in recordings) of a recorder “tape”.

`[connect]`

- `conn_type` select between `serial` and `tcpip`(ethernet) communication.

- `machine` is the machine name of the robot. It is the name you use to `login` or `telnet` to the robot. The robot comes with its name set as `nomad`.

- `tcpip_port` is the port used by the server to communicate with the robot via ethernet.

- `serial_port` and `serial_baud` configure the serial connection (usually radio modem)

- `retrans` sets the repeat of every message that generates a time out. It should be used only with serial port connection.

- `normal_timeout` is the timeout for all commands that are supposed to return immediately (`zr` and `ws` are not: they have their own internal timeout).

- `special_timeout` is the timeout for user packets only (See section 5.4).

5.4 User Packets

In addition to the regular information going to the robot (mostly commands) and coming from the robot (mostly sensors data), it is possible to define custom messages that will be interpreted by a user program on the robot. Note that this feature is not available in simulation since it involves a recompilation of the robot daemon.

To build these "special requests", you need to:

- on the client side, write a client program that builds, sends and receives user packets. The main function to do so is the function `special_request` that sends a packet to the robot and receives an answer.

- on the robot side, write a `user_action` routine that will be called by the main event loop upon reception of a user packet. This function should process the packet and send the answer back.

5.4.1 Client Program

To create a communication program, you have to create the relevant variables, then write a "send" section and a "receive" version.

Initialization

You need to allocate memory for a send buffer and a receive buffer. The usual way is to declare two arrays of `unsigned char`. The maximum allowed size is `USER_BUFFER_LENGTH` declared in `Nclient.h`. This maximum size is currently 65535 bytes (`0xFFFF`), but due to protocol bytes, the number of effective bytes is 65526. Although the user shouldn't have to worry about the internal format of the packets, here it is:

Name	Description	Size
begin_pkg	always 1	1 byte
length	bytes between begin_pkg and end_pkg	2 bytes
type	ID of the packet	1 byte
data	the data to be transmitted	n bytes
chksum	sum of previous bytes mod 256	1 byte
end_pkg	always 92	1 byte

Table 5.1: The format of the data packages.

In the case of user packets, the data portion contains a 2-bytes length, a 1-byte serial number and a 1-byte type [3].
You also have to declare two variables as `short`: an index to the buffer, and a length variable. The index is used by all the communication functions for internal management, you shouldn't modify its value by yourself.

Sending

To send user packets, you need to:

- Initialize the send buffer with `init_send_buffer`: this fills the buffer with the protocol bytes and positions the index.
- Fill the send buffer with data using the `stuff*` functions. These functions fill the send buffer according to the type of their argument (`char`, `double`, etc.) and increment the index accordingly.
- Complete the send buffer header with `stuff_send_buffer_header`. This function will complete the header by computing a checksum. Note among its arguments the `serial_number` and the `packet_type`: these are not used internally, but are provided for your own management of the communication.
- Send the user packet with `special_request`. This function will wait for an answer from the robot program. The timeout is set in the `robot.setup` file, section `[connect]`, item `special_timeout`.

[3] Why do all the protocol bytes add up to 10, when the difference between maximum size and effective size is only 9 ? Because the end_pkg byte is not stored internally.

Receiving

To process data from the robot, you need to:

- Initialize the receive buffer with `init_receive_buffer`, and extract the header with `extract_receive_buffer_header`.
- Extract the data using the `extract*` functions. These functions extract data from the receive buffer according to the type of their argument, and increment the index accordingly.

A Simple Example

Here is an example of a program sending and receiving long integers; it assumes that the default `user_action` routine is running on the robot side (see below).

```
#include <stdio.h>
#include "Nclient.h"

#define UINT4_TYPE       4

int main( int argc, char **argv )
{
  short i, b_index, b_length;
  unsigned char serial_number;
  unsigned char packet_type;
  unsigned char user_send_buffer[USER_BUFFER_LENGTH];
  unsigned char user_receive_buffer[USER_BUFFER_LENGTH];

  /* Establish connection */

  SERV_TCP_PORT = 7019;                     /* Matches world.setup         */
  strcpy( SERVER_MACHINE_NAME, "zulu"     ); /* machine for server(Nclient) */

  connect_robot(1);

  /* init buffer */
  init_send_buffer( &b_index );

  /* fill buffer with a long integer */
  stuff4byteuint( 0x01010101, user_send_buffer, &b_index );

  /* put the header into the buffer, has to match the data */
  stuff_send_buffer_header( b_index, 0, UINT4_TYPE, user_send_buffer );

  /* send the buffer */
```

```
  printf( "Sending   : 0x10101010.\n" );
  special_request( user_send_buffer, user_receive_buffer );

  /* process received_buffer */
  init_receive_buffer( &b_index );
  extract_receive_buffer_header( &b_length, &serial_number, &packet_type,
 user_receive_buffer);
  printf ( "Receiving : %lx\n",
  extract4byteuint( user_receive_buffer, &b_index ) );

  /* closing connection */
  disconnect_robot (1);
  return ( 1 );
}
```

This program is a simplified version of a detailed example given in the Language Reference Manual. This detailed example also shows how to get raw infrared data (intensity data), and how to access the serial ports of the mother board. See below how to access the auxiliary port on the Intellisys100 board.

