NOMAD XRDEV SOFTWARE MANUAL

RELEASE 1.0

Nomadic Technologies, Inc.
2133 Leghorn Street
Mountain View, CA 94043-1605
TEL 650.988.7200
FAX 650.988.7201
Email: nomad@robots.com

Rev. Date 3/99

WHERE CAN I GET HELP?

1. By email: send a description of your problem, if possible with the source program, to: support@robots.com.
2. By phone: call +1.650.988.7200 and ask for Technical Support.

WHERE CAN I GET SOFTWARE?

For the convenience of timely software distribution, Nomadic has set up a web site for this and future Nomadic software releases. From this FTP site, you can download the most up-to-date software distributed by Nomadic. This includes the Nomadic Host Development Environment, the Nomadic Robot Control Software, and documentation.

To download software from this FTP site, simply go to: http://www.robots.com (205.162.4.11) and click on the Downloads link. Please read the software license agreement and click on the "I agree! Take me to the downloads page!" link.

When prompted for a user name and password, type:

Name: robots Password: N0mad1C

The 'o' is a ZERO and the 'i' is a ONE, the letters 'N' and 'C' are CAPITALIZED. Since the software is intended to be used by Nomad users only, please keep the FTP site information confidential.

Once you are logged in, you will come to a page that lets you select between Host Development Environment, Robot Control, and Manuals. Select the appropriate link and download the software appropriate for your operating system.

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

To order additional copies of this manual or other manuals, please call +1.650.988.7200 and ask for the Sales Department.

DISCLAIMER AND WARRANTY INFORMATION

Thank you for purchasing a Nomadic Technologies, Inc. product. The Nomad XR4000™ is warranted to the original purchaser (Customer), to be free from defects in materials and workmanship for a period of two years for mechanical components and one year for electrical components. The warranty is effective 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 data. The Customer's 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 +1.650.988.7200 for instructions.

In the event that service is required, after notifying Nomadic Technologies and receiving an 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:Ngui –h myhost –s 65000
```

2 ☑ is a remark, note, or tip, as in:

☑ *You can have up to 100 robots controlled simultaneously from the same GUI.*

3  denotes trouble shooting information, as in:

 **The robot does not move.**

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

CONTENTS

CHAPTER 1: GETTING STARTED

OVERVIEW

Nomadic has developed a new robot software architecture known as the Xtreme Robotics Development Environment or XRDev.

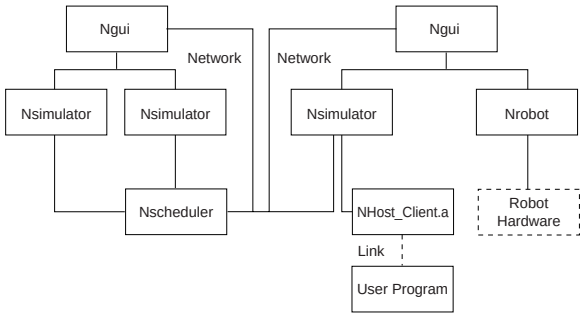


Figure 1. Software Architecture

The new software consists of the following libraries and executables:

1. `Nrobot` - This is the robot program that runs on the robot itself and listens to commands. Upon receiving these commands, it directs the robot hardware accordingly. This program is also extensible, so that the user may add new behaviors to the robot without requiring the source code.
2. `Nhost_client.a` - This is the library of functions used by a client to direct the robot from any machine on the network. The user merely compiles his program, links with this library, and then runs the resulting executable program. The library itself deals with all of the underlying network communication occurring while the user's program is talking to one or more `Nrobot` processes. This library is smart enough to use optimized communication when the client is running on the same machine as Nrobot.
3. `Ngui` - This is the graphical user interface, which provides a graphical representation of robot activity/sensor-readings while allowing users to direct robots manually.

RELEASE 1.0 Contents

This release contains:

- A README file with release notes, and an INSTALL file with installation instructions, both located in the top level directory.
- The graphical user interface program `Ngui`, located in the `bin/` directory.
- One library file, `Nhost_client.a`, located in the `lib/` directory.
- One header file, `Nclient.h`, located in the `include/` directory.
- Several example programs located in the `examples`/ directory.
- This documentation file located in the `doc/` directory.

In addition, the robot distribution (`xrdev-xr4000-1.0.0-i386-unknown-linux`) contains additional files required for the robot server functionality:

- The server program, `Nrobot`, located in the `sbin/` directory.
- Several utility programs, located in the `bin/` directory.
- The motor driver module, `xrm.o`, located in the `modules/` directory.
- Embedded code for the various distributed nodes on the robot, located in the `embedded/` directory.
- Additional documentation and notes in the `doc/` directory.

This release requires:

- A Nomad XR4000 robot running under Linux release 2.0.0 or higher.

Or,

- A UNIX workstation running under Linux release 2.0.0 or higher.

A Simple Example

Here is how to compile and run the `swerve.c` example provided with this release. You can also start the robot and control it from the Graphic User Interface

(GUI) without any user program. Please refer to *"Chapter 4: The Graphic User Interface"* for information on the GUI.

In this scenario, the program `swerve.c` is compiled and linked to `Nhost_client.a` and runs on a workstation while the robot is running `Nrobot`.

1 Verify that `Nrobot` is running. If not running, see the `INSTALL` in this distribution.

2 Choose a machine to run your client program on. This can be any machine connected to the network, including the robot itself.

3 Login to the machine and change to the proper examples directory, such as `/xrdev-1.0.0-i386-unknown-linux/examples/`

5 Compile `swerve` on the client machine:

```
make swerve
```

6 Run `swerve`, giving it the hostname of your robot with the `--host` command-line option. For example, to control a robot with hostname "fatboy", you would type

```
swerve --host fatboy
```

Also, if you started `Nrobot` with a robot ID other than 1, you will need to give the robot ID with the `--robot_id` command-line option:

```
swerve --host fatboy --robot_id 2
```

CHAPTER 2: THE XRDEV ARCHITECTURE

DESIGN FEATURES

XRDev is a multi-process architecture. An XRDev application is made of several processes: robot processes, user processes, interface processes, etc., communicating through the network. There is no limit (but efficiency and network load) to the number of robots that can be controlled under XRDev and no limit to the number of user programs that control them. There is also no restriction on the number of robots that can be controlled by one user program and no restriction on the shared control of one robot by several user programs. Finally, user programs can be executed anywhere on the network.

Processes and Configurations

There are three types of processes:

1. An `Nrobot` process drives a real robot. There can be any number of `Nrobot` processes in the application, but only one `Nrobot` process running on each real robot.
2. An `Ngui` process sets a graphic interface. From this interface, a user can send commands to and retrieve data from any Robot Process. There can be any number of `Ngui` processes, connected to any number of Robot Processes.
3. A user process embeds the user program that directs robot processes.

There are a number of possible configurations using various combinations of these processes.

The simplest configuration is `Nrobot` running on the robot and `Ngui` running on another machine, communicating over the radio Ethernet. The user manually controls the robot from `Ngui`

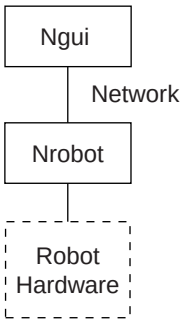


Figure 2. Simple Configuration - One GUI, one robot

A more elaborate configuration involves an `Nrobot` process and a user client program linked with the Host Client Library:

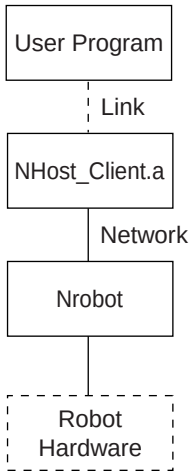


Figure 3. Simple Network Configuration
One robot, One User Program, running across a network.

In this configuration, the User Program linked to the Host Client Library can be compiled and executed on any machine with a (wired or radio) connection to the robot. The `Nrobot` process runs on the robot.

CHAPTER 3: NROBOT

INTRODUCTION

Nrobot is the server that communicates with the robot hardware by sending commands and receiving sensor data. All user programs interact with the robot through this process.

Nrobot is usually run from the system startup scripts. The installation process installs a file in `/etc/rc.d/init.d` called `robot`. This is a shell script which handles initialization and shutdown of robot hardware devices, and starting and stopping of the Nrobot process.

The default setup for Nrobot should be fine for most installations, and you will be able to use the programming API simply by specifying the hostname of your robot. However, if any of the following are true for you, you may need to change your Nrobot configuration:

- You have additional hardware devices, such as a SICK laser or a Nomad XRLift mechanism.
- You have more than one XR4000 at your site, in which case one of your robots will need to have a robot ID number other than the default of 1.
- You wish to change the default state to which Nrobot initializes your sensors (currently, Nrobot only initializes sonars).

This chapter will describe how to change the configuration of Nrobot via setup file (for permanent configuration), and via the command line (for one-time only testing). It also presents a few troubleshooting tips, should you run into problems.

XRDEV CONFIGURATION OPTIONS

As you will learn in *Chapter 5: The Robot Language*, clients must go through a two-step process in order to establish a data connection to a given robot. The first step establishes a TCP/IP connection to a *scheduler*, which is a process on the network that keeps a table mapping robot ID numbers to socket connection information. The client then requests the socket connection information for the ID number of the robot it wishes to connect to, and the scheduler provides it.

When Nrobot is started, it establishes a TCP/IP listener socket for clients to connect to. It also connects to a scheduler and informs it of it's robot ID, as well as its host name and the port number of its listener socket. From this point on, any clients connecting to the scheduler are informed of this robot.

In the default configuration, *Nrobot acts as its own scheduler.* That is, clients connect directly to Nrobot and are informed of one robot entry, which resides on the connection already established.

However, if you have more than one robot, you may wish for one of them to act as a scheduler for both robots. In this case, robot number 1 must be configured to connect to robot number 2 (which is acting as scheduler) and register its robot ID (1). Then, a client wishing to connect to robot number 1 would ask the scheduler (robot number 2) where on the network robot number 1 resides. Once it had this information, it then establishes a second connection to robot number 1. If desired, this client could also establish a connection to robot number 2, since robot number two is scheduling itself as well as robot number 1. This allows a single client program to control two robots simultaneously.

Nrobot accepts the following configuration options:

Robot ID: the number by which client programs refer to this robot. This can be any integer number greater than zero. The default is 1.

Listener port: the TCP port number on which Nrobot listens for data connections (as well as scheduler connections, if it is scheduling itself). The default is 7073. Any port number between 1024 and 65535 may be used.

Scheduler location: a hostname and TCP/IP port number which describe the location on the network where Nrobot should register its robot ID. By default Nrobot schedules itself.

Command line options

To configure these parameters for one-time testing, Nrobot provides the following command-line options:.

Option	*Description*
`-help`	Prints out a usage message
`-robotid n`	The ID number to register as.
`-port n`	The TCP port on which Nrobot will listen for connections
`-schedulerhost name`	The hostname of the scheduler
`-schedulerport n`	The TCP port on which the scheduler is listening
`-f filename`	The name of an alternative setup file
`-d`	Do not fork to run in the background

Setup file options

The above parameters can also be specified on a permanent basis by adding them to Nrobot's configuration file. By default, this file is `/usr/local/xrdev/etc/nrobot.cfg`; this can be overridden using the `-f` command-line option.

The configuration file consists of sections and key/value pairs. Sections are denoted by surrounding braces. Keywords and their values are always separated by an equals (=) sign. In the following example, the `[baz]` section has one entry which sets the keyword `foo` equal to "`bar`":

```
[baz]
foo = bar
```

Nrobot's robot ID and data port may be specified in the `[robot]` section of its configuration file. To specify the robot ID, edit the value associated with the keyword `robot_id`. To specify the port, the keyword is `listener_port`. For example, the following section specifies a robot ID of 2 and a listener port of 1001:

```
[robot]
robot_id = 2
listener_port = 1001
```

Similarly, the scheduler that Nrobot registers with can be specified in the `[scheduler]` section, via the `hostname` and `listener_port` keywords. The following example configures Nrobot to register with a scheduler located at robot2.my_lab.edu on port 7073:

```
[scheduler]
hostname = robot2.my_lab.edu
listener_port = 7073
```

Keep in mind that command-line options always override setup file parameters.

DEFAULT SENSOR STATES

When Nrobot initializes, it looks in its setup file for default sonar firing orders; if it finds these, it will send a corresponding configuration command to the appropriate door. Each sonar set has its own section of the form [sonsetN] where N is replaced by the sonar set number (0-5; see *Chapter 5: The Robot Language* for a description of these indices). For example, the section `[sonset1]` configures the bottom sonar set on the front door. To configure the firing orders, add a line to the desired section of the form:

```
firing_order = fo0 fo1 fo2 ... foN
```

Each `f0N` value is a sonar index from 0 to 7. The special value 255 indicates termination of the firing order. For example, to configure the bottom sonar set on the left door to alternately fire the first and eight sonar, use the following:

```
[sonset4]
firing_order = 0 7 255
```

To configure the top sonar set on the front door to be completely off, use

```
[sonset0]
firing_order = 255
```

By default, each door initializes its sonar firing order to all on.

ADDING NEW HARDWARE

If you have purchased a Nomad XRlift mechanism or one or more SICK lasers with your robot, Nrobot must be told that they are present so that they will be initialized upon startup. The setup file `[lift]` section is used to configure your lift mechanism, and the `[s550]` section is used to configure SICK lasers.

There are two keywords in the `[lift]` section. The first is `type`, and this should always be set to `xrlift`. The second, `config_path`, should be set to the pathname of the configuration file generated by the utility program `xrlconfig`. This program and the calibration process are described in *Chapter 6: XR4000 Lift Mechanism*. The following is a typical XRlift description:

```
[lift]
type = xrlift
config_path = /usr/local/xrdev/etc/
xrlift.cfg
```

SICK lasers are described in the `[s550]` section. The first keyword, `s550s`, gives a list of all connected Sensus 550s. The entries in this list will be taken as keywords for following lines which each describe an individual sensor. A SICK description line takes the form

```
keyword = CreateS550 name pointcount x
y cosine sine reference port
```

`CreateS550`	Keyword, do not change
`name`	An arbitrary name assigned to this sensor
`pointcount`	The maximum number of points supported by this sensor; usually 361
`x, y`	The X, Y location of the center *of the scan* relative to the base object
`cosine, sine`	The sine and cosine of the sensors heading *relative to the base object*
`reference_object`	The name of the object this sensor is positioned relative to; use `base`
`port`	The serial port to which this sensor is attached.

For example, the following section describes two SICK lasers attached to the robot. The first SICK is mounted in the standard position, facing out the provided slot in the front door, and is connected to the first serial port. The second SICK is mounted on top of the robot, and the center of its scanning is coincident with the center of the XR4000. It is facing backwards and is connected to the second serial port:

```
[s550]
s550s = s550a s550b
s550a = CreateS550 s550a 361 0 150 0 1
base /dev/ttyS0
s550b = CreateS550 s550b 361 0 0 0 -1
base /dev/ttyS1
```

The positions described by these configurations are relayed to the GUI so that it can display the laser data appropriately.

TROUBLESHOOTING

Only one process may be accessing the robot's Arcnet system at a given time. This means that there may be only one instance of the Nrobot process running in the system. To prevent multiple instances, Nrobot uses a lockfile mechanism in the directory `/tmp/.nrobot/`. Nrobot creates a lockfile in this directory when it initializes. This file is deleted when Nrobot is killed. If, upon initialization, Nrobot gives an error message claiming that the lockfile already exists, this probably means that the system was not shut down cleanly. To remedy the situation, first verify that there is indeed no other Nrobot process currently running (you can use the `ps` command to check this); if Nrobot is not running, you may safely delete the lockfile:

```
# rm /tmp/.nrobot/*.lock
```

Then, you can restart Nrobot manually:

```
# /usr/local/xrdev/sbin/Nrobot
```

It may be wise at this point to investigate why the system was not shut down cleanly!

CHAPTER 4: THE GRAPHIC USER INTERFACE

INTRODUCTION

Ngui is the primary graphical interface for the control and monitoring of the Nomad XR4000.

Ngui allows the user to:

- Control a robot using the software joystick
- Monitor the sensor readings of a robot instantaneously and accumulated over time

Ngui consists of the two main windows:

- The **World Window** on which all sensor data and robots are displayed
- The **Robot Window** that corresponds to each robot that Ngui is connected to.

In addition, there are several panels used to control both interface and control settings.

Getting Started

Run Ngui by name:

```
Ngui
```

Upon invocation, you will see an empty World Window as shown in Figure 4. There is only one World Window per Ngui process, but there are as many Robot Windows as there are connected robots

Command Line Options

The behavior of Ngui can be modified by command line options listed below.

Key	Purpose	Value	Default
-h	Scheduler Host	Machine name	localhost
-s	Scheduler Port	Integer > 1024	65000

For example, to run Ngui and tell it to automatically connect to the robot "fatboy", invoke:

```
Ngui -h fatboy -s 7073
```

where 7073 is the default Nrobot socket number.

World Window

The **World** Window is the primary window of Ngui and is always present, even in the absence of connections to either robots or simulators.

The **World** Window contains three areas:

- **World Drawing Area**: within this area are the time-accumulated sensor readings and a icon representation of the robot and its orientation.
- **World Window Status Area**: located at the bottom of the world window, this area consists of two lines of text: the window bounds description and the information line. The window bounds description always contains a copy of the current window bounds (in world coordinates). The information line is used by Ngui to display various messages about ongoing operation
- **World Window menu bar**: located at the top of the window, this is where the pulldown menus are located. The

functionality of the menu items is described in the next section.

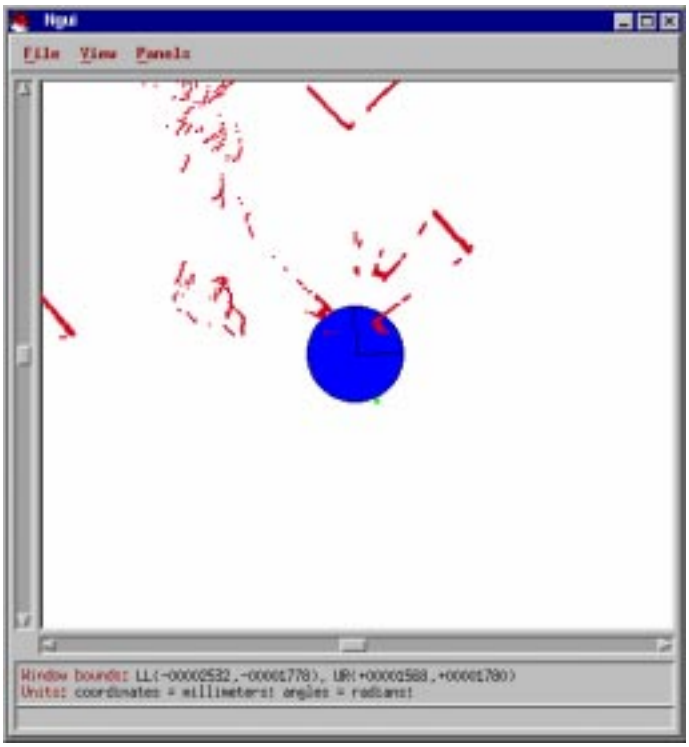


*Figure 4. The **World** Window*

World Menu Bar

File Menu

- **Exit**: Exit Ngui application. All windows will be closed.

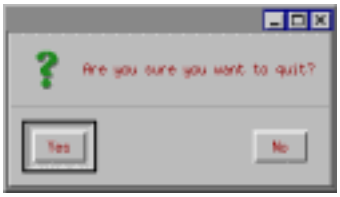


*Figure 5. The **Exit** Panel*

View Menu

- **Zoom**: Allows the user to zoom in by pressing the left button and zoom out by pressing the right button.
- **Unzoom**: Same as zoom, with buttons reversed
- **Clip**: Allows the user to resize the viewing area by dragging with the left button to delimit new bounds.
- **Center**: Allows the user to center the screen on a left-clicked point
- **Slide**: Allows the user to pan the view by left dragging in the drawing area

Panel Menu

- **Display**:The display panel allows control over what is displayed in the world window's drawing area. For each robot or simulator, the display panel provides a button for each type of data that may be shown. Click in the

boxes you wish to display, and click OK to apply the changes.

*Figure 6. The **Display** Panel*

- **Refresh**: The refresh panel allows the user to “refresh” (i.e. erase old content to make room for new content) various types of data. For each robot or simulator, the display panel provides a button for each type of data that may be refreshed. Click in the boxes you wish to refresh, and click OK to apply the changes.

*Figure 7. The **Refresh** Panel*

- **Connect**: If Ngui is not connected to a scheduler, then the connect panel will allow the user to specify the hostname and port number of a scheduler to connect with. Type in the name of the host and port number, then click OK to continue.

*Figure 8. The **Connect** Panel*

If Ngui is currently connected to a scheduler, then the connect panel will present the user with a list of the currently registered processes, which the user may then connect to. Click on the box next to the status “Connected” box and

click OK to connect as shown in Figure 9.

*Figure 9. The **Connect** Panel*

- **Units**: The units panel allows the user to change the units used by `Ngui`.

*Figure 10. The **Units** Panel*

Robot Window

This window gives the user full access to the entire state of the robot, and provides an interface for control. The Robot window is separated into the following subwindows:

- **Short Range Sensor**: contains instantaneous readings of the tactile and infrared sensors.
- **Long Range Sensor**: contains instantaneous readings of the laser and sonars sensors.
- **Joystick**: contains a software joystick that allows the robot to be moved by using a mouse.
- **Robot Information**: contains instantaneous information about the Integrated Configurations (axis positions), axis speeds, axis accelerations, compass and battery values.

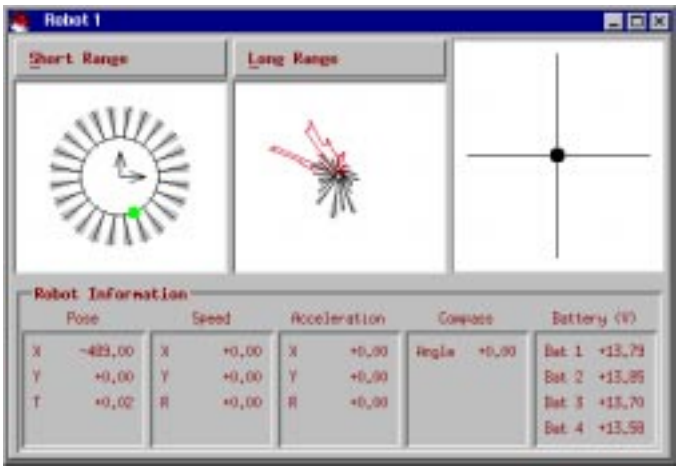


*Figure 11. The **Robot** Window*

Short Range Sensor

The Short Range Senor display is located in the top left-hand corner of the Robot Window. It shows a graphical depiction of the robot's short-range sensors. These sensors currently include the robot's infrared and tactile sensors. Infrared sensors are displayed by a series of cones and/or rays, while tactile sensors appear as small colored squares. Green squares represent soft bumper hits and red squares represent hard bumper hits. Also displayed is an oriented circle representing the robot's base.

The Short Sensor Options menu is used to control the sensor display in the Short Sensor display window. It offers the following options:

- **Show Infrared Rays**: toggles whether or not the infrared sensors will be depicted as central rays.
- **Show Infrared Cones**: toggles whether or not the infrared sensors will be depicted as oriented cones.
- **Global/Local View**: in global mode, the robot's orientation will reflect its orientation in global world coordinates, and all sensor data will also be shown globally. In local mode, the robot's orientation will remain fixed, and sensor data will be displayed in robot-relative coordinates.
- **IR Set Display**: a series of toggle buttons to control the display of the robot's different sets of infrared sensors.
- **Bumper Set Display**: a series of toggle buttons to control the display of the robot's different sets of bumpers.

Long Range Sensor

The Long Range Sensor display is located in the top middle of the Robot Window. It shows a graphical depiction of the robot's long-range sensors. These sensors currently include the robot's sonar and lasers. Sonar sensors are displayed by a series of cones and/or rays, while lasers appear as a set of connected red lines. Also displayed is an oriented circle representing the robot's base.

The Long Sensor Options menu is used to control the sensor display in the Long Sensor display window. It offers the following options:

- **Show Sonar Rays**: toggles whether or not the sonar sensors will be depicted as central rays.
- **Show Sonar Cones**: toggles whether or not the sonar sensors will be depicted as oriented cones.
- **Global/Local View**: in global mode, the robot's orientation will reflect its orientation in global world coordinates, and all sensor data will also be shown globally. In local mode, the robot's orientation will remain fixed, and sensor data will be displayed in robot-relative coordinates.
- **Sonar Set Display**: a series of toggle buttons to control the display of the robot's different sets of sonar sensors.
- **Laser Display**: a toggle button to display of the robot's laser sensor data.

Joystick

The Software Joystick, located in the upper-right corner of the Robot window, can be used to control the robot remotely from Ngui. The black dot in the window represents the current position of the joystick. By clicking on the black dot with certain combinations of mouse buttons, motion similar to the real joystick is realized:

- **Left Mouse Button, X -Y mode**: dragging the mouse while holding this button will move the robot's X and Y position and will not change the orientation.
- **Right Mouse Button, Frisbee mode**: dragging the mouse while holding this button will cause the robot to lock onto a fixed direction in world coordinates and move along that direction as commanded by the y-axis of the joystick and rotates as commanded by the x-axis.
- **Both Mouse Buttons, Differential mode**: Dragging the mouse while holding both buttons will move the robot forward as commanded by the y-axis of the joystick and turn as commanded by the x-axis. This is the most intuitive mode.

For all modes, the speed depends on how far from the center you move the joystick dot (the farther, the faster). Release the button to stop the motion.

☑ *A common mistake is to give an absolute meaning to the soft joystick directions, like N, S, E, W. 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 buttons together.*

Info Window

The Info window is at the bottom of the robot window. Inside the Info window are displayed the current values for the following:

Pose: this is a display of the robot's instantaneous integrated configuration (i.e. its position in world coordinates).

- **Speed**: this is a display of the robot's instantaneous speed.
- **Acceleration**: this is a display of the robot's current acceleration values.
- **Compass**: this is a display of the robot's compass value. This is not functional for release 1.0.
- **Batteries**: this is an instantaneous display of all four of the robot's battery voltages.

CHAPTER 5: THE ROBOT LANGUAGE

INTRODUCTION

The programming paradigm of XRDev is very simple: A single data structure, the `N_RobotState` structure, holds all the configuration and sensor data. A pointer to this structure is given by the function `N_GetRobotState`. The user then sets the desired parameters by modifying the contents of this structure. For example, to move the robot, the contents of the AxisSet field within `N_RobotState` are modified to reflect the desired motion, followed by a call to `N_SetAxes()`. Similarly, to retrieve information about the robot's axes, a call to `N_GetAxes()` is followed by reading the appropriate fields in `N_RobotState`. This example can be easily generalized to each subsystem on the robot, which has an appropriate "get" and often a corresponding "set" command.

COMMANDS

This section describes the various sensing and motion facilities on a typical Nomad robot and how they can be used. Please bear in mind that not all facilities are installed on every Nomad robot. For a list of accessories that have been installed on your robot, please consult the purchasing and/or factory paperwork.

Establishing Communication

The client program must first connect to a scheduler. The scheduler acts as a global timer (among other things) for the simulation when using a simulated robot and is required whenever there are more than two (real or simulated) robots in the client program. When connecting to a single real robot, however, the scheduler is not necessary because the real robot can act as its own scheduler. Connecting to the scheduler is done by calling `N_InitializeClient()`, which is declared as:

```
int N_InitializeClient (const char *scheduler_hostname, unsigned short
scheduler_socket);
```

Here, `scheduler_hostname` is the network hostname of the machine that is running Nscheduler and socket is the TCP/IP socket number that the scheduling process is listening on. As a special case, when there is only one robot (real or simulated), `N_InitializeClient()` should be called with the robot's hostname and listener socket. Socket numbers can be given to `Nscheduler` and `Nrobot` as command-line arguments or configuration file entries, and are displayed when these programs start running.

After establishing communication with the scheduler, communication to the robot must be established. This is accomplished with `N_ConnectRobot`, which is declared as:

```
int N_ConnectRobot (long RobotID);
```

where `RobotID` is the identification number of the robot.

Similarly, when the user wishes to discontinue communication with a robot, a call to `N_DisconnectRobot` should be made.

`N_DisconnectRobot` is declared as:

```
N_DisconnectRobot (long RobotID);
```

Timer Mechanism

The Timer Mechanism is a built-in safefy feature the prevents the robot from continuing to move when a client program is no longer functioning. It accomplishes this by keeping an internal timer that is reset each time a client program makes a call to the client library with a "set" or "get" command.

When the timer exceeds a user specified threshold, the motors in the base are turned off, as if the emer-

gency stop button was pressed. The timer threshold is stored in the N_RobotState structure in the Timer field, which is defined as:

```
struct N_Timer
{
  long Timeout;
  unsigned long Time;
};
```

Here, Timeout is the timer threshold, after which the base motors will be turned off. Time is the robot's master reference clock and represents the amount of time the robot has been powered up. Both parameters are in units of milliseconds. For safety reasons, the user cannot set the Timeout parameter to exceed 1500 milliseconds.

To set and retrieve the Timeout parameter, the client program can make calls to N_SetTimer and N_GetTimer, respectively. These functions are declared as:

```
int N_GetTimer (long RobotID);
int N_SetTimer (long RobotID);
```

Timestamps

Since there is often latency associated with obtaining sensor information, each sensor reading has a timestamp associated with it that provides the time at which the measurement was taken. The timestamp is found in the TimeStamp field of the associated part of the N_RobotState structure. The sensor latency is governed by the update rate of the sensor and the amount of time it takes to send the sensor information from the robot to the user.

For example, when the robot is moving rapidly and obtaining information both from low update-rate sensors (such as sonar) and high update-rate sensors (such as Integrated Configuration), it is very likely that the information from the two sensors is obtained at inconsistent times. This can cause large errors if the information from these sensors is being fused somehow (such as in a map.) The timestamps from the two sensors tell the user when the measurements were made with respect to a master reference clock (the Time field in N_Timer) and allows the information from the two sensors to be reconciled.

Extending the example further, a reasonable method to fuse sonar and Integrated Configuration information is to determine the Integrated Configuration values at the time the sonar measurements were taken (at the sonar's timestamp). This can be done easily by linearly interpolating between two appropriate Integrated Configuration values -- most likely an Integrated Configuration value that was taken *before* the sonar's timestamp and one that was taken *after*.

The N_RobotState Structure

The N_RobotState structure is the repository used in all data exchanges between a client program and a robot (real or simulated). It contains fields for all facilities available on the robot. It is defined as follows:

```
struct N_RobotState
{
  N_CONST long RobotID;
  N_CONST char RobotType;
  struct N_Integrator Integrator;
  struct N_AxisSet AxisSet;
  struct N_LiftController LiftController;
  struct N_Joystick Joystick;
  struct N_SonarController SonarController;
  struct N_InfraredController InfraredController;
  struct N_BumperController BumperController;
```

```
  struct N_Compass Compass;
  struct N_LaserSet LaserSet;
  struct N_S550Set S550Set;
  struct N_BatterySet BatterySet;
  struct N_Timer Timer;
};
```

A client program gains access to the `N_RobotState` structure with a call to `N_GetRobotState()` command, which is declared as:

```
struct N_RobotState *N_GetRobotState (long RobotID);
```

It returns a pointer to the active `N_RobotState` for the robot specified by `RobotID`. In general, `N_GetRobotState` is called only once at the beginning of the program for each robot that the client wishes to connect to. The pointer it returns is then referred to throughout the program when exchanging information with the robot.

When the client program wishes to retrieve all information from all facilities on the robot, a call to `N_GetState()` can be made. It is declared as:

```
int N_GetState (long RobotID);
```

A call to `N_GetState` is equivalent to executing all "get" commands except `N_GetTimer()` which must be called explicitly for an update.

Base Motion

Holonomic Versus Nonholonomic

A rigid body constrained to a plane has up to three degrees of freedom. In Cartesian space, these are often thought of as X, Y and rotation. The same rule applies to a mobile robot base moving on the floor. In the case of the Nomad 200, which uses a synchro-drive base, there are two axes of motion (not including turret): steering and translation. When the user wants to accelerate in a given direction, the wheels must first be oriented along that direction using the steering axis. This limits maneuverability and adds complexity to the control algorithm, which must explicitly take steering into account when the acceleration direction changes. The XR4000, on the other hand, can accelerate in any direction at any time, making it holonomic.

Global and Joint modes

The XR4000 has three convenient and intuitive axes of motion: X, Y and Rotation. We have defined two possible ways to control the XR4000 base:

Joint mode: this mode treats the XR4000 axes as joints. That is, a joint has a position, a positive direction and a negative direction that is defined with respect to the body to which it is attached. In joint mode, the XR4000 has three joints attached to the center of the robot. The Y joint (axis) creates linear movement along the forward / backward direction with respect to the robot. The X joint (axis) creates linear movement along the left / right direction with respect to the robot. The Rotation joint (axis) creates rotational movement with respect to the center of the robot.

For example, if we want the robot to move forward (by forward, we mean with respect to the robot's front), we instruct the Y axis to move in the positive direction. If we want, we can instruct the X and Y axes to move simultaneously to create diagonal motion. If we instruct the Rotation axis to move in the positive direction, the robot spins counter clockwise about its center. Simultaneous Rotation and Y axis movement in the positive direction will cause the robot to move in a circle. This is because the direction of the Y axis is defined with respect to the robot, which is rotating.

Global mode: this mode controls the XR4000 axes with respect to a fixed global reference frame. This global reference frame can be thought of as "drawn on the floor" and is defined when the robot is first turned

on or when calls to N_SetIntegratedConfiguration are made. In global mode, motion with the Y axis always causes movement along the Y direction in the reference frame regardless of the robot's rotational orientation. Thus, simultaneous rotation with the Rotation axis and Y axis motion will cause straight-line movement along the Y direction in the reference frame (as opposed to circular motion in Joint mode).

These modes specified in the N_AxisSet structure of N_RobotState. If the Global field in the N_AxisSet structure is set to TRUE, the axes are put in Global mode. If the Global field is set to FALSE, the axes are put in Joint mode. Note that this affects all axes, and Global/Joint modes cannot mixed among the axes. Mixing the Global/Joint modes could create ambiguous motion commands.

Axis Modes

Each of the robot's axes can be controlled in a separate axis mode (not to be confused with Global/Joint modes which apply to all axes: X, Y and Rotation). This mode is specified in the Mode field of the N_Axis structure, of which there is one per axis. The N_Axis structure is part of the N_AxisSet structure of N_RobotState.

Velocity Mode

In velocity mode, a user-specified velocity is set for the axis, and the axis maintains that velocity until the velocity or the mode is changed for that axis. The user also specifies an acceleration parameter that constrains how fast the velocity can change if the velocity needs to increase or decrease. This mode is specified by setting the Mode field to N_AXIS_VELOCITY.

Position Modes

In position mode, a user-specified destination position is set for the axis, and the axis moves to that position and stops. The user also specifies a desired speed and acceleration parameter. The desired speed specifies the magnitude of the velocity while the axis is moving to the destination position. The acceleration constrains how fast the velocity can change if the velocity needs to increase or decrease. The two possible position modes are *absolute* and *relative*, which are specified by setting the Mode field to N_AXIS_POSITION_ABSOLUTE or N_AXIS_POSITION_RELATIVE, respectively. Absolute position mode moves the axis to the absolute position with respect to the zero position of the axis. Relative position mode moves the axis to the position relative to the current position.

The N_Axis and N_AxisSet Structures

Inside the N_RobotState structure is the N_AxisSet structure which contains all the information concerning the base motion axes:

```
struct N_AxisSet
{
  BOOL Global;
  unsigned char Status;
  N_CONST unsigned int AxisCount;
  struct N_Axis Axis[N_MAX_AXIS_COUNT];
};
```

<toc 2>

- Global: when set to TRUE puts the base in Global mode and when set to FALSE puts the base in Joint mode.
- Status: can be one of the following values:
  N_AXES_READY: This indicates that the axes are available for movement.
  N_JOYSTICK_IN_USE: This status indicates that the base is being controlled via joystick.
  N_ESTOP_DOWN: This status indicates that one or more of the emergency stop buttons is

depressed, preventing the robot from moving.
N_MOTION_ERROR: The base was unable to execute the command motion.

The N_Axis structure contains information for each individual axis:

```
struct N_Axis
{
  BOOL DataActive;
  BOOL TimeStampActive;
  BOOL Update;
  unsigned long TimeStamp;
  char Mode;
  long DesiredPosition;
  long DesiredSpeed;
  long Acceleration;
  long TrajectoryPosition;
  long TrajectoryVelocity;
  long ActualPosition;
  long ActualVelocity;
  BOOL InProgress;
 };
```

- DataActive: A TRUE value for this parameter causes the values in this structure to be updated -- namely Mode, DesiredPosition, DesiredSpeed, Acceleration, TrajectoryPosition, TrajectoryVelocity, ActualPosition, ActualVelocity, InProgress, and TrajectoryVelocity.
- TimeStampActive: A TRUE value for this parameter causes the TimeStamp parameter to be updated.
- Update: A TRUE value for this parameter causes the input values (DesiredSpeed, DesiredPosition, and Acceleration) to be loaded into the set of working values for this axis when a call to N_SetAxes is made. The Update parameter allows one or more axes to be loaded with new input values simultaneously.
- TimeStamp: the time value in milliseconds that the axis values were measured.
- Mode: One of the following:

  N_AXIS_POSITION_RELATIVE: Specifies that the axis move relative to the current position.
  N_AXIS_POSITION_ABSOLUTE: Specifies that the axis move to an absolute position with respect to the absolute zero location of the axis.
  N_AXIS_VELOCITY: Specifies that the axis move with a constant velocity.
  N_AXIS_STOP: Causes the axis to decelerate to zero velocity.

- DesiredPosition: Specifies the desired endpoint position of the axis. The units are in millimeters for translational axes and milliradians for rotational axes. When Mode is set to N_AXIS_POSITION_RELATIVE or N_AXIS_POSITION_ABSOLUTE this specifies the endpoint position relative to the current position or the absolute position, respectively. This parameter is not used when Mode is set to N_AXIS_VELOCITY or N_AXIS_STOP. If in Global mode (Global=TRUE), the DesiredPosition is with respect to the global reference frame. If in Joint mode (Global=FALSE), the DesiredPosition is in joint coordinates.
- DesiredSpeed: Specifies the speed at which this move is to be executed. This parameter is not used when Mode is set to N_AXIS_STOP.
- Acceleration: Specifies the acceleration to the DesiredSpeed or subsequent accelerations if DesiredSpeed is increased during a move. This also specifies the deceleration from the DesiredSpeed when an endpoint position is reached (when Mode is either N_AXIS_POSITION_RELATIVE or

N_AXIS_POSITION_ABSOLUTE) or when the DesiredSpeed is decreased. The units are in millimeters / second for translational axes and milliradians / second for rotational axes.

- TrajectoryPosition: Provides the current position of the trajectory generator. If in Global mode (Global=TRUE), the TrajectoryPosition is with respect to the global reference frame. If in Joint mode (Global=FALSE), the TrajectoryPosition is in joint coordinates.
- TrajectoryVelocity: Provides the current velocity of the trajectory generator. If in Global mode (Global=TRUE), the TrajectoryVelocity is with respect to the global reference frame. If in Joint mode (Global=FALSE), the TrajectoryVelocity is in joint coordinates.
- ActualPosition: Provides the actual position of the axis. The value of this field is based on the setting of the Global field. If in Global mode (Global=TRUE), this field provides the position of the axis in the global reference frame (this same value can be found in the N_Integrator structure.) If in Joint mode (Global=FALSE) this field provides the joint position of the axis.
- ActualVelocity: Provides the actual measured velocity of the axis. If in Global mode (Global=TRUE), the ActualVelocity is with respect to the global reference frame. If in Joint mode (Global=FALSE), the ActualVelocity is in joint coordinates.

This value, like ActualPosition is based on the setting of the Global field in N_AxisSet. If the Global field is set, this field provides a value that is with respect to a fixed global reference frame.

- InProgress: Provides a boolean value that informs the user that an axis is currently moving.

Information in the N_AxisSet structure is updated with calls to N_GetAxes, which is declared as:

```
int N_GetAxes(long RobotID);
```

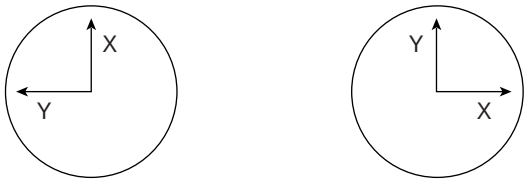
Modified values in the N_AxisSet are uploaded into the motion axes for movement upon calling N_SetAxes, which is declared as:

```
int N_SetAxes(long RobotID);
```

The Integrated Configuration

Each Nomad robot is constantly estimating its Cartesian position (X, Y, Rotation) with respect to a global coordinate frame that is “drawn on the floor”. This is often called the dead-reckoned position, and it is useful as an “extra sensor” that indicates where the robot is located in the environment. However, this estimate tends to drift over time with respect to the actual position, limiting its usefulness over long periods.

It is estimated by measuring changes in position (dX, dY, dRotation) over very small time increments (typically 5 ms) and integrating those changes over time -- hence it is called the Integrated Configuration.



Nomad 200 Nomad XR4000

The Integrated Configuration can be obtained by making calls to N_GetIntegratedConfiguration() and retrieving the values in the N_Integrator field of the N_RobotState. Similarly, the Integrated Configuration can be set to any configuration by modifying the contents of the N_Integrator and making a call to N_SetIntegratedConfiguration().

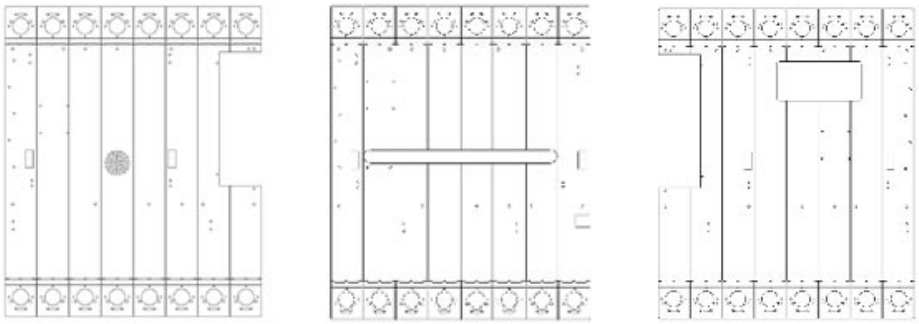
The `N_Integrator` structure is defined as follows:

```
struct N_Integrator
{
  BOOL DataActive;
  BOOL TimeStampActive;
  long x;
  long y;
  long Steering;
  long Rotation;
  unsigned long TimeStamp;
};
```

Tactile Sensing

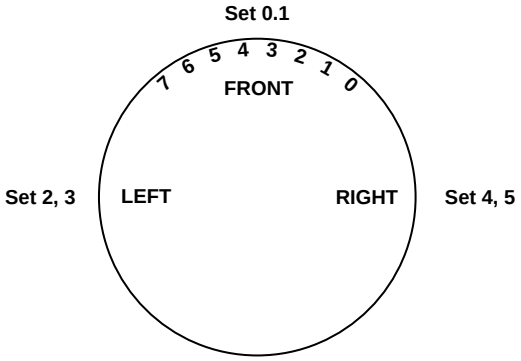
Tactile sensors provide information about physical contact with objects in the environment. It is hoped that non-contact or proximity sensing will sense all obstacles and prevent physical contact, but this is not guaranteed. Tactile sensors are often called “collision sensors”. The Nomad XR4000 has forty-eight bi-level tactile sensors that surround its top and bottom perimeters. Additionally, the XR4000 has four “door bumpers” on each door that sense contact *between* the top and bottom perimeters. Together, these tactile sensors provide tactile information over the entire robot.

The XR4000 has 3 doors that go counterclockwise and there are two (sonar / infrared / bumper) sets per door to make six sets total.



Left *Front* *Right*

Looking at the top of the robot:



Set 0 = top set on front door (sensor #0–7)
Set 1 = bottom set on front door (sensor #0–7)
Set 2 = top set on left door (sensor #0–7)
Set 3 = bottom set on left door (sensor #0–7)
Set 4 = top set on right door (sensor #0–7)
Set 5 = bottom set on right door (sensor #0–7)

The tactile sensor information is contained in the `N_BumperController` structure of `N_RobotState`, which is defined as:

```
struct N_BumperController
{
  N_CONST unsigned int BumperSetCount;
  struct N_BumperSet BumperSet[N_MAX_BUMPER_SET_COUNT];
};
```

- `BumperSetCount`: the number of bumper sets in the controller.

The `N_BumperSet` structure is defined as follows:

```
struct N_BumperSet
{
  BOOL DataActive;
  BOOL TimeStampActive;
  N_CONST unsigned int BumperCount;
  struct N_Bumper Bumper[N_MAX_BUMPER_COUNT];
};
```

- `DataActive`: `TRUE` if bumper data for this set is to be updated.
- `TimeStampActive`: `TRUE` if the time (time of acquisition of the tactile range) is to be updated for this set.
- `BumperCount`: the number of bumpers in the set.

The XR4000's four door bumper values are stored at the end of the top set of each door (indices 8 through 11). For instance, bumper set 2 on the XR4000 is the top set of bumpers on the second door, so the readings for the door bumpers on the second door will be stored in bumper set 2, so there are actually 12 readings in bumper set 2. This is why `N_XR4000_BUMPER_COUNT` is 12 instead of 10. Door bumper readings are either `N_BUMPER_NONE` or `N_BUMPER_LOW`.

The `N_Bumper` structure is defined as:

```
struct N_Bumper
{
  char Reading;
  unsigned long TimeStamp;
};
```

The values defined for each bumper are:

- `Reading`: one per bumper and it will always be one of the three bumper value constants defined in `Nclient.h`. For the Nomad 200, bumpers have only the `N_BUMPER_NONE` or `N_BUMPER_LOW` value. For the XR4000, they can also be `N_BUMPER_HIGH` (for a hard hit).
- `TimeStamp`: the time of the acquisition of the reading.

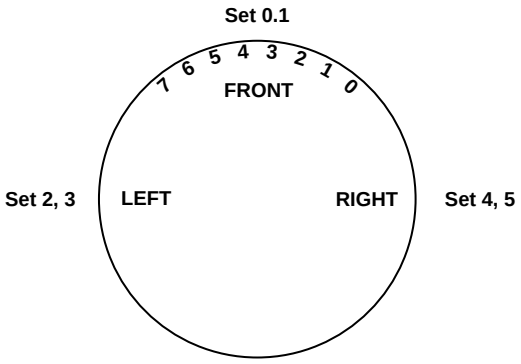
Obtaining information from the tactile sensors is accomplished with calls to `N_GetBumper()`, which puts the updated bumper information into the `N_BumperController` structure.

Infrared Proximity Sensing

The infrared proximity sensors provide range information to nearby objects (typically less than 30 to 50 centimeters away). They determine range by emitting infrared energy using high-current LED's and sensing the amount of returned energy with infrared photodiodes. The returned energy is inversely proportional to the distance to the closeby object -- thus, these sensors are used as distance or proximity sensors.

The returned energy is also a function of the object's reflectivity. High reflectivity objects return large amounts of IR energy and low reflectivity objects return proportionally lower amounts of IR energy. The difference in reflectivity between objects can cause errors in range measurements if not taken into account.

The XR4000 has 48 infrared proximity sensors on the top and bottom perimeters of the robot arranged in six sets of eight. That is, the XR4000 has 3 doors that go counterclockwise and there are two (sonar / infrared / bumper) sets per door for a total of six sets.



Set 0 = top set on front door (sensor #0–7)
Set 1 = bottom set on front door (sensor #0–7)
Set 2 = top set on left door (sensor #0–7)
Set 3 = bottom set on left door (sensor #0–7)
Set 4 = top set on right door (sensor #0–7)
Set 5 = bottom set on right door (sensor #0–7)

Infrared proximity sensing information is contained in the `N_InfraredController` structure in the `N_RobotState` structure. `N_InfraredController` is defined as:

```
struct N_InfraredController
{
  BOOL InfraredPaused;
  N_CONST unsigned int InfraredSetCount;
  struct N_InfraredSet InfraredSet[N_MAX_INFRARED_SET_COUNT];
};
```

The configuration data used globally by all the infrared sets on the robot are:

- `InfraredPaused`: When set to `TRUE`, all the infrared sensors will be stopped.
- `InfraredSetCount`: the number of infrared sets in the controller.

The `N_InfraredSet` structure is defined as follows:

```
struct N_InfraredSet
{
  BOOL DataActive;
  BOOL TimeStampActive;
  N_CONST unsigned int InfraredCount;
  struct N_Infrared Infrared[N_MAX_INFRARED_COUNT];
};
```

It contains an array of infrared structures (one per infrared transducer in the set), plus a number of configuration parameters. The configuration parameters are:

- `DataActive`: `TRUE` if infrared data for this set is to be updated.
- `TimeStampActive`: `TRUE` if the time (time of acquisition of the infrared range) is to be updated for this set.
- `InfraredCount`: the number of infrared sensors in the set.

The `N_Infrared` structure is defined as:

```
struct N_Infrared
{
  long Reading;
```

```
  unsigned long TimeStamp;
};
```

The values defined for each infrared are:

- `Reading`: a value from 0 to 255 representing the amount of reflected infrared energy returned from a closeby object. A value of 0 represents no energy reflected (far object) while a value of 255 represents maximum energy reflected (close object). If a distance value is needed, the user should build a calibration table by measuring the energy returned by a material sample (representative of what can be found in the environment), at various distances.
- `TimeStamp`: the time of the acquisition of the raw value.

Obtaining information from the infrared sensors is accomplished with calls to `N_GetInfrared()`, which puts the updated infrared information into the `N_InfraredController` structure.

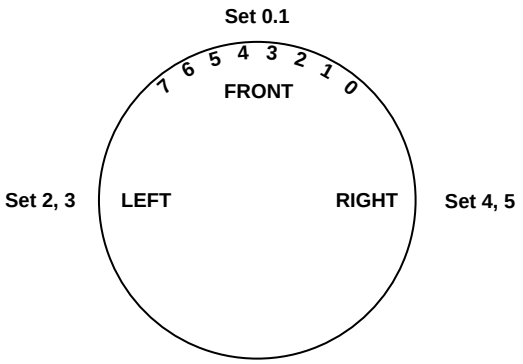
Sonar Proximity Sensing

The sonar proximity sensors provide range information to objects that are relatively far away (between 15 and 700 centimeters.) Distance information is obtained by multiplying the speed of sound by the "time of flight" of a short ultrasonic pulse travelling to and from a nearby object.

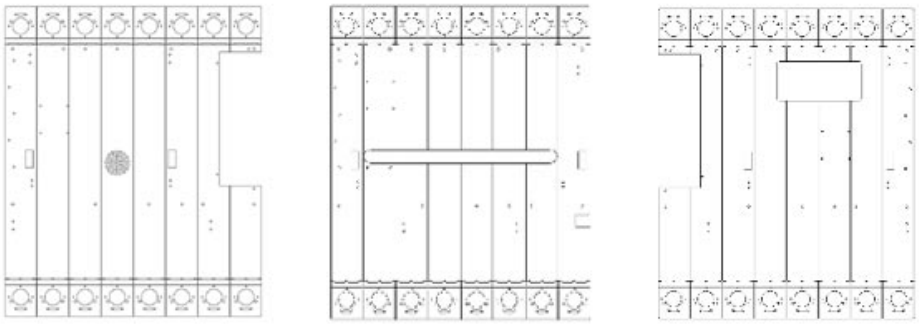
The sonar controller structure holds all the configuration data that is valid for all the sonar sets on the robot. Sonar sets are groups of sonar that act together. For instance, a firing order can be defined among the sonar of a given set. The sonar controller structure has an array of `N_SonarSet` structures, each describing one particular set.

The Nomad 200 has only one set of 16 sonar, going counterclockwise from 0 to 15, with 0 in front.

The XR4000 has 48 sonar proximity sensors on the top and bottom perimeters of the robot arranged in six sets of eight. That is, the XR4000 has 3 doors that go counterclockwise and there are two (sonar / infrared / bumper) sets per door for a total of six sets.



Set 0 = top set on front door (sensor #0–7)
Set 1 = bottom set on front door (sensor #0–7)
Set 2 = top set on left door (sensor #0–7)
Set 3 = bottom set on left door (sensor #0–7)
Set 4 = top set on right door (sensor #0–7)
Set 5 = bottom set on right door (sensor #0–7)



Left *Front* *Right*

The `SonarController` structure in `N_RobotState` is defined as:

```
struct N_SonarController
{
  N_CONST unsigned int SonarSetCount;
  struct N_SonarSet SonarSet[N_MAX_SONAR_SET_COUNT];
  BOOL SonarPaused;
};
```

The configuration data used globally by all the sonar sets on the robot are:

- `SonarSetCount`: the number of sonar sets in the controller.

The `N_SonarSet` structure is defined as follows:

```
struct N_SonarSet
{
  unsigned int FiringOrder[N_MAX_SONAR_COUNT + 1];
  long FiringDelay;
  long BlankingInterval;
  BOOL DataActive;
  BOOL TimeStampActive;
  N_CONST unsigned int SonarCount;
  struct N_Sonar Sonar[N_MAX_SONAR_COUNT];
};
```

It contains an array `Sonar` of sonar structures (one per sonar transducer in the set), plus a number of configuration parameters. The configuration parameters are:

- `FiringDelay`: delay in milliseconds between two consecutive sonar firings. Setting this parameter to a large value can prevent echos from previous sonars from being received.
- `BlankingInterval`: the time in milliseconds to wait after a sonar sensor has fired before the sensor begins to listen. **This is currently not implemented on the XR4000 Release 1.0.**
- `DataActive`: set to a `TRUE` value if the sonar data is to be updated for this set.
- `TimeStampActive`: set to a `TRUE` value if the time (time of acquisition of the sonar range) is to be updated for this set.
- `SonarCount`: number of sonars in this set.
- `FiringOrder`: an array of sonar indices terminated by `N_END_SONAR_FIRING_ORDER` if the length of the array is less than the `SonarCount`.

The N_Sonar structure contains the sensor readings:

```
struct N_Sonar
{
  long Reading;
  unsigned long TimeStamp;
};
```

The values defined for each sonar are:

- Reading: the distance measurement of the sensor in millimeters. When a sensor receives no echo (e.g. due to specularity or excessive distance), this value will be N_SONAR_TIMEOUT.
- TimeStamp: the time that the measurement took place.

Obtaining information from the sonar proximity sensors is accomplished with calls to N_GetSonar(), which puts the updated sonar information into the N_SonarController structure.

Similarly, obtaining configuration information is accomplished with calls to N_GetSonarConfiguration(). Setting the sonar configuration is accomplished by modifying the configuration parameters in the N_SonarSet structure(s) (e.g. FiringOrder, FiringDelay, BlankingInterval) and calling N_SetSonarConfiguration().

Laser (Sensus 550)

The Sensus 550 is a "time of flight" laser rangefinding system based on the Sick Electro-optic LMS sensor. Obtaining information from the sensor is accomplished with calls to N_GetS550() after which the data can be retrieved from the N_S550Set structure in N_RobotState.

The N_S550Set structure is defined as:

```
struct N_S550Set
{
  N_CONST unsigned int S550Count;
  struct N_S550 S550[N_MAX_S550_COUNT];
};
```

- S550Count: the number of Sensus 550 laser devices in the set

The N_S550 structure is defined as:

```
struct N_S550
{
  N_CONST unsigned int TotalPoints;
  unsigned int RequestedPoints;
  unsigned long Readings[N_MAX_S550_POINTS];
  unsigned char StatusFlags[N_MAX_S550_POINTS];
  unsigned char SummaryFlags;
  unsigned long TimeStamp;
  BOOL DataActive;
  BOOL TimeStampActive;
};
```

- TotalPoints: The number of points in the Readings array. TotalPoints gets set upon initialization to either 180 or 361 points according to the model used and should not be modified.
- RequestedPoints: The number of measurements that the user desires per laser scan. This can be one of a set of possible values (9, 10, 15, 18, 30, 45, 90, 180, 361). If a value other than these values is requested, an N_INVALID_ARGUMENT result will be returned. Fewer parameters than 361 cause the

sensor to return the request amount of measurements distributed evenly across the 180° viewing angle (e.g. for a `RequestedPoints` of 10, the measurements will be 18° apart.)

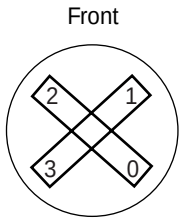
- `Readings`: The set of sensor readings in millimeters with the 0th element in the array being the first, rightmost reading. That is, the sensor takes measurements in order from right to left with respect to the sensor.
- `StatusFlags`: Status states for each entry in the `Readings` array. This will eventually be used to indicate if a reading violated the safe or warning fields, and various other possible conditions. **Not used in 1.0.**
- `SummaryFlags`: Status flags applicable to the entire set of readings. E.g. will indicate if any of the readings violated the safe field. **Not used in 1.0.**
- `TimeStamp`: The time at which the most current laser scan took place.
- `DataActive`: If this is set to a TRUE value, this structure will be updated when calls to the client library are made.
- `TimestampActive`: If this is set to a TRUE value, the TimeStamp field will be updated when calls to the client library are made.

Power System

Information about the power system consists of the measured voltages of each of the batteries on the XR4000. These can be retrieved with calls to `N_GetBattery()` after which the contents of the `N_BatterySet` structure of the `N_RobotState` can be read. The `N_BatterySet` is defined as:

```
struct N_BatterySet
{
  struct N_Battery Battery[N_MAX_BATTERY_COUNT];
  BOOL DataActive;
};
```

The mapping between the Battery array position and the physical batteries on the robot is given below. This is a view from the top of the XR4000:



The N_Battery structure is defined as:

```
struct N_Battery
{
   long Voltage;
};
```

- `Voltage`: Provides the measured battery voltage in millivolts.

Compass

The Compass sensor provides information about the robot's heading with respect to magnetic North. Obtaining this information is accomplished with calls to `N_GetCompass()`, after which the contents of the `N_Compass` structure in the `N_RobotState` can be read. **The compass is not support in release 1.0.**

The `N_Compass` structure is defined as follows:

```
struct N_Compass
{
  long Reading;
  unsigned long TimeStamp;
  BOOL DataActive;
  BOOL TimeStampActive;
};
```

- `Reading` : represents the heading of the turret with respect to the magnetic north in milliradians
- `TimeStamp`: the time of the acquisition of the reading
- `DataActive`: `TRUE` if compass data for this set is to be updated
- `TimeStampActive`: `TRUE` if the time (time of acquisition of the compass data) is to be updated for this set.

Voice Synthesizer

The voice synthesizer allows strings of ASCII text to be spoken. It uses an embedded text to speech algorithm that translates plain english text into phonemes. This is accomplished by passing a null terminated string to the `N_Speak()` command.

`N_Speak` is declared as:

`N_SpeakN_Speak` (unsigned long RobotID, char *Text);

Lift Mechanism

See *"Chapter 6: Nomad XR4000 Lift Mechanism Reference"*.

CHAPTER 6: NOMAD XR4000 LIFT MECHANISM REFERENCE

INTRODUCTION

The Nomad XR4000 Lift Mechanism is a unique three stage telescoping mechanism that mounts in the back of the XR4000 mobile robot. The Lift Mechanism has a servo-controlled deploy axis and can automatically retract and be stored inside the base when not in use. With its gripper, this system can be used to perform mobile manipulation tasks. The Lift Mechanism provides motion in the Z direction and utilizes the drive system of the base to provide X, Y, and Rotation degrees of freedom. Although this system does not use brakes, it will hold its position when the robot is shutdown.

The Lift mechanism consists of two controllable axes and the Deploy axis:

1 The Lift Axis, which moves the gripper up and down (normal to the floor). It can move the gripper to within 3 cm of the floor and to a height of 940 cm above the floor.

2 The Grip axis, moves the fork-like "fingers" of the gripper to a fully closed position and to an open position of 20 cm between fingers.

3 The Deploy axis, simply moves the entire Lift Mechanism in and out of the robot. When the Deploy axis is in the deployed position movement of the Grip and Lift axes is permitted, however the mechanism extends beyond the outside diameter of the robot. In this position, it is possible for the mechanism to collide with obstacles and damage the mechanism and / or the environment. This makes path-planning and obstacle avoidance more difficult. When the Deploy axis is in the retracted position, the mechanism does not extend beyond the outside diameter of the robot. This protects the mechanism from potential damage and simplifies path-planning and obstacle avoidance.

Below is a chart which specifies the sign conventions and travel limits of the axes:

Parameter	Lift Axis	Grip Axis	Units
Positive travel	toward ceiling	fingers moving apart	
Negative travel	toward floor	fingers moving closed	
Zero Position	center of travel	fingers completely closed	
Positive travel limit	4800	2000	0.1 mm
Negative travel limit	-4650	0	0.1 mm
Maximum velocity	800	600	0.1 mm/s
Maximum acceleration	800	600	0.1 mm/s^2

Here, the Positive travel or Negative travel convention is the direction of movement that results in the position of the axis becoming more positive or more negative, respectively. The Zero Position is the location of the axis where the absolute position is zero. The Positive and Negative limits are the absolute positions within which movement is allowed.

Controlling the Lift Mechanism follows the same paradigm as controlling the base of the XR4000 -- it uses

a data structure, N_LiftController, which contains the desired control parameters as well as the configuration information. For example, to move the lift mechanism, the desired control parameters are written into the structure followed by a call to N_SetLift(). Similarly when information about the lift mechanism is needed (such as position of gripper), a call to N_GetLift() is followed by reading the needed information from the N_LiftController data structure.

Zeroing

When the robot is turned on, the position of the lift mechanism's axes are not known. In order to control the lift mechanism, the axis positions must be determined. This is done by "zeroing" the lift mechanism with a call to N_ZeroLift(). Only one call the N_ZeroLift() is required for each boot cycle of the robot (each power-on / power-off cycle.). N_ZeroLift() is declared as:

```
int N_ZeroLift(long RobotID, BOOL Force);
```

Where RobotID is the identification number of the robot with the Lift Mechanism. The Force argument when passed as FALSE, will zero the lift mechanism *if it has not been zeroed during the current boot cycle*. The zeroing motions entail the mechanism fully deploying, the grip axis closing fully and the lift axis moving to the highest position. During the same boot cycle, subsequent calls to N_ZeroLift() (with the Force argument passed as FALSE) will not result in the mechanism performing the zeroing motions. However, the mechanism will perform the zeroing motions each time the Force argument is passed as TRUE.

While the Lift Mechanism is zeroing. the InProgress field in the N_LiftController structure (see "*The LiftController Structure*"section) is set to a TRUE value and reset to FALSE upon completion. While this field is TRUE, no other movement commands to the lift mechanism can be or should be executed.

Deploying and Retracting

In order to move the Grip or Lift axes, the Lift Mechanism must be deployed first if it has not been already. This is accomplished with a call to N_DeployLift(), which is declared as:

```
int N_DeployLift(long RobotID);
```

where RobotID is the identification number of the robot with the lift mechanism. In general, before you attempt to move either the Lift or the Grip axes a call to N_DeployLift() should be made to ensure that the mechanism is deployed. If it is not deployed, an error will be returned when a movement is attempted, but no movement will result.

When the user wishes to retract the arm into the body of the robot to prevent damage due to collisions or simplify path planning, a call to N_RetractLift() can be made. This procedure moves the Lift and Grip axes into a "retractable" position and retracts the entire mechanism into the body of the XR4000. It is declared as:

```
int N_RetractLift(long RobotID);
```

where RobotID is the identification number of the robot with the lift mechanism.

While the Lift Mechanism is deploying or retracting (or zeroing) the InProgress field in the N_LiftController structure (see "*The LiftController Structure*" below) is set to a TRUE value and reset to FALSE upon completion. While this field is TRUE, no other movement commands to the lift mechanism can be or should be executed.

The LiftController Structure

All control and configuration information of the Lift Mechanism is handled through the LiftController structure which is part of the N_RobotState structure. Control is accomplished by modifying the LiftController contents (writing the desired motion parameters) and making a call to N_SetLift().

Similarly, configuration information can be obtained by making a call to N_GetLift() and reading the desired configuration information out of the LiftController structure. The LiftController type is defined as follows:

```
struct N_LiftController
{
  BOOL Deployed;
  BOOL InProgress;
  struct N_LiftAxis Axis[N_LIFT_AXIS_COUNT];
};
```

The fields are described as follows:

- Deployed: provides information on whether the Lift Mechanism is fully deployed or not. A TRUE value indicates that the Lift mechanism is fully deployed.
- InProgress: a TRUE value in this field indicates that the Lift Mechanism is currently being retracted, deployed, or zeroed (i.e. due to a call to N_DeployLift(), N_RetractLift(), or N_ZeroLift()). While this field is TRUE, no motion commands to the Lift mechanism are accepted.
- Axis: an array that contains the motion parameters for the two controllable axes: the Lift and Grip axes. The two possible array indices are respectively N_LIFT and N_GRIP.

The axis parameters are contained in the N_LiftAxis structure, which is defined as:

```
struct N_LiftAxis
{
  BOOL DataActive;
  BOOL TimeStampActive;
  BOOL Update;
  unsigned long TimeStamp;
  char Mode;
  long Status;
  long DesiredPosition;
  long DesiredVelocity;
  long MaxMotor;
  long Acceleration;
  long Position;
  long Velocity;
};
```

- DataActive: a TRUE value for this field causes the parameters of this structure to be updated with each call to N_GetAxes(). These parameters are: Status, DesiredPosition, DesiredVelocity, DesiredAcceleration, MaxMotor, Position and Velocity.
- TimeStampActive: a TRUE value for this field causes the TimeStamp field to be updated with each call to N_GetAxes().
- Update: a TRUE value for this field causes the motion parameters to be loaded into the motor controller for execution upon calling N_SetAxes(). These parameters are: DesiredPosition, DesiredVelocity, DesiredAcceleration, and MaxMotor.
- TimeStamp: contains the time in milliseconds that the motion parameters were obtained.
- Mode: contains the movement mode for the axis. It can be set to one of the following posible modes:

  N_LIFT_POSITION_RELATIVE: causes the axis to move to a position relative to the current position as specified by the DesiredPosition field.

  N_LIFT_POSITION_ABSOLUTE: causes the axis to move to an absolute position (see *"Lift Mechanism*

Convention" table) as specified by the `DesiredPosition` field.

`N_LIFT_VELOCITY`: causes the axis to move at a velocity specified by the `DesiredVelocity` field.

- `Status`: provides a bitmap of possible errors. One or more of these bits will be set if an error occurs during a movement. To test for a particular error, the value or result can be "ored" with the error bits described below:

  N_LIFT_POS_LIMIT: indicates the positive limit switch has been reached for the axis. This should never happen, as the software limits travel to never traverse the positive limit of neither the Grip nor Lift axes after the Lift Mechanism is zeroed.

  N_LIFT_NEG_LIMIT: indicates the negative limit switch has been reached for the axis. This should never happen for the Lift axis, but the Grip axis when fully closed depresses the gripper's negative limit switch.

  `N_LIFT_MOTION_ERROR`: indicates that the motion was unable to complete. This is often due to an obstacle in the desired path of the axis.

  `N_LIFT_ESTOP`: indicates that the motion is not possible because one or more of the emergency stop switches is depressed.

  `N_LIFT_CANNOT_MOVE`: indicates that the motion is not possible because the Lift Mechanism is not zeroed or not deployed.

- `DesiredPosition`: specifies the desired endpoint position of the axis in units of 0.1 mm. The position is either relative to the current position or absolute with respect to the zero position as specified by the `Mode` field.
- `DesiredVelocity`: specifies the desired velocity at which the axis should move in units of 0.1 mm/s.
- `MaxMotor`: specifies the percentage of available torque to use for the axis. The possible values range between 0 and 100, where 100 specifies the default of 100% available torque. For example, this is useful to set for the gripper when picking up fragile objects.
- `Acceleration`: specifies the desired acceleration at which the axis should move in units of 0.1 mm/s^2.
- `Position`: provides the current absolute position of the axis in units of 0.1mm.
- `Velocity`: provides the current velocity of the axis in units of 0.1mm/s.

CHAPTER 7: VISION REFERENCE

OVERVIEW

This chapter describes the use of different vision systems available for the Nomad XR4000 and Nomad 200 robots. The Sensus 700 is an embedded vision system with on-board DSP's for processing images.

SENSUS 450 MONOCHROME VISION

The Sensus 450 is a complete monochrome vision system including a camera and PCI framegrabber card. The device driver and simple example programs are provided for the user. More elaborate image analysis algorithms are left to the user to customize.

Running a demo

If your robot is equipped with an auxiliary processor, the vision system is installed on the auxiliary processor. Plug a keyboard and VGA monitor into the "Keyboard2" and "VGA2" ports to run the demo. If your robot has a single processor, plug a keyboard and VGA monitor into the "Keyboard1" and "VGA1" ports to run the demo.

Through the VGA monitor and keyboard, change directories into `/usr/local/robot-devices/dt3155/examples/video` and run `video`. This program will switch video modes on the monitor and display a grayscale image at about 15 frames per second. *Note that the iris of the camera may be closed, resulting in little or no image.*

If your robot is equipped with two Sensus 450's, run the video program with command line arguments 0 and 1 to see video from the different systems, for example:

```
video 0
```

for the first device, and

```
video 1
```

for the second device.

SENSUS 460 COLOR VISION

The Sensus 460 is a complete color vision system including a color composite camera and PCI framegrabber card. The device driver and simple example programs are provided for the user. More elaborate image analysis algorithms are left to the user to customize.

Running a simple demo

If your robot is equipped with an auxiliary processor, the vision system is installed on the auxiliary processor. Plug a keyboard and VGA monitor into the "Keyboard2" and "VGA2" ports to run the demo. If your robot has a single processor, plug a keyboard and VGA monitor into the "Keyboard1" and "VGA1" ports to run the demo.

Through the VGA monitor and keyboard, change directories to `/usr/local/robot-devices/meteor/examples/video` and run `video` to see continuous color video images. You should see a color image at about 5 frames / second. *Note that the iris of the camera may be closed, resulting in little or no image.*

If your robot is equipped with two cameras, but one Meteor card, run the video program with the command line arguments 0 and 1 to see video from the different cameras, for example:

```
video 0
```

for the first camera, and

```
video 1
```

for the second camera.

If your robot is equipped with two Meteor cards, run the video program with two command line arguments, for example: `video 1 0` for second card, first camera, `video 1 1` for second card, second camera, and `video 0` for first card, first camera.

SENSUS 700 HIGH SPEED COLOR VISION SYSTEM

The driver, documentation and examples for the Sensus 700 are in the directory`/usr/local/robot-modules/sensus7h` directory on the robot. The following sections describe how to run a simple demonstration program and how to use the rpc library. More elaborate image analysis algorithms are left to the user to customize.

Demonstration program

Log onto the robot from an X server and export the display by typing:

```
export DISPLAY=<machine name or IP number>:0
```

where “machine name or IP number” is the name of the machine with the X server or its IP address (the IP address must be used if the name is not included in `/etc/hosts`} file on the robot). You may also need to run `xhost +` locally on the X server you are using to give the robot permission to use the X server.

Change directories into `/usr/local/xrdev/robot-modules/sensus7h/host/video` on the robot and run `video`. Although it is slow over radio Ethernet, it should pop-up a window and display grayscale video frames on the X server. Note that the video is actually color, but the display only shows grayscale.

Using the RPC Library with the Sensus 700

Introduction

The Sensus 700 is an “embedded processing” card. It differs from traditional frame-grabbers in that it has processing onboard for doing image-processing tasks. The resulting processed data is typically the only thing that is exchanged between the host computer (486, Pentium) and the vision system. This leaves the host processor free to do other things, which is very useful in mobile robotics. That is, the host processor has other things to do such as avoid obstacles, build maps, etc.

This means that an application that uses the Sensus 700 is comprised of two programs that run simultaneously — one that runs on the host processor and one that runs on the vision system. These two programs communicate through the device driver. Typically, the host processor is controlling what the vision system does by making specific requests such as “where in the image is a particular beacon?” or “do you see a coke can?”. Once the host processor makes the request, it can do other things until the result comes back. This computational model resembles a client-server type system.

The Remote Procedure Call (rpc) library makes designing such systems intuitive and easy. For example, the host makes an rpc request to the vision system that results in a procedure being called on the vision system. The vision system runs the code in the procedure and returns the result to the host processor. The host processor receives the result much like it receives data from a procedure that it calls locally, except this time, the procedure was executed onboard the vision system. Thus the rpc library allows both the exchange of data and control of execution.

Examples

The best way to learn about the rpc library is to look at the example programs in the following directories:

```
../host/hworld     A simple “hello world” program.
```

../host/rpcexamp	Performs several rpc requests to the vision system passing different data types and using different mechanisms to receive data.
../host/video	Grabs video frames from the digitizer and displays them in black and white on an X-server.

Let's look at the rpcexamp program. In rpcexamp.h there is the rpc_table which each program that is linked with the rpc library MUST have. It consists of a list of possible rpc procedures for that application and the type of data that are exchanged. For example, the "add" procedure passes three integers to the vision system (From the Host, FH) and returns one integer From the Vision system (FV). Each entry of the table always has the list of data types from the vision system first followed by the list of data types from the host. Each entry in the rpc_table is meant to resemble (as much as possible) a C procedure declaration, namely for "add":

```
int add (int arg1, int arg2, int arg3);
```

One thing that the rpc library allows (that C doesn't) is multiple data types to be returned from a procedure call. An example of this is in get_ints_int_chars. Here, the vision system returns an array of ints, a single int and an array of chars.

Let's look at rpcexamp.c to see how to write a program that uses the rpc mechanism. The first call of an rpc program is the vis_upload() call, which loads the code that is executed on the vision system into the vision system's embedded processors. The pathname of the vision system application is given. The vision application is also written and compiled in C (Parallel C). We will talk about how embedded programs for the vision system are written and compiled in Section 5.3. The next call is to rpc_init(), which initializes various data structures, etc. Next, we make an "add" rpc request to the vision system by calling rpc_call_func(ADD,10,20,30). The vision system receives this request and adds the three numbers together. Looking at the vision system code (../vis/rpcexamp.c) the "add" procedure gets called. Notice that it has three arguments as intended. Inside this procedure the vision system returns the data with the rpc_return_data() call. Notice that one integer is returned as indicated in the rpc_table.

Meanwhile, the host is waiting for the data while waiting in rpc_dispatch(). rpc_dispatch() can either wait or not wait for data depending on if WAIT or NO_WAIT is passed. Here, we want to wait. If, for example, we didn't want to wait, passing NO_WAIT would fall through and return TRUE if a request was serviced. This allows the developer to write something like:

```
rpc_call_func(ADD, 10, 20, 30);
while(!rpc_dispatch(NO_WAIT))
{
  /* do something useful */
  build_map();
  avoid_obstacles();
}
```

Here, we want to do something else until the result returns. When the result returns from "add" in rpcexamp, it arrives in an rpc request form from the vision system and the "add" function on the host side gets called (this time with the result). This is referred to as the "callback" mechanism. The callback mechanism is convenient for an event-driven coding style. This brings up a key point in the rpc library — it is a completely symmetric protocol. That is, we've restricted our discussion to rpc requests that originate from the host. The vision system can originate requests as well, hence making the rpc protocol symmetric (as we've seen already in hworld and rpcexamp, printf() executed in the vision system is an rpc request that originates from the vision system to the host). For example, if the vision system had a piece of code that detected obstacles in the path of the robot, the vision system may make an rpc request to the host hence notifying of the possible collision condition. (without the host requesting the information) This is

accomplished with rpc_call_func() from code running on the vision system in exactly the same way we have described.

The other data return mechanism (other than callback) is the "select" mechanism, which gets its name from the Unix select() system call. The select mechanism allows the return data to be retrieved with a library call instead of a callback. Notice in rpcexamp the rpc_select_func() call after the rpc_call_func(). This call waits until the sum result returns from the vision system and puts it in the result variable. Notice that the WAIT option was used. We could have used NO_WAIT in much the same way we use it in the rpc_dispatch() example described above. That is, if NO_WAIT was used, rpc_select_func() would return TRUE if the data was returned and FALSE otherwise. For example:

```
rpc_call_func(ADD, 10, 20, 30);
while(!rpc_select_func(WAIT, ADD, NULL, &result))
{
  /* do something useful */
  build_map();
  avoid_obstacles();
{
printf("Result has arrived: %d\n", result);
```

This allows us to do a useful task until the result from add arrives. The result is conveniently returned in a local variable.

The get_ints_int_chars() procedure in rpcexamp demonstrates passing different data types, namely integer and character arrays. The only difference here is the way in which arrays are passed. Both the size of the array and the pointer must be passed in order for the rpc library and the receiver to deal with it properly. Also notice in rpc_select_func() that the buffer "buf" is passed (where NULL was passed previously in rpcexamp). This buffer space is used by the rpc library to write received data into. It must be big enough to accommodate ALL of the expected array data. In rpcexamp the buffer is 1024 in size which makes it plenty large for the 7 integers and 8 characters that are sent in array form.

The intrinsic rpc_table

The intrinsic rpc_table is a table of special calls that are always available to the developer and serves essentially the same function that libc does. The key purpose is to provide the embedded vision system the ability to share resources with the host computer that it doesn't have (e.g. console, display, hard drive, etc) It is intended to be augmented with calls the developer deems useful. Currently, there are calls for displaying images, saving images in tiff format and printing text. Other capabilities such as opening, reading and writing files are good examples of useful additions. The intrinsic rpc_table is in rpcintri.c.

Intrinsic procedures called from the vision system

printf(char *format, ...)

We've seen printf() used in hworld and rpcexamp. It behaves identically to standard libc printf() and causes the desired text to be printed on the host's console that the application is running from.