5.4.2 `user_action` program

The robot daemon automatically calls a function `user_action` upon reception of a user packet. This program is in `user.c`, and is compiled and linked with the other components of the robot daemon when calling `make`. There is first a selection on the serial number number, so that different users can connect to the same robot and share information without conflict in processing. The user action is then selected on the packet type variable. The simplest way to add your own action is to add your own index in the `switch` instruction.

For user's convenience, a number of standard actions have been implemented under serial number 0. They are (according to the packet type):

- Type 0 to 5: takes inputs of different types (characters, signed ints, etc.) and returns them multiplied by two. This is essentially a communication test feature.
- Type 10: accesses the raw infrared data (energy levels) for all the infrared sensors, returns it.
- Type 11: reads and bitwise ORs the auxiliary port (port 1B on the 6811 board) 100 times. This is used by a prototype beaconing system from Nomadic Technologies (information on request).
- Type 20: initializes the serial ports of the mother board (the port number, from 1 to 4, is given as a short integer). Returns a char equal to 1 if success, 0 if failure.

- Type 21: puts/gets a string on a serial port.
- Type 30: gets the auxiliary port data direction register (port 1B on the 6811 board). The configuration is read as an 8-bit unsigned char. A 1 means that the corresponding pin is configured as an output, a 0 an input.
- Type 31: sets the auxiliary port data direction register (port 1B on the 6811 board). The configuration is read as an 8-bit unsigned char. A 1 means that the corresponding pin is configured as an output, a 0 an input.
- Type 32: reads the value on the auxiliary port (port 1B on the 6811 board) as an 8-bits unsigned char, sends it.
- Type 33: writes the given value on the auxiliary port (port 1B on the 6811 board). Only the pins configured in "output" will be affected.

♣ *Every user action should return an answer !! The* `special_request` *function waits for it .*

5.5 Accessing the auxiliary port on the Intellisys100

The auxiliary port is the port labelled "1B" on the 6811 (Intellisys100) board. This port has 20 pins, but only 16 are actually available for communication. The pins are assigned as illustrated Figure 5.10. The odd-numbered pins are all connected to the ground, while the even-numbered 2-16 pins are used for signals.

1	3	5	7	9	11	13	15	17	19	Ground
2	4	6	8	10	12	14	16	18	20	Signal (+5V)

Figure 5.10: The auxiliary port pins configuration

The port 1B can be configured and accessed from a client program using user packets. The client program sends (unsigned) characters whose 8 bits are interpreted as the 8 even numbered pins of the port: the least significant bit in the character corresponds to pin 2, the next one to pin 4, etc. Each pin can be configured as "input" (0), or "output" (1) by sending a character in a user packet of type 31. For instance, sending `0x00` will set all the pins 2-16 as input.

To access the port, you first have to connect to the real robot, in direct or server mode. Then each time you send a configuration character, or a read/write character, you have to

- initialize the send buffer
- stuff the send buffer with the character you want to send, using `stuffuchar`,

- initialize the buffer header, with the correct packet type (30-33),
- send your request
- initialize your receive buffer,
- extract the receive buffer header,
- extract the character using `extractuchar`

This is exactly identical to the simple example given above, the only difference being the packet type and the data sent (character). For instance, first send the character `0xFF` with packet type 31 to set the pins 2-16 as output. Then, sending `0x01` with packet type 33 will set the pin 2 high, `0x0C` will set the pins 4 and 8 high. Writing will affect only the pins configured as output. Note that setting (type 31) and writing (type 33) will echo back the character sent to the port.

Drive characteristics This is a CMOS part having limited drive capability. It should only be used to drive TTL parts (preferably CMOS). Any other device must be buffered. Configured as output, it will source or sink to 2.3mA. Configured as input, typical high input current is 400μA, maximum low input current is 2.6mA.

Appendix A

BUG REPORT FORM

Thank you for reporting bugs, malfunctions andsuggested improvements regarding this release of Nomadic's software. Please E-mail the following information to `bugs@robots.com`. This information will help us to figure out your problem faster.

A.1 Information About Yourself

- Your Name:
- Your Organization:
- Your E-mail address:
- Your robot(s) serial number(s):
- The date of this report:

A.2 Information About Your Environment

- Your machine type (ex Sparc10):
- Your Operating System (ex SunOS):
- The program and version you are using (ex Nserver version 2.6.1):

A.3 Description Of Your Problem

Include the answer to the following questions:

- What were you doing when the problem occurred ?
- Is the problem reproducible ?
- Does the problem occur on the robot and/or the simulator ?

A.4 Additional Information

- Copy of the world.setup and robot.setup files
- Core file, if there is one.
- Smallest sample of code that reproduces the problem

You can also mail this form to:

Nomadic Technologies
Software Department
2133 Leghorn St
Mountain View, CA 94043
U.S.A.