```
void disp_8image(char *image, int xws, int yws,
      int x_offset, int y_offset,
      int gx, int gy, int words_per_row, int format)
```

If a display window does not exist, this causes an X window to pop up on the host's X server (or remote X server pointed to by the host's DISPLAY environment variable.) If a display window does exist, it simply overwrites the current display. The arguments are as follows:

```
image:         Array containing the image.
xws:           Size of the x dimension X window
```

```
yws:           Size of the y dimension X window
x_offset:      x offset into the X window to display the image (image can be
               smaller than the X window)
y_offset:      y offset into the window to display the image (image can be
               smaller than the X window)
gx:            width of image (x dimension)
gy:            height of image (y dimension)
words_per_row: number of pixel words per row in the image
format:        One of either:
               AVG_GREYSCALE:  Takes RGB data in "image" and displays as
               grayscale
               LSBYTE:  Takes least significant 8 bits (remember, a 'C40 char
               is 32 bits) and displays as grayscale.
```

void write_8image(char *image, int gx, int gy, int words_per_row, int format, char *filename)

Writes `image` to the host's file system in tiff image format. The arguments are as follows:

```
image:         Array containing the image.
gx:            width of image (x dimension)
gy:            height of image (y dimension)
words_per_row: number of pixel words per row in the image
format:        One of either:
AVG_GREYSCALE: Takes RGB data in "image" and saves as
               grayscale.
LSBYTE:        Takes least significant 8 bits (remember, a 'C40 chars
               32 bits) and saves as grayscale
filename:      Name of saved image file.
```

void crosshair(int x0, int y0, int size, int color)

Displays a square crosshair on the current X window. If a crosshair already exists, the existing crosshair is moved to the newly specified location. The arguments are as follows:

```
x0:            x dimension location in X window
y0:            y dimension location in X window
size:          width and height of crosshair in pixels
color:         color in grayscale: 0 is black; 255 is white
```

Intrinsic procedures called from the host

These are available for the host program to call.

void get_8image(int xg, int yg, int gx, int gy)

This causes an rpc request to the vision system that then calls `disp_8image()`} of the current unprocessed video data. The arguments are as follows:

```
xg:            x dimension offset into the 480x512-video image.
yg:            y dimension offset into the 480x512-video image.
gx:            x dimension size of image (width)
yg:            y dimension size of image (height)
```

`void get_write_8image(int xg, int yg, int gx, int gy)`

This causes an rpc request to the vision system that then calls `write_8image` with the current unprocessed video data. The arguments are the same as `get_8image()`.

Compiling Sensus 700 code

The Sensus 700 uses the Texas Instruments TMS320C44 digital signal processor (DSP). Which means that we compile the Sensus 700 code with a cross compiler. Unfortunately, the compiler only runs under DOS.

Linux has support for dosemu, which is a DOS emulator. This should be installed on the hard-drive that came with the Sensus 700. To invoke, type dos at any robot console, or xdos if you are connected to an X server. Dosemu comes up in drive E, which is the DOS partition. (the DOS partition can be accessed in Linux in the /dos directory) When you want to exit dosemu, type exitemu and it will return you to Linux.

Change directories into \sensus7h\. Notice that this directory has many of the same directories that the ~/sensus7h/host directory does in Linux. This is where the corresponding vision system code resides for applications using the Sensus 700. Change into the rpcexamp directory and type make. It should compile with no errors. New Sensus 700 applications can be put in the \sensus7h\ directory and the corresponding make.bat can be created. It is recommended that \sensus7h\vis\rpcexamp\make.bat serve as a good example. Simply copy it into the newly created application directory and replace occurrences of rpcexamp with the name of the new application.

Also, as a convention it is nice to put the header file that contains the rpc_table in the DOS partition and make a soft link to it from the Linux partition. For example, in Linux type:

```
ls -l rpcexamp.h
```

while in the /root/sensus7h/host/rpcexamp directory. This will indicate that it is a soft link. To create a soft link, for example, rpcexamp.h type:

```
ln -s ../../vis/rpcexamp/rpcexamp.h
```

while in the /root/sensus7h/host/rpcexamp directory. For more information consult the man pages of ln.

Sensus 700 video

The following are video-related functions:

vram

On the frame grabber module in the Sensus 700 is a 1M-byte video RAM where captured frames are stored. It is arranged as a 512 by 512 array of 4-byte unsigned integers. A call to capture_single() puts a frame of data in this area of memory. VRAM is a global array that is declared when you include cfg44.h} and it is declared as follows:

```
extern unsigned int *vram;
```

For NTSC cameras, the first 32 rows of vram contain garbage. It is useful to declare a modified version of vram as follows:

```
unsigned int *mvram=(unsigned int *)(vram + 32*512);
```

Hence for NTSC cameras, the resulting image is 480 rows by 512 columns. In mvram (described above) this is stored beginning with the upper left of the image and working its way across the first scan line, which ends at mvram + 511. mvram + 512 begins the second scan line and so on, down the image.

The ints in vram can be in either grayscale or RGB. The default is RGB and it is stored as follows for each 4-byte pixel:

Byte3	Byte2	Byte1	Byte0
Empty	Blue	Green	Red

Consult the HECFG44 manual if you want to change pixel and capture formats.

capture_single()

Grabs a single frame of video data and puts it in `vram`}, a global array that is declared when you include `cfg44.h`}. For NTSC cameras, the image is 480 rows by 512 columns.

`video_init()`

This MUST be called before calling `capture_single()`}.

Miscellaneous notes

The TMS320C44

To optimize execution speeds, the ‘C44 DSP cannot address memory in pieces smaller than 32 bits. This results in chars, shorts, ints, longs, and floats ALL being 32 bits in length. The only exception are doubles which are 64 bits. This makes writing code for the ‘C44 quite tricky if you don’t keep this in mind.

CHAPTER 8 - PROGRAMMING REFERENCE

QUICK REFERENCE

Communication Commands

`N_InitializeClient` - initialize the communication
`N_ConnectRobot` - connect to a given robot
`N_DisconnectRobot` - disconnect from a given robot
`N_SetTimer` - set the timeout period of the robot
`N_GetTimer` - get the timeout period of the robot
`N_Speak` - download a string to the speech card

Base Motion Setting Commands

`N_SetAxes` - sets the axes position, velocity, and acceleration
`N_SetIntegratedConfiguration` - sets the integrated configuration
`N_SetJoystick` - sets parameters for software joystick control

Base Motion Parameters Retrieving Commands

`N_GetAxes` - gets the current axes position, velocity and acceleration
`N_GetIntegratedConfiguration` - gets the integrated configuration

Lift Mechanism Motion Setting Commands

`N_SetLift` - sets the motion parameters for the Lift Mechanism
`N_DeployLift` - deploys the Lift Mechanism
`N_RetractLift` - retracts the Lift Mechanism
`N_ZeroLift`- zeroes the Lift Mechanism

Lift Mechanism Retrieving commands

`N_GetLift` - gets configuration information about the Lift Mechanism

Sensing Parameters Setting Commands

`N_SetLaserConfiguration` - sets the laser configuration
`N_SetSonarConfiguration` - sets the sonar configuration

Sensing Parameters Retrieving Commands

`N_GetBattery` - gets the battery data for the given robot
`N_GetBumper` - gets the bumper data for the given robot
`N_GetCompass` - gets the compass data for the given robot
`N_GetInfrared` - gets the infrared data for the given robot
`N_GetLaser` - gets the laser data for the given robot
`N_GetS550` - gets the Sensus 550 laser data for the given robot
`N_GetSonar` - gets the sonar data for the given robot
`N_GetSonarConfiguration` - gets the sonar configuration
`N_GetState` - gets the current state for the given robot
`N_GetRobotState` - get the `N_RobotState` Structure

N_CONNECTROBOT

NAME

N_ConnectRobot

PURPOSE

To connect to a given robot.

SYNTAX

```
int N_ConnectRobot(long RobotID);
```

ARGUMENTS

`long RobotID` - robot identification number.

RETURNED VALUE

`N_NO_ERROR` - success;
`N_UNINITIALIZED` - N_InitializeClient has not yet been called
`N_ROBOT_NOT_FOUND` - the given robot ID was not registered with the scheduler.
`N_CONNECTION_FAILED` - connection failed;
`N_OUT_OF_MEMORY` - could not allocate data structures.

UPDATED GLOBALS

`N_RobotState`

DESCRIPTION

This function requests a connection to the robot with the specified `RobotID`. Such a connection is needed before the program can send any commands to the robot.

EXAMPLES

`N_Connect_Robot.c`

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

#define SCHEDULER_HOSTNAME “fatboy”
#define SCHEDULER_PORT 7073

int main()
{
  N_InitializeClient(SCHEDULER_HOSTNAME, SCHEDULER_PORT);
  switch (N_ConnectRobot(1))
  {
  case N_NO_ERROR:
    printf(“Successfully connected to Robot\n”);
    break;
  case N_ROBOT_NOT_FOUND:
    printf(“Robot not found\n”);
    break;
  case N_CONNECTION_FAILED:
    printf(“Connection failed\n”);
    break;
```

```
  }
N_DisconnectRobot(1);
exit(0);
}
```

SEE ALSO

N_DisconnectRobot

N_DISCONNECTROBOT

NAME

N_DisconnectRobot

PURPOSE

To disconnect from a given robot.

SYNTAX

```
int N_DisconnectRobot(long RobotID);
```

ARGUMENTS

`long RobotID` - robot identification number.

RETURNED VALUE

`N_NO_ERROR` - success.

`N_CONNECTION_FAILED` - the connection to this robot no longer exists.
`N_ROBOT_NOT_FOUND` - robot not found.

UPDATED GLOBALS

None

DESCRIPTION

This function requests a close to the connection with the robot with the specified `RobotID`.

EXAMPLES

`N_Disconnect_Robot.c`

```
#include <stdio.h>
#include “Nclient.h”

#define SCHEDULER_HOSTNAME “fatboy”
#define SCHEDULER_PORT 7073

int main()
{
  N_InitializeClient(SCHEDULER_HOSTNAME, SCHEDULER_PORT);
  N_ConnectRobot(1);
  switch (N_DisconnectRobot(1))
  {
  case N_NO_ERROR:
    printf(“Successfully disconnected from Robot\n”);
    break;
  case N_CONNECTION_FAILED:
    printf("Connection was lost\n");
    break;
  case N_ROBOT_NOT_FOUND:
    printf(“Robot not found\n”);
    break;
  }
```

```
exit(0);
}
```

SEE ALSO

N_ConnectRobot

N_DEPLOYLIFT

NAME

N_DeployLift

PURPOSE

To deploy the Lift Mechanism and permit control of its axes.

SYNTAX

```
int N_DeployLift(long RobotID);
```

ARGUMENTS

`long RobotID` - robot identification number.

RETURNED VALUE

`N_NO_ERROR` - success.
`N_ROBOT_NOT_FOUND` - robot not found.
`N_UNKNOWN_ERROR` - the motion was not able to execute.

UPDATED GLOBALS

`N_RobotState`

DESCRIPTION

This function initiates the deployment of the Lift Mechanism which may reside inside the robot in a retracted state. In general, before you attempt to move the either the Lift or the Grip axes a call to `N_DeployLift()` should be made to ensure that the mechanism is deployed. If it is not deployed, an error will be returned when a movement is attempted, but no movement will result.

EXAMPLES

`N_DeployLift.c`

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

#define SCHEDULER_HOSTNAME “fatboy”
#define SCHEDULER_PORT 7073


int main()
{
  struct N_RobotState *state;
  struct N_LiftController *lcont;


  N_InitializeClient(SCHEDULER_HOSTNAME, SCHEDULER_PORT);
  N_ConnectRobot(1);


  /* zero lift -- but only if necessary */
  N_ZeroLift(1, FALSE);
  N_DeployLift(1);
```

```
  /* wait for lift to deploy */
  do
  {
     state = N_GetRobotState(1);
     lcont=&(state->LiftController);
  }
  while (lcont->InProgress);

  N_DisconnectRobot(1);
  exit(0);
}
```

SEE ALSO

N_RetractLift

N_GETAXES

NAME

N_GetAxes

PURPOSE

To fill the N_RobotState Structure axes encoder data from the robot sensors.

SYNTAX

```
int N_GetAxes(long RobotID);
```

ARGUMENTS

long RobotID - robot identification number.

RETURNED VALUE

N_NO_ERROR - success.
N_ROBOT_NOT_FOUND - robot not found.
N_CONNECTION_FAILED - the socket was disconnected since the last client call.
N_UNKNOWN_ERROR - Nrobot failed to retrieve the requested information.

UPDATED GLOBALS

N_RobotState

DESCRIPTION

This function gets the specified robot's configuration for all of the robot's axes and updates the N_RobotState Structure.

```
struct N_AxisSet
{
  BOOL Global;
  unsigned char Status;
  N_CONST unsigned int AxisCount;
  struct N_Axis Axis[N_MAX_AXIS_COUNT];
};
```

- Global: Only used by XR4000 series robots. When this parameter is set to TRUE, it puts the mobile base axes in "global" or "world" mode after which control for the mobile base is with respect to the fixed global reference frame. The global reference frame is set when the robot is powered up and "zeroes" or when N_SetIntegratedConfiguration() is called. While in global mode (Global=TRUE), movement of the rotation axis will not change the direction of the x or y axis movement. For example, simultaneous movement of the Rotation and Y axes will cause a pirouette (motion in a straight line while spinning.)

  When this parameter is FALSE, the mobile base axes are controlled with respect to a "local" or "joint" reference frame. For example, simultaneous movement of the Rotation and Y axes will cause the robot to move in a circle -- that is, the robot will constantly move forward, but rotation causes the Y direction to change, producing a circle.

- Status: one of the following values:

  N_AXES_READY: This status indicates that the axes are available for movement.
  N_JOYSTICK_IN_USE: This status indicates that the base is being controlled via joystick.
  N_ESTOP_DOWN: This status indicates that one or more of the emergency stop buttons is depressed,

preventing the robot from moving.
N_MOTION_ERROR: The last executed motion failed.

```
struct N_Axis
{
  BOOL DataActive;
  BOOL TimeStampActive;
  BOOL Update;
  unsigned long TimeStamp;
  char Mode;
  long DesiredPosition;
  long DesiredSpeed;
  long Acceleration;
  long TrajectoryPosition;
  long TrajectoryVelocity;
  long ActualPosition;
  long ActualVelocity;
  BOOL InProgress;
  long TrajectoryVelocity;
};
```

- DataActive: A value for this parameter causes the values in this structure to be updated -- namely Mode, DesiredPosition, DesiredSpeed, Accleration, TrajectoryPosition, TrajectoryVelocity, ActualPosition, ActualVelocity, InProgress, and TrajectoryVelocity.
- TimeStampActive: A TRUE value for this parameter causes the TimeStamp parameter to be updated.
- Update: A TRUE value for this parameter causes the input values (DesiredSpeed, DesiredPosition, and Acceleration) to be loaded into the set of working values for this axis when a call to N_SetAxes() is made. The Update parameter allows one or more axes to be loaded with new input values simultaneously.
- TimeStamp: the time value in milliseconds that the axis values were measured.
- Mode: One of the following possible modes:
  N_AXIS_POSITION_RELATIVE: Specifies that the axis move relative to the current position.
  N_AXIS_POSITION_ABSOLUTE: Specifies that the axis move to an absolute position with respect to the absolute zero location of the axis.
  N_AXIS_VELOCITY: Specifies that the axis move with a constant velocity.
  N_AXIS_STOP: Specifies that the axis decelerate to zero velocity.
- DesiredPosition: Specifies the desired endpoint position of the axis. The units are in millimeters for translational axes and milliradians for rotational axes. When Mode is set to N_AXIS_POSITION_RELATIVE or N_AXIS_POSITION_ABSOLUTE this specifies the endpoint position relative to the current position or the absolute position, respectively. This parameter is not used when Mode is set to N_AXIS_VELOCITY. **This parameter can be positive or negative.**
- DesiredSpeed: This specifies the desired speed at which to move to the DesiredPosition, or the constant speed at which to move if Mode is set to N_AXIS_VELOCITY. The units are in millimeters / second for translational axes and milliradians / second for rotational axes.
- Acceleration: Constrains the rate at which the speed can change. The units are in millimeters /

second^2 for translational axes and milliradians / second^2 for rotational axes. **This parameter can only be positive.**

- `TrajectoryPosition`: Provides the current position of the trajectory generator.
- `TrajectoryVelocity`: Provides the current velocity of the trajectory generator.
- `ActualPosition`: Provides the actual position of the axis. The value of this field is based on the setting of the Global field in N_AxisSet. If in Joint mode (`Global=FALSE`), this field provides the joint position of this axis. If in `Global` mode (`Global=TRUE`), this field provides the position of the axis in a global reference frame. In `Global` mode, this value is equivalent to the corresponding field in the `N_Integrator` structure.
- `ActualVelocity`: Provides the actual measured velocity of the axis. This value, like ActualPosition is based on the setting of the Global field in `N_AxisSet`. In Global mode (`Global=TRUE`), this field provides a value that is with respect to a global reference frame. In Joint mode, (`Global=FALSE`) this field provides a value that is with respect to the local reference frame.
- `InProgress`: Provides a boolean value that informs the user that an axis is currently moving.

The interpretation and range for the axes and their values according to the robot type are given below:

	Nomad 200	**XR4000**
Axis 0 is	Translation	X
Axis 1 is	Steering	Y
Axis 2 is	Turret	Theta
Axis 0 velocity range	[-609mm/s, 609mm/s]	[-1500mm/s, 1500mm/s]
Axis 1 velocity range	[-785mRad/s, 785mRad/s]	[-1500mm/s, 1500mm/s]
Axis 2 velocity range	[-785mRad/s, 785mRad/s]	[-5000mRad/s, 5000mRad/s]
Axis 0 acceleration range	[0mm/s^2, 762mm/s^2]	[0mm/s^2, 1500mm/s^2]
Axis 1 acceleration range	[0mRad/s^2, 872mRad/s^2]	[0mm/s^2, 1500mm/s^2]
Axis 2 acceleration range	[0mRad/s^2, 872mRad/s^2]	[0mRad/s^2, 5000mRad/s^2]

EXAMPLES

`N_GetAxes.c`

```
#include <stdio.h>
#include “Nclient.h”

#define SCHEDULER_HOSTNAME “fatboy”
#define SCHEDULER_PORT 7073

int main()
{
  struct N_RobotState *state;
  struct N_Axis *axis;
```

```
  N_InitializeClient(SCHEDULER_HOSTNAME, SCHEDULER_PORT);
  N_ConnectRobot(1);
  state = N_GetRobotState(1);
  axis = &(state->AxisSet.Axis[N_XTRANSLATION]);

  while(1)
  {
    N_GetAxes(1);
    printf("Position %d mm\n", axis->CurrentPosition);
  }
  N_DisconnectRobot(1);
  exit(0);
}
```

SEE ALSO

N_SetAxes, N_GetState

N_GETBATTERY

NAME

N_GetBattery

PURPOSE

To fill the N_RobotState Structure with battery data.

SYNTAX

```
int N_GetBattery(long RobotID);
```

ARGUMENTS

long RobotID - robot identification number.

RETURNED VALUE

N_NO_ERROR - success.
N_ROBOT_NOT_FOUND - robot not found.
N_CONNECTION_FAILED - the socket was disconnected since the last client call.

UPDATED GLOBALS

N_RobotState

DESCRIPTION

The BatterySet structure simply holds an array of pointers to Battery structures. Battery data is available for the XR4000 only.

```
struct N_BatterySet
{
  struct N_Battery Battery[N_MAX_BATTERY_COUNT];
  BOOL DataActive;
};
```

The mapping between the Battery array position and the physical batteries is given below:

```
struct N_Battery
{
   long Voltage;
};
```

- Voltage: Provides the current battery voltage in millivolts.

EXAMPLES

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

#define SCHEDULER_HOSTNAME "fatboy"
#define SCHEDULER_PORT 7073

int main()
{
  struct N_RobotState *state;
```

```
  struct N_Battery *battery;

  N_InitializeClient(SCHEDULER_HOSTNAME, SCHEDULER_PORT);
  N_ConnectRobot(1);

  state = N_GetRobotState(1);
  battery = &(state->BatterySet.Battery[0]);

  N_GetBattery(1);
  printf("Battery Voltage %d\n", battery->Voltage);

  N_DisconnectRobot(1);
  exit(0);
}
```

SEE ALSO

N_GetState

N_GETBUMPER

NAME

N_GetBumper

PURPOSE

To fill the N_BumperSet field in N_RobotState with data from the tactile sensors.

SYNTAX

```
int N_GetBumper(long RobotID);
```

ARGUMENTS

long RobotID - robot identification number.

RETURNED VALUE

N_NO_ERROR - success;
N_ROBOT_NOT_FOUND - robot not found;
N_CONNECTION_FAILED - the socket was disconnected since the last client call.

UPDATED GLOBALS

N_RobotState

DESCRIPTION

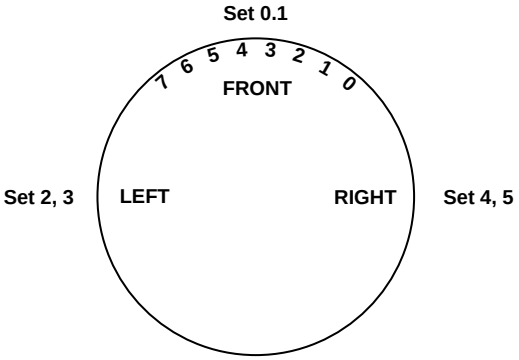
This command retrieves the bumper data from the tactile sensors and updates the N_RobotState Structure.

The bumper controller structure holds all the configuration data that is valid for all the bumper sets on the robot. Bumper sets are groups of tactile sensors that act together. The bumper controller structure has an array of N_BumperSet structures, each describing one particular set.

Forty-eight bi-level sensing elements surround the top and bottom perimeters of the XR4000, providing both the exact location of contact as well as its contact force (none, low, and high). A light touch on each bi-level tactile switch will result in a green LED, while a firm press will light the red LED. Rugged energy absorbing rubber molding protects the top and bottom perimeters of sensing elements.

The Nomad 200 has two sets of 10 bumpers, going counterclockwise from 0 to 9, with 0 in front.

The XR4000 has 3 doors that go counterclockwise and there are two (sonar / infrared / bumper) sets per door for a total of 6 sets:



Set 0 = top set on front door (sensor #0–7)
Set 1 = bottom set on front door (sensor #0–7)
Set 2 = top set on left door (sensor #0–7)
Set 3 = bottom set on left door (sensor #0–7)
Set 4 = top set on right door (sensor #0–7)
Set 5 = bottom set on right door (sensor #0–7)

```
struct N_BumperController
{
  N_CONST unsigned int BumperSetCount;
  struct N_BumperSet BumperSet[N_MAX_BUMPER_SET_COUNT];
};
```

The configuration data used globally by all the bumper sets on the robot are:

- `BumperSetCount`: the number of bumper sets in the controller.

The `N_BumperSet` structure is defined as follows:

```
struct N_BumperSet
{
  BOOL DataActive;
  BOOL TimeStampActive;
  N_CONST unsigned int BumperCount;
  struct N_Bumper Bumper[N_MAX_BUMPER_COUNT];
};
```

It contains an array `Bumper` of bumper structures plus a number of configuration parameters. The configuration parameters are:

- The `DataActive` flag: `TRUE` if bumper data for this set is to be updated.
- The `TimeStampActive` flag: `TRUE` if the time (time of acquisition of the tactile range) is to be updated for this set.
- `BumperCount`: the number of bumpers in the set.

The XR4000 also has 4 “door bumpers” for each door, and these are stored at the end of the top set of each door (indexes 8 through 11). For instance, bumper set 2 on the XR4000 is the top set of bumpers on the second door, so the readings for the door bumpers on the second door will be stored in bumper set 2. So there are actually 12 readings in bumper set 2. This is why `N_XR4000_BUMPER_COUNT` is 12 instead of 10. Door bumper readings are either `N_BUMPER_NONE` or `N_BUMPER_LOW`.

The `N_Bumper` structure is the lowest level in the general description of the tactile system.

```
struct N_Bumper
{
  char Reading;
  unsigned long TimeStamp;
};
```

The values defined for each bumper are:

- `Reading`: one per bumper and it will always be one of the three bumper value constants defined in `Nclient.h`. For the Nomad 200, bumpers have only the `N_BUMPER_NONE` or `N_BUMPER_LOW` value. For the XR4000, they can also be `N_BUMPER_HIGH` (for a hard hit).
- `TimeStamp`: the time of the tactile event (collision).

EXAMPLES

N_GetBumper.c

```
#include <stdio.h>
#include “Nclient.h”

#define SCHEDULER_HOSTNAME “fatboy”
#define SCHEDULER_PORT 7073

int main()
{
  struct N_RobotState *state;
  struct N_Bumper *bumper;
  short i;

  N_InitializeClient(SCHEDULER_HOSTNAME, SCHEDULER_PORT);
  N_ConnectRobot(1);

  state = N_GetRobotState(1);
  bumper = state->BumperController.BumperSet[0].Bumper;

  while(1)
  {
    N_GetBumper(1);
    printf(“BumperSet 0: “);

    for (i = 0; i < 8; i++)
    {
      printf(“%d “, bumper[i].Reading);
    }
    printf(“\n”);

  N_DisconnectRobot(1);
  exit(0);
  }
```

SEE ALSO

N_GetState

N_GETCOMPASS

NAME

N_GetCompass

PURPOSE

To fill the N_RobotState Structure with heading data from the compass.

SYNTAX

```
int N_GetCompass(long RobotID);
```

ARGUMENTS

long RobotID - robot identification number.

RETURNED VALUE

N_NO_ERROR - success.
N_ROBOT_NOT_FOUND - robot not found.
N_CONNECTION_FAILED - the socket was disconnected since the last client call.

UPDATED GLOBALS

N_RobotState

DESCRIPTION

This command retrieves the heading data from the compass and updates the N_Compass field in the N_RobotState Structure.

The Compass structure is defined as follows:

```
struct N_Compass
{
  long Reading;
  unsigned long TimeStamp;
  BOOL DataActive;
  BOOL TimeStampActive;
};
```

- The Reading field represents the heading of the turret with respect to the magnetic north in milliradians
- TimeStamp: the time of the acquisition of the reading
- The DataActive flag: TRUE if compass data for this set is to be updated
- The TimeStampActive flag: TRUE if the time (time of acquisition of the compass data) is to be updated for this set.

EXAMPLES

N_GetCompass.c

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

#define SCHEDULER_HOSTNAME "fatboy"
#define SCHEDULER_PORT 7073

int main()
```

```
{
  struct N_RobotState *state;
  struct N_Compass *compass;
  short i;

  N_InitializeClient(SCHEDULER_HOSTNAME, SCHEDULER_PORT);
  N_ConnectRobot(1);

  state = N_GetRobotState(1);
  compass = &(state->Compass);

  while(1)
  {
    N_GetCompass(1);
    printf("Compass Direction %d \n", compass->Reading);
  }
  N_DisconnectRobot(1);
  exit(0);
}
```

SEE ALSO

N_GetState

N_GETINFRARED

NAME

N_GetInfrared

PURPOSE

To fill the N_RobotState Structure with range data form the infrared proximity sensors.

SYNTAX

```
int N_GetInfrared(long RobotID);
```

ARGUMENTS

long RobotID - robot identification number.

RETURNED VALUE

N_NO_ERROR - success.
N_ROBOT_NOT_FOUND - robot not found.
N_CONNECTION_FAILED - the socket was disconnected since the last client call.

UPDATED GLOBALS

N_RobotState

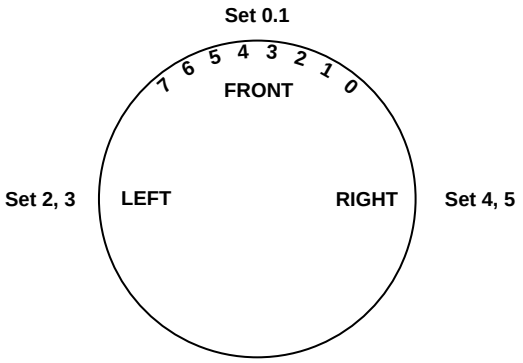
DESCRIPTION

This command retrieves range data from the infrared proximity sensors and updates the N_RobotState Structure.

The infrared controller structure holds all the configuration data that is valid for all the infrared sets on the robot. Infrared sets are groups of infrared sensors that act together. The infrared controller structure has an array of N_InfraredSet structures, each describing one particular set.

The Nomad 200 has one set of 16 infrared, going counterclockwise from 0 to 15, with 0 in front.

The XR4000 has 3 doors that go counterclockwise and there are two (sonar / infrared / bumper) sets per door for a total of six sets:



Set 0 = top set on front door (sensor #0–7)
Set 1 = bottom set on front door (sensor #0–7)
Set 2 = top set on left door (sensor #0–7)
Set 3 = bottom set on left door (sensor #0–7)
Set 4 = top set on right door (sensor #0–7)
Set 5 = bottom set on right door (sensor #0–7)

```
struct N_InfraredController
{
  BOOL InfraredPaused;
  N_CONST unsigned int InfraredSetCount;
  struct N_InfraredSet InfraredSet[N_MAX_INFRARED_SET_COUNT];
};
```

The configuration data used globally by all the infrared sets on the robot are:

- `InfraredPaused`: When set to `TRUE`, all the infrared sensors will be stopped.
- `InfraredSetCount`: The number of infrared sets in the controller.

The `N_InfraredSet` structure is defined as follows:

```
struct N_InfraredSet
{
  BOOL DataActive;
  BOOL TimeStampActive;
  N_CONST unsigned int InfraredCount;
  struct N_Infrared Infrared[N_MAX_INFRARED_COUNT];
};
```

It contains an array `Infrared` of infrared structures (one per infrared transducer in the set), plus a number of configuration parameters. The configuration parameters are:

- The `DataActive` flag: `TRUE` if infrared data for this set is to be updated -- namely the data in the Infrared array.
- The `TimeStampActive` flag: `TRUE` if the TimeStamps in the Infrared array is to be updated for this set.
- `InfraredCount`: the number of infrared proximity sensors in the set.

The `N_Infrared` structure is defined as:

```
struct N_Infrared
{
  long Reading;
  unsigned long TimeStamp;
};
```

The values defined for each infrared are:

- `Reading`: a value from 0 to 255 representing the amount of reflected infrared energy returned from a closeby object. A value of 0 represents no energy reflected (far object) while a value of 255 represents maximum energy reflected (close object). Should a distance value be needed, the user should build a calibration table by measuring the energy returned by a material sample (representative of what can be found in the environment), at various distances.
- `TimeStamp`: the time of the acquisition of the raw value.

EXAMPLES

`N_GetInfrared.c`

```
#include <stdio.h>
#include “Nclient.h”
#define SCHEDULER_HOSTNAME “fatboy”
#define SCHEDULER_PORT 7073
int main()
```

```
{
  struct N_RobotState *state;
  struct N_Infrared *infrared;
  short i;

  N_InitializeClient(SCHEDULER_HOSTNAME, SCHEDULER_PORT);
  N_ConnectRobot(1);

  state = N_GetRobotState(1);
  infrared = state->InfraredController.InfraredSet[0].Infrared;

  while(1)
  {
    N_GetInfrared(1);
    printf("InfraredSet 0: ");
    for (i = 0; i < 8; i++)
      printf("%d ", infrared[i].Reading);
    printf("\n");
  }

  N_DisconnectRobot(1);
  exit(0);
}
```

SEE ALSO

N_GetState

N_GETINTEGRATEDCONFIGURATION

NAME

N_GetIntegratedConfiguration

PURPOSE

To fill the N_RobotState Structure with integrated configuration data.

SYNTAX

```
int N_GetIntegratedConfiguration(long RobotID);
```

ARGUMENTS

long RobotID - robot identification number.

RETURNED VALUE

N_NO_ERROR - success.
N_ROBOT_NOT_FOUND - robot not found.
N_CONNECTION_FAILED - the socket was disconnected since the last client call.

UPDATED GLOBALS

N_RobotState

DESCRIPTION

This command retrieves the integrated configuration from the robot and updates the N_RobotState Structure.

The N_Integrator structure contains the geometric configuration of the robot with respect to a globally fixed coordinate frame. For example, the XR4000 has a configuration consisting of its x, y location and its rotation angle since it was powered on or since the last call to N_SetIntegratedConfiguration.

```
struct N_Integrator
{
  BOOL DataActive;
  BOOL TimeStampActive;
  unsigned long TimeStamp;
  long x;
  long y;
  long Steering;
  long Rotation;
}
```

EXAMPLES

N_GetIntegratedConfiguration.c

```
#include <stdio.h>
#include “Nclient.h”

#define SCHEDULER_HOSTNAME “fatboy”
#define SCHEDULER_PORT 7073

int main()
{
```

```
  struct N_RobotState *state;
  struct N_Configuration *integrated_configuration;

  N_InitializeClient(SCHEDULER_HOSTNAME, SCHEDULER_PORT);
  N_ConnectRobot(1);

  state = N_GetRobotState(1);
  integrated_configuration =&state->Integrator;

  while (1)
  {
    N_GetIntegratedConfiguration(1);
    printf("Integrated Configuration: x %d y %d Steering %d
    Rotation %d\n",
       integrated_configuration->x,
       integrated_configuration->y,
       integrated_configuration->Steering,
       integrated_configuration->Rotation);
  }
  N_DisconnectRobot(1);
  exit(0);
}
```

SEE ALSO

N_SetIntegratedConfiguration

N_GETLIFT

NAME

N_GetLift

PURPOSE

To retrieve configuration information about the Lift Mechanism's axes.

SYNTAX

```
int N_GetLift(long RobotID);
```

ARGUMENTS

`long RobotID` - robot identification number.

RETURNED VALUE

`N_NO_ERROR` - success.
`N_ROBOT_NOT_FOUND` - robot not found.
`N_CONNECTION_FAILED` - the socket was disconnected since the last client command

UPDATED GLOBALS

None

DESCRIPTION

This function retrieves information about the Lift and Grip axes of the Lift Mechanism and stores the results in the `LiftController` structure in `N_RobotState`.

The `LiftController` structure has the following definition:

```
struct N_LiftController
{
  BOOL Deployed;
  BOOL InProgress;
  struct N_LiftAxis Axis[N_LIFT_AXIS_COUNT];
};
```

The fields are described as follows:

- `Deployed`: provides information on whether the Lift Mechanism is fully deployed or not. A TRUE value indicates that the Lift mechanism is fully deployed.
- `InProgress`: a `TRUE` value in this field indicates that the Lift Mechanism is currently being retracted, deployed, or zeroed (i.e. due to a call to `N_DeployLift()`, `N_RetractLift()`, or `N_ZeroLift()`). While this field is set, no motion commands to the Lift mechanism are accepted.
- `Axis`: an array that contains the motion parameters for the two controllable axes: the Lift and Grip axes. The two possible array indices are respectively `N_LIFT` and `N_GRIP`.

The axis parameters are contained in the `N_LiftAxis` structure, which is defined as:

```
struct N_LiftAxis
{
  BOOL DataActive;
  BOOL TimeStampActive;
  BOOL Update;
  unsigned long TimeStamp;
```

```
  char Mode;
  long Status;
  long DesiredPosition;
  long DesiredVelocity;
  long Acceleration;
  long MaxMotor;
  long Position;
  long Velocity;
};
```

- `DataActive`: a `TRUE` value for this field causes the parameters of this structure to be updated with each call to `N_GetAxes()` -- namely `Mode`, `Status`, `DesiredPosition`, `DesiredVelocity`, `Acceleration`, `MaxMotor`, `Position` and `Velocity`.
- `TimeStampActive`: a TRUE value for this field causes the TimeStamp field to be updated with each call to `N_GetAxes()`.
- `Update`: a `TRUE` value for this field causes the motion parameters to be loaded into the motor controller for execution upon calling `N_SetAxes()`.
- `TimeStamp`: contains the time in milliseconds that the motion parameters were obtained.
- `Mode`: contains the movement mode for the axis. It can be set to one of the following possible modes:

  `N_LIFT_POSITION_RELATIVE`: causes the axis to move to a position relative to the current position as specified by the `DesiredPosition` field.

  `N_LIFT_POSITION_ABSOLUTE`: causes the axis to move to an absolute position (see table above) as specified by the `DesiredPosition` field.

  N_LIFT_VELOCITY: causes the axis to move at a velocity specified by the DesiredVelocity field.
- `Status`: provides a bitmap of possible errors. One or more of these bits will be set if an error occurs during a movement. To test for a particular error, the value or result can be "ored" with the error bits described below:

  `N_LIFT_POS_LIMIT`: indicates the positive limit switch has been reached for the axis. This should never happen, as the software limits travel to never traverse the positive limit of neither the Grip nor Lift axes after the Lift Mechanism is zeroed.

  `N_LIFT_NEG_LIMIT`: indicates the negative limit switch has been reached for the axis. This should never happen for the Lift axis, but the Grip axis when fully closed depresses the gripper's negative limit switch.

  `N_LIFT_MOTION_ERROR`: indicates that the motion was unable to complete. This is often due to an obstacle in the desired path of the axis.

  `N_LIFT_ESTOP`: indicates that the motion is not possible because one or more of the emergency stop switches is depressed.

  `N_LIFT_CANNOT_MOVE`: indicates that the motion is not possible because the Lift Mechanism is not zeroed or not deployed.
- `DesiredPosition`: specifies the desired endpoint position of the axis in units of 0.1 mm.. The position is either relative to the current position or absolute with respect to the zero position as specified by the Mode field.
- `DesiredVelocity`: specifies the desired velocity at which the axis should move in units of 0.1 mm/s.
- `DesiredAcceleration`: specifies the desired acceleration at which the axis should move in units of 0.1 mm/s^2.

- `MaxMotor`: specifies the percentage of available torque to use for the axis. The possible values range between 0 and 100, where 100 specifies the default of 100% available torque. For example, this is useful to set for the gripper when picking up fragile objects.
- `Position`: provides the current absolute position of the axis in units of 0.1mm.
- `Velocity`: provides the current velocity of the axis in units of 0.1mm/s.

EXAMPLES

N_GetLift.c

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

#define SCHEDULER_HOSTNAME “fatboy”
#define SCHEDULER_PORT 7073

int main()
{
  struct N_RobotState *state;
  struct N_LiftController *lcont;

  N_InitializeClient(SCHEDULER_HOSTNAME, SCHEDULER_PORT);
  N_ConnectRobot(1);

  /* zero lift -- but only if necessary */
  N_ZeroLift(1, FALSE);
  /* Lift Mechanism must be deployed in order to move */
  N_DeployLift(1);
  /* wait for lift to deploy */
  do
  {
     state = N_GetLift(1);
    lcont=&(state->LiftController);
  }
  while (lcont->InProgress);

  /* set the gripper torque to 50% */
  lcont->Axis[N_GRIP].MaxMotor=50;
  /* move the gripper to the specified absolute position */
  lcont->Axis[N_GRIP].Mode=N_LIFT_POSITION_ABSOLUTE;
  lcont->Axis[N_GRIP].DesiredVelocity=300;              /* move at 30 mm/s */
  lcont->Axis[N_GRIP].DesiredAcceleration=300;  /* accelerate at 30 mm/s/s */
  lcont->Axis[N_GRIP].DesiredPosition=0;       /* move the gripper to zero */
  N_SetLift();
  lcont->Axis[N_GRIP].DataActive=TRUE;

  /* wait until finished */
  do
  {
     N_GetLift(1);
     printf(“Grip velocity %d\n”, lcont->Axis[N_GRIP].Velocity);
  }
  while (lcont->Axis[N_GRIP].Velocity != 0);

  N_DisconnectRobot(1);
  exit(0);
}
```

SEE ALSO

N_SetLift

N_GETS550

NAME

N_GetS550

PURPOSE

To fill the N_RobotState Structure with range data from the laser sensor (Sensus 550).

SYNTAX

```
int N_GetS550(long RobotID);
```

ARGUMENTS

long RobotID - robot identification number.

RETURNED VALUE

N_NO_ERROR - success.
N_INVALID_ARGUMENT - invalid requested points.
N_SENSOR_NOT_READY - the device has not yet initialized itself.
N_ROBOT_NOT_FOUND - robot not found.
N_CONNECTION_FAILED - the socket was disconnected since the last client call.

UPDATED GLOBALS

N_RobotState

DESCRIPTION

This command retrieves data from the laser sensor (Sensus 550) and puts it in the N_S550Set field of the N_RobotState Structure.

```
struct N_S550Set
{
  N_CONST unsigned int S550Count;
  struct N_S550 S550[N_MAX_S550_COUNT];
};
```

- S550Count: the number of Sensus 550 laser devices in the set

The N_S550 structure is defined as:.

```
struct N_S550
{
  N_CONST unsigned int TotalPoints;
  unsigned int RequestedPoints;

  unsigned long Readings[N_MAX_S550_POINTS];
  unsigned char StatusFlags[N_MAX_S550_POINTS];
  unsigned char SummaryFlags;

  unsigned long TimeStamp;
  BOOL DataActive;
  BOOL TimeStampActive;
};
```

- TotalPoints - The number of points in the Readings array. TotalPoints gets set upon initialization to either 180 or 361 points according to the model used and should not be modified.

- `RequestedPoints` - The number of measurements that the user desires per laser scan. This can be of a one of a set of possible values (9, 10, 15, 18, 30, 45, 90, 180, 361). If a value other than these values is requested, an `N_INVALID_ARGUMENT` result will be returned. Fewer parameters than 361 cause the sensor to return the request amount of measurements distributed evenly across the 180 degree viewing angle (e.g. for a `RequestedPoints` of 10, the measurements will be 18 degrees apart.)
- `Readings` - The set of sensor readings in millimeters with the 0th element in the array being the first, rightmost reading. That is, the sensor takes measurements in order from right to left with respect to the sensor.
- `StatusFlags` - Contains status states for each entry in the `Readings` array. Will eventually be used to indicate if a reading violated the safe or warning fields, and various other possible conditions. **Not used in 1.0.**
- `SummaryFlags` - Contains status flags applicable to the entire set of readings. e.g. will indicate if any of the readings violated the safe field. **Not used in 1.0.**
- `TimeStamp` - The time at which the most current laser scan took place.
- `DataActive` - If this is set to a TRUE value, this structure will be updated when calls to the client library are made.
- `TimestampActive` - If this is set to a TRUE value, the TimeStamp field will be updated when calls to the client library are made.

EXAMPLES

```
N_GetS550.c
#include <stdio.h>
#include “Nclient.h”

#define SCHEDULER_HOSTNAME “fatboy”
#define SCHEDULER_PORT 7073

int main()
{
  struct N_RobotState *state;
  struct N_S550 *S550;
  short i;

  N_InitializeClient(SCHEDULER_HOSTNAME, SCHEDULER_PORT);
  N_ConnectRobot(1);

  state = N_GetRobotState(1);
  S550 = &state->S550Set.S550[0];

  while (1)
  {
    N_GetS550(1, 0);

    printf(“Front Reading for S550 0: %d\n”, S550->Readings[89]);
  }
  N_DisconnectRobot(1);
  exit(0);
}
```

SEE ALSO

N_GetState

N_GETSONAR

NAME

N_GetSonar

PURPOSE

To fill the N_RobotState Structure with range data from the sonar proximity sensors.

SYNTAX

```
int N_GetSonar(long RobotID);
```

ARGUMENTS

long RobotID - robot identification number.

RETURNED VALUE

N_NO_ERROR - success.
N_ROBOT_NOT_FOUND - robot not found.
N_CONNECTION_FAILED - the socket was disconnected since the last client call.

UPDATED GLOBALS

N_RobotState

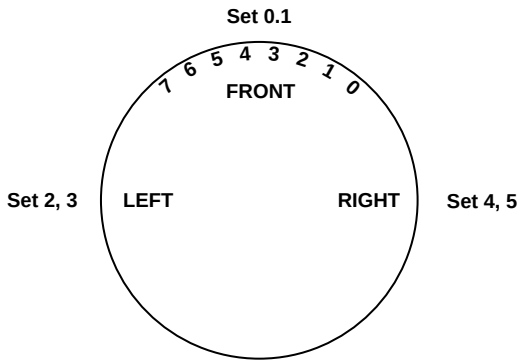
DESCRIPTION

This command retrieves range data from the sonar proximity sensors and stores them in the N_SonarController field of N_RobotState.

The sonar controller structure holds all the configuration data that is valid for all the sonar sets on the robot. Sonar sets are groups of sonar that act together. For instance, a firing order can be defined among the sonar of a given set. The sonar controller structure has an array of N_SonarSet structures, each describing one particular set.

The Nomad 200 has one set of 16 sonar, going counterclockwise from 0 to 15, with 0 in front.

The XR4000 has 3 doors that go counterclockwise and there are two (sonar / infrared / bumper) sets per door for a total of six sets:



Set 0 = top set on front door (sensor #0–7)
Set 1 = bottom set on front door (sensor #0–7)
Set 2 = top set on left door (sensor #0–7)
Set 3 = bottom set on left door (sensor #0–7)
Set 4 = top set on right door (sensor #0–7)
Set 5 = bottom set on right door (sensor #0–7)

```
struct N_SonarController
{
  N_CONST unsigned int SonarSetCount;
```

```
  struct N_SonarSet SonarSet[N_MAX_SONAR_SET_COUNT];
  BOOL SonarPaused;
};
```

The configuration data used globally by all the sonar sets on the robot are:

- `SonarSetCount`: the number of sonar sets in the controller.

The `N_SonarSet` structure is defined as follows:

```
struct N_SonarSet
{
  unsigned int FiringOrder[N_MAX_SONAR_COUNT + 1];
  long FiringDelay;
  long BlankingInterval;
  BOOL DataActive;
  BOOL TimeStampActive;
  N_CONST unsigned int SonarCount;
  struct N_Sonar Sonar[N_MAX_SONAR_COUNT];
};
```

It contains an array `Sonar` of sonar structures (one per sonar transducer in the set), plus a number of configuration parameters. The configuration parameters are:

- `FiringOrder`: an array of sonar indices terminated by `N_END_SONAR_FIRING_ORDER` if the length of the array is less than the `SonarCount`.
- `FiringDelay`: delay in milliseconds between two consecutive sonar firings.
- `BlankingInterval`: the time in milliseconds to wait after a sonar sensor has fired before the sensor begins to listen. **This is currently not implemented on the XR4000 Release 1.0.**
- `DataActive`: set to a TRUE value if the sonar data is to be updated for this set.
- `TimeStampActive`: set to a TRUE value if the time (time of acquisition of the sonar range) is to be updated for this set.
- `SonarCount`: number of sonars in this set.

The `N_Sonar` structure contains the sensor readings:

```
struct N_Sonar
{
  long Reading;
  unsigned long TimeStamp;
};
```

The values defined for each sonar are:

- `Reading`: the distance measurement of the sensor in millimeters.
- `TimeStamp`: the time that the measurement took place in milliseconds.

EXAMPLES

`N_GetSonar.c`

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


#define SCHEDULER_HOSTNAME "fatboy"
```

```
#define SCHEDULER_PORT 7073

int main()
{
  struct N_RobotState *state;
  struct N_Sonar *sonar;
  short i;

  N_InitializeClient(SCHEDULER_HOSTNAME, SCHEDULER_PORT);
  N_ConnectRobot(1);

  state = N_GetRobotState(1);
  sonar = state->SonarController.SonarSet[0].Sonar;

  while(1)
  {
    N_GetSonar(1);
    printf("SonarSet 0: ");
    for (i = 0; i < 8; i++)
      printf("%d ", sonar[i].Reading);
    printf("\n");
  }

  N_DisconnectRobot(1);
  exit(0);
}
```

SEE ALSO

N_GetSonarConfiguration, N_GetState

N_GETSONARCONFIGURATION

NAME

N_GetSonarConfiguration

PURPOSE

To fill the N_RobotState Structure with configuration data from the sonar proximity sensors.

SYNTAX

```
int N_GetSonarConfiguration(long RobotID);
```

ARGUMENTS

long RobotID - robot identification number.

RETURNED VALUE

N_NO_ERROR - success.
N_ROBOT_NOT_FOUND - robot not found.
N_CONNECTION_FAILED - the socket was closed since the last client call.

UPDATED GLOBALS

N_RobotState

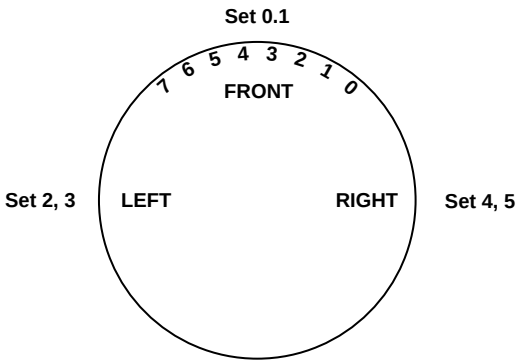
DESCRIPTION

This command retrieves the sonar configuration from the robot and updates the N_RobotState Structure.

The sonar controller structure holds all the configuration data that is valid for all the sonar sets on the robot. Sonar sets are groups of sonar that act together. For instance, a firing order can be defined among the sonar of a given set. The sonar controller structure has an array of N_SonarSet structures, each describing one particular set.

The Nomad 200 has one set of 16 sonar, going counterclockwise from 0 to 15, with 0 in front.

The XR4000 has 3 doors that go counterclockwise and there are two (sonar / infrared / bumper) sets per door for a total of six sets:



Set 0 = top set on front door (sensor #0–7)
Set 1 = bottom set on front door (sensor #0–7)
Set 2 = top set on left door (sensor #0–7)
Set 3 = bottom set on left door (sensor #0–7)
Set 4 = top set on right door (sensor #0–7)
Set 5 = bottom set on right door (sensor #0–7)

```
struct N_SonarController
{
  N_CONST unsigned int SonarSetCount;
```

```
  struct N_SonarSet SonarSet[N_MAX_SONAR_SET_COUNT];
  BOOL SonarPaused;
};
```

The configuration data used globally by all the sonar sets on the robot are:

- `SonarSetCount` -- the number of sonar sets in the controller.

The `N_SonarSet` structure is defined as follows:

```
struct N_SonarSet
{
  unsigned int FiringOrder[N_MAX_SONAR_COUNT + 1];
  long FiringDelay;
  long BlankingInterval;
  BOOL DataActive;
  BOOL TimeStampActive;
  N_CONST unsigned int SonarCount;
  struct N_Sonar Sonar[N_MAX_SONAR_COUNT];
};
```

It contains an array `Sonar` of sonar structures (one per sonar transducer in the set), plus a number of configuration parameters. The configuration parameters are:

- `FiringOrder`: an array of sonar indices terminated by `N_END_SONAR_FIRING_ORDER` if the length of the array is less than the `SonarCount`.
- `FiringDelay` -- delay in milliseconds between two consecutive sonar firings.
- `BlankingInterval` -- the time in milliseconds to wait after a sonar sensor has fired before the sensor begins to listen. **This is currently not implemented on the XR4000 Release 1.0.**
- `DataActive`: set to a TRUE value if the sonar data is to be updated for this set.
- `TimeStampActive`: set to a TRUE value if the time (time of acquisition of the sonar range) is to be updated for this set.
- `SonarCount`: number of sonars in this set.

The `N_Sonar` structure contains the sensor readings:

```
struct N_Sonar
{
  long Reading;
  unsigned long TimeStamp;
};
```

The values defined for each sonar are:

- `Reading`: the distance measurement of the sensor in millimeters.
- `TimeStamp`: the time that the measurement took place in milliseconds.

EXAMPLES

`N_GetSonarConfiguration.c`

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

#define SCHEDULER_HOSTNAME "fatboy"
#define SCHEDULER_PORT 7073
```

```
int main()
{
  struct N_RobotState *state;
  struct N_SonarSet *sonar_set;
  short i;

  N_InitializeClient(SCHEDULER_HOSTNAME, SCHEDULER_PORT);
  N_ConnectRobot(1);

  state = N_GetRobotState(1);
  sonar_set = &state->SonarController.SonarSet[0];

  N_GetSonarConfiguration(1);

  printf("Configuration of Sonar Set 0\n");
  printf("Data Active: %s Time Stamp Active: %s\n",
         sonar_set->DataActive ? "Yes" : "No",
         sonar_set->TimeStampActive ? "Yes" : "No");
  printf("Firing order: ");
  i = 0;
  while (sonar_set->FiringOrder[i]!=N_END_SONAR_FIRING_ORDER)
  {
    printf("%d ", sonar_set->FiringOrder[i]);
    i+=1;

    printf("\n");
  }
  N_DisconnectRobot(1);
  exit(0);
}
```

SEE ALSO

N_GetSonar, N_SetSonarConfiguration, N_GetState

N_GETSTATE

NAME

N_GetState

PURPOSE

To fill the N_RobotState Structure with data from the robot sensors.

SYNTAX

```
int N_GetState(long RobotID);
```

ARGUMENTS

long RobotID - robot identification number.

RETURNED VALUE

N_NO_ERROR - success.
N_ROBOT_NOT_FOUND - robot not found.
N_CONNECTION_FAILED - the socket was disconnected since the last client call.

UPDATED GLOBALS

N_RobotState

DESCRIPTION

N_GetState is the equivalent of doing all gets except for N_GetTimer, which must be called explicitly for an update.

The N_RobotState Structure is the repository used in all data exchanges between a user program and a Robot Process. Each time a user program connects to a Robot, an N_RobotState Structure is created to receive data from the robot. There are as many N_RobotState Structures as there are robots connected to the user program.

The N_RobotState Structure is defined in Nclient.h h as:

```
struct N_RobotState
{
  N_CONST long RobotID;
  N_CONST char RobotType;
  struct N_Integrator Integrator;
  struct N_AxisSet AxisSet;
  struct N_LiftController LiftController;
  struct N_Joystick Joystick;
  struct N_SonarController SonarController;
  struct N_InfraredController InfraredController;
  struct N_BumperController BumperController;
  struct N_Compass Compass;
  struct N_LaserSet LaserSet;
  struct N_S550Set S550Set;
  struct N_BatterySet BatterySet;
  struct N_Timer Timer;
};
```

EXAMPLES

N_GetState.c

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

#define SCHEDULER_HOSTNAME "fatboy"
#define SCHEDULER_PORT 7073

int main()
{
  struct N_RobotState *state;
  struct N_SonarSet *sonar_set;
  short i;

  N_InitializeClient(SCHEDULER_HOSTNAME, SCHEDULER_PORT);
  N_ConnectRobot(1);

  state = N_GetRobotState(1);
  N_GetState(1);
  printf("Got all robot data in local state structure\n");

  N_DisconnectRobot(1);
  exit(0);
}
```

SEE ALSO

None

N_GETTIMER

NAME

N_GetTimer

PURPOSE

To get the current timeout setting of the robot and / or to get the master clock value.

SYNTAX

```
int N_GetTimer(long RobotID);
```

ARGUMENTS

`long RobotID` - robot identification number.

RETURNED VALUE

`N_NO_ERROR` - success.
`N_ROBOT_NOT_FOUND` - robot not found.
`N_CONNECTION_FAILED` - the socket was disconnected since the last client call.

UPDATED GLOBALS

`N_RobotState`

DESCRIPTION

This function reads the current timeout setting of the robot and updates the `N_RobotState` Structure.

```
struct N_Timer
{
  long Timeout;
  unsigned long Time;
};
```

`Timeout` is the timer threshold, after which the base motors will be turned off in units of milliseconds. For safety reasons, the user cannot set the `Timeout` parameter to exceed 1500 milliseconds.

`Time` is the robot's master reference clock and represents the amount of time the robot has been powered up. Both parameters are in units of milliseconds.

EXAMPLES

`N_GetTimer.c`

```
#include <stdio.h>
#include “Nclient.h”

#define SCHEDULER_HOSTNAME “fatboy”
#define SCHEDULER_PORT 7073

int main()
{
  struct N_RobotState *state;
  struct N_Timer *timer;
  short i;

  N_InitializeClient(SCHEDULER_HOSTNAME, SCHEDULER_PORT);
```

```
  N_ConnectRobot(1);

  state = N_GetRobotState(1);
  timer = &(state->Timer);

  N_GetTimer(1);

  printf("Timeout is currently set to %d seconds\n", timer->Timeout);

  N_DisconnectRobot(1);
  exit(0);
}
```

SEE ALSO

None

N_INITIALIZECLIENT

NAME

`N_InitializeClient`

PURPOSE

To initialize the communication with the scheduler.

SYNTAX

```
int N_InitializeClient(const char *scheduler_hostname,
                       unsigned short scheduler_socket);
```

ARGUMENTS

`const char *scheduler_hostname` - hostname or IP address
`unsigned short scheduler_socket` - socket number

RETURNED VALUE

None

UPDATED GLOBALS

None

DESCRIPTION

This function initializes the client library and specifies the hostname and port number of the scheduler. A scheduler is required whenever there are more than two (real or simulated) robots in the application. As a special case, when there is only one real robot, `N_InitializeClient` should be called with the real robot's hostname and listener socket.

EXAMPLES

`N_InitializeClient.c`

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

#define SCHEDULER_HOSTNAME "fatboy"
#define SCHEDULER_PORT 7073

int main()
{
  printf("Initializing the client with scheduler on machine %s,
         port %d\n", SCHEDULER_HOSTNAME, SCHEDULER_PORT);

  N_InitializeClient(SCHEDULER_HOSTNAME, SCHEDULER_PORT);
  N_ConnectRobot(1);

  /* Something useful... */

  N_DisconnectRobot(1);
  exit(0);
}
```

SEE ALSO

None

N_RETRACTLIFT

NAME

N_RetractLift

PURPOSE

To retract the Lift Mechanism into the robot of the robot to prevent damage to the Lift Mechanism and to simplify path planning.

SYNTAX

```
int N_RetractLift(long RobotID);
```

ARGUMENTS

`long RobotID` - robot identification number.

RETURNED VALUE

`N_NO_ERROR` - success.
`N_ROBOT_NOT_FOUND` - robot not found.
`N_CONNECTION_FAILED` - the socket was disconnected since the last client call.

UPDATED GLOBALS

`N_RobotState`

DESCRIPTION

This function initiates retracting the Lift Mechanism which may be in a deployed state.

EXAMPLES

`N_RetractLift.c`

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

#define SCHEDULER_HOSTNAME “fatboy”
#define SCHEDULER_PORT 7073

int main()
{
  struct N_RobotState *state;
  struct N_LiftController *lcont;
  N_InitializeClient(SCHEDULER_HOSTNAME, SCHEDULER_PORT);
  N_ConnectRobot(1);

  /* zero lift -- but only if necessary */
  N_ZeroLift(1, FALSE);
  N_RetractLift(1);
  /* wait for lift to retract */
  do
  {
    state = N_GetRobotState(1);
    lcont=&(state->LiftController);
  }
  while (lcont->InProgress);
```

```
  N_DisconnectRobot(1);
  exit(0);
}
```

SEE ALSO

N_RetractLift

N_SETAXES

NAME

N_SetAxes

PURPOSE

To move the robot based on the axis parameters in the N_RobotState structure.

SYNTAX

```
int N_SetAxes(long RobotID);
```

ARGUMENTS

long RobotID - robot identification number.

RETURNED VALUE

N_NO_ERROR - success.
N_INVALID_ARGUMENT - invalid argument.
N_ROBOT_NOT_FOUND - robot not found.
N_CONNECTION_FAILED - the socket was disconnected since the last client call.
N_UNKNOWN_ERROR - Nrobot was not able to communicate with the motor device.
N_AXES_NOT_READY - The axes are not free to move. Check the Status field.

UPDATED GLOBALS

N_RobotState

DESCRIPTION

This function updates the motor axes to reflect the current axis parameters in the N_AxisSet structure of N_RobotState and cause the desired movement.

```
struct N_AxisSet
{
  BOOL Global;
  unsigned char Status;
  N_CONST unsigned int AxisCount;
  struct N_Axis Axis[N_MAX_AXIS_COUNT];
};
```

- Global: Only available on XR4000 series robots. When this parameter is set to a TRUE value, it puts the mobile base axes in “global” or “world” mode after which all control of the base axes is with respect to the fixed global reference frame. The global reference frame is set when the robot is powered up and or when N_SetIntegratedConfiguration is called. While in Global mode (Global=TRUE), movement of the rotation axis will not change the direction of the x or y axis movement. For example, simultaneous movement of the Rotation and Y axes will cause a pirouette (motion in a straight line while spinning.)

  When this parameter is set to FALSE, the mobile base axes are controlled with respect to a “local” or “joint” reference frame. For example, simultaneous movement of the Rotation and Y axes will cause the robot to move in a circle -- that is, the robot will constantly move forward, but rotation causes the Y direction to change, producing a circle.

- `Status`: one of the following values:

  `N_AXES_READY`: This status indicates that the axes are available for movement.
  `N_JOYSTICK_IN_USE`: This status indicates that the base is being controlled via joystick.
  `N_ESTOP_DOWN`: This status indicates that one or more of the emergency stop buttons is depressed, preventing the robot from moving.

```
struct N_Axis                              {
  BOOL DataActive;
  BOOL TimeStampActive;
  BOOL Update;
  unsigned long TimeStamp;
  char Mode;
  long DesiredPosition;
  long DesiredSpeed;
  long Acceleration;
  long TrajectoryPosition;
  long TrajectoryVelocity;
  long ActualPosition;
  long ActualVelocity;
  BOOL InProgress;
  long TrajectoryVelocity;
};
```

- `DataActive`: A value for this parameter causes the values in this structure to be updated -- namely `Mode`, `DesiredPosition`, `DesiredSpeed`, `Accleration`, `TrajectoryPosition`, `TrajectoryVelocity`, `ActualPosition`, `ActualVelocity`, `InProgress`, and `TrajectoryVelocity`.
- `TimeStampActive`: A `TRUE` value for this parameter causes the TimeStamp parameter to be updated.
- `Update`: A TRUE value for this parameter causes the input values (`DesiredSpeed`, `DesiredPosition`, and `Acceleration`) to be loaded into the set of working values for this axis when a call to `N_SetAxes()` is made. The `Update` parameter allows one or more axes to be loaded with new input values simultaneously.
- `TimeStamp`: the time value in milliseconds that the axis values were measured.
- `Mode`: One of the following possible modes:

  `N_AXIS_POSITION_RELATIVE`: Specifies that the axis move relative to the current position.

  `N_AXIS_POSITION_ABSOLUTE`: Specifies that the axis move to an absolute position with respect to the absolute zero location of the axis.

  `N_AXIS_VELOCITY`: Specifies that the axis move with a constant velocity.

  `N_AXIS_STOP`: Specifies that the axis should decelerate to zero.

- `DesiredPosition`: Specifies the desired endpoint position of the axis. The units are in millimeters for translational axes and milliradians for rotational axes. When Mode is set to `N_AXIS_POSITION_RELATIVE` or `N_AXIS_POSITION_ABSOLUTE` this specifies the endpoint position relative to the current position or the absolute position, respectively. This parameter is not used when Mode is set to `N_AXIS_VELOCITY`. **This parameter can be positive or negative.**
- `DesiredSpeed`: This specifies the desired speed at which to move to the `DesiredPosition`, or the constant speed at which to move if Mode is set to `N_AXIS_VELOCITY`. The units are in millimeters / second for translational axes and milliradians / second for rotational axes. **This parameter can only be positive.**

- `Acceleration`: Constrains the rate at which the speed can change. The units are in millimeters / second^2 for translational axes and milliradians / second^2 for rotational axes. **This parameter can only be positive.**
- `TrajectoryPosition`: Provides the current position of the trajectory generator.
- `TrajectoryVelocity`: Provides the current velocity of the trajectory generator.
- `ActualPosition`: Provides the actual position of the axis. The value of this field is based on the setting of the Global field in `N_AxisSet`. If in Joint mode (`Global=FALSE`), this field provides the joint position of this axis. If in `Global` mode (`Global=TRUE`), this field provides the position of the axis in a global reference frame. In `Global` mode, this value is equivalent to the corresponding field in the `N_Integrator` structure.
- `ActualVelocity`: Provides the actual measured velocity of the axis. This value, like `ActualPosition` is based on the setting of the Global field in `N_AxisSet`. In `Global` mode (`Global=TRUE`), this field provides a value that is with respect to a global reference frame. In Joint mode, (`Global=FALSE`) this field provides a value that is with respect to the local reference frame.
- `InProgress`: Provides a boolean value that informs the user that an axis is currently moving.

The interpretation and range for the axes and their values according to the robot type are given below:

EXAMPLES

	Nomad 200	**XR4000**
Axis 0 is	Translation	X
Axis 1 is	Steering	Y
Axis 2 is	Turret	Theta
Axis 0 velocity range	[-609mm/s, 609mm/s]	[-1500mm/s, 1500mm/s]
Axis 1 velocity range	[-785mRad/s, 785mR/s]	[-1500mm/s, 1500mm/s]
Axis 2 velocity range	[-785mRad/s, 785mRad/s]	[-5000mRad/s, 5000mRad/s]
Axis 0 acceleration range	[0mm/s^2, 762mm/s^2]	[0mm/s^2, 1500mm/s^2]
Axis 1 acceleration range	[0mRad/s^2, 872mRad/s^2]	[0mm/s^2, 1500mm/s^2]
Axis 2 acceleration range	[0mRad/s^2, 872mRad/s^2]	[0mRad/s^2, 5000mRad/s^2]

`N_GetAxes.c`

```
#include <stdio.h>
#include “Nclient.h”

#define SCHEDULER_HOSTNAME “fatboy”
#define SCHEDULER_PORT 7073

int main()
{
  struct N_RobotState *state;
```

```
  struct N_Axis *axis;
  struct N_AxisSet *axisSet;
  int inProgress;

  N_InitializeClient(SCHEDULER_HOSTNAME, SCHEDULER_PORT);
  N_ConnectRobot(1);

  state = N_GetRobotState(1);
  axisSet=&(state->N_AxisSet);
  axis = &(AxisSet->Axis[N_YTRANSLATION]);
  axisSet->Global=FALSE;  /* local axis mode */
  axis->DataActive=TRUE;
  axis->Mode=N_AXIS_POSITION_RELATIVE;
  axis->DesiredSpeed=500;         /* mm/s */
  axis->acceleration=500;         /* mm/s/s */
  axis->DesiredPosition=1000;     /* mm */
  axis->Update=TRUE;
  N_SetAxes(1);

  /* move 1 meter forward and stop */
  do
  {
     N_GetAxes(1);
     inProgress=axis->InProgress;
   }
  while(inProgress);
  N_DisconnectRobot(1);
  exit(0);
}
```

SEE ALSO

N_GetAxes

N_SETINTEGRATEDCONFIGURATION

NAME

N_SetIntegratedConfiguration

PURPOSE

To set the integrated configuration values currently in the N_RobotState Structure.

SYNTAX

```
int N_SetIntegratedConfiguration(long RobotID);
```

ARGUMENTS

long RobotID - robot identification number.

RETURNED VALUE

N_NO_ERROR - success.
N_ROBOT_NOT_FOUND - robot not found.
N_CONNECTION_FAILED - the socket was disconnected since the last client call.
N_UNKNOWN_ERROR - Nrobot was not able to communicate with the device.

UPDATED GLOBALS

N_RobotState

DESCRIPTION

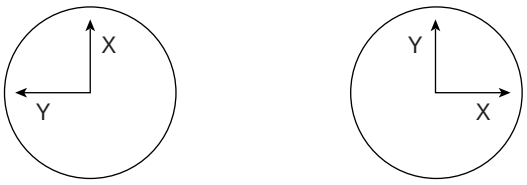
This function sets the integrated configuration values of the robot with the values currently in the N_Integrator field of the N_RobotState.

The N_Integrator structure contains the geometric configuration of the robot with respect to a globally fixed coordinate frame.

```
struct N_Integrator
{
  BOOL DataActive;
  BOOL TimeStampActive;
  long x;
  long y;
  long Steering;
  long Rotation;
  unsigned long TimeStamp;
};
```

- The DataActive is set to TRUE if the fields in this structure are to be updated -- namely x, y, Steering and Rotation.
- The TimeStampActive is set to TRUE if the TimeStamp field is to be updated.
- The x and y coordinates represent the integrated coordinates of the reference point of the robot (which is the center of the base for both the Nomad 200 and the XR4000) in the global reference frame. These values are expressed in millimeters. The reference frame is by default aligned with the position of the robot at startup.

Reference Frames



Nomad 200 Nomad XR4000

- The `Steering` angle represents the orientation of the wheels for the Nomad 200, with respect to the "X" axis of the reference frame. For the XR4000, the value is equal to `Rotation`. Angle units are milliradians.
- `Rotation`: For the Nomad 200, this is the angle in milliradians of the turret with respect to the "X" axis of the reference frame. For the XR4000, this is the angle in milliradians of the body with respect to the global reference frame.
- The `TimeStamp` is the time at which the integrated values were computed. It is expressed in milliseconds.

EXAMPLES

N_SetIntegratedConfiguration.c

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

#define SCHEDULER_HOSTNAME "fatboy"
#define SCHEDULER_PORT 7073

int main()
{
  struct N_RobotState *state;
  struct N_Integrator *configuration;
  N_InitializeClient(SCHEDULER_HOSTNAME, SCHEDULER_PORT);
  N_ConnectRobot(1);

  state = N_GetRobotState(1);
  configuration =&(state->Integrator);

  configuration->x = 0;
  configuration->y = 0;
  configuration->z = 0;
  configuration->Steering = 0;
  configuration->Rotation = 0;

  N_SetIntegratedConfiguration(1);

  printf("Integrated Configuration set to: x %d y %d z %d
         Steering %d Rotation %d\n",
         configuration->x,
         configuration->y,
         configuration->z,
         configuration->Steering,
         configuration->Rotation);
```

```
  N_DisconnectRobot(1);
  exit(0);
}
```

SEE ALSO

N_GetIntegratedConfiguration

N_SETJOYSTICK

NAME

N_SetJoystick

PURPOSE

To move the robot based on the parameters in the RobotState structure.

SYNTAX

```
int N_SetIntegratedConfiguration(long RobotID);
```

ARGUMENTS

`long RobotID` - robot identification number.

RETURNED VALUE

`N_NO_ERROR` - success.
`N_ROBOT_NOT_FOUND` - robot not found.
`N_CONNECTION_FAILED` - the socket was disconnected since the last client call.
`N_INVALID_ARGUMENT` - one of the entries in the N_Joystick structure was invalid.
`N_UNKNOWN_ERROR` - Nrobot was not able to communicate with the device.

UPDATED GLOBALS

None

DESCRIPTION

This function sends the values in the N_Joystick substructure of the state structure to the robot, in order to move the robot as though the user were using the joystick. The N_Joystick structure has fields for the two joystick axes (as well as a potential third one in the future) as well as three buttons.

```
struct N_Joystick
{
  double X;
  double Y;
  double Theta;
  BOOL ButtonA;
  BOOL ButtonB;
  BOOL ButtonC;
};
```

- X, Y, and Theta are axis values and must fall in the range [-1.0, 1.0].
- The three Button parameters are set to TRUE to indicate that the button is pressed. If none are set to TRUE, this indicates that the joystick has been released.

N_SETLIFT

NAME

N_SetLift

PURPOSE

To move the Lift and Grip axes of the Lift Mechanism.

SYNTAX

```
int N_SetLift(long RobotID);
```

ARGUMENTS

`long RobotID` - robot identification number.

RETURNED VALUE

`N_NO_ERROR` - success.
`N_ROBOT_NOT_FOUND` - robot not found.
`N_CONNECTION_FAILED` - the socket was disconnected since the last client call.

UPDATED GLOBALS

None

DESCRIPTION

This function initiates movement of the Lift and Grip axes of the Lift Mechanism after the contents of the `LiftController` structure of `N_RobotState` have been modified to reflect the desired Lift Mechanism motion.

The `LiftController` structure has the following definition:

```
struct N_LiftController
{
  BOOL Deployed;
  BOOL InProgress;
  struct N_LiftAxis Axis[N_LIFT_AXIS_COUNT];
};
```

<toc 2>

The fields are described as follows:

- `Deployed`: provides information on whether the Lift Mechanism is fully deployed or not. A `TRUE` value indicates that the Lift mechanism is fully deployed.
- `InProgress`: a `TRUE` value in this field indicates that the Lift Mechanism is currently being retracted, deployed, or zeroed (i.e. due to a call to `N_DeployLift`, `N_RetractLift`, or `N_ZeroLift`). While this field is set, no motion commands to the Lift mechanism are accepted.
- `Axis`: an array that contains the motion parameters for the two controllable axes: the Lift axis and Grip axis. The two possible array indices are respectively `N_LIFT` and `N_GRIP`.

The axis parameters are contained in the `N_LiftAxis` structure, which is defined as:

```
struct N_LiftAxis
{
  BOOL DataActive;
```

```
  BOOL TimeStampActive;
  BOOL Update;
  unsigned long TimeStamp;
  char Mode;
  long Status;
  long DesiredPosition;
  long DesiredVelocity;
  long DesiredAcceleration;
  long MaxMotor;
  long Position;
  long Velocity;
};
```

- `DataActive`: a `TRUE` value for this field causes the parameters of this structure to be updated with each call to `N_GetAxes` -- namely `Mode`, `Status`, `DesiredPosition`, `DesiredVelocity`, `Acceleration`, `MaxMotor`, `Position` and `Velocity`.
- `TimeStampActive`: a `TRUE` value for this field causes the `TimeStamp` field to be updated with each call to `N_GetAxes`.
- `Update`: a `TRUE` value for this field causes the motion parameters to be loaded into the motor controller for execution upon calling `N_SetAxes()`.
- `TimeStamp`: contains the time in milliseconds that the motion parameters were obtained.
- `Mode`: contains the movement mode for the axis. It can be set to one of the following possible modes:

  `N_LIFT_POSITION_RELATIVE`: causes the axis to move to a position relative to the current position as specified by the `DesiredPosition` field.

  `N_LIFT_POSITION_ABSOLUTE`: causes the axis to move to an absolute position (see table above) as specified by the `DesiredPosition` field.

  `N_LIFT_VELOCITY`: causes the axis to move at a velocity specified by the `DesiredVelocity` field.

- `Status`: provides a bitmap of possible errors. One or more of these bits will be set if an error occurs during a movement. To test for a particular error, the value or result can be "ored" with the error bits described below:

  `N_LIFT_POS_LIMIT`: indicates the positive limit switch has been reached for the axis. This should never happen, as the software limits travel to never traverse the positive limit of neither the Grip nor Lift axes after the Lift Mechanism is zeroed.

  `N_LIFT_NEG_LIMIT`: indicates the negative limit switch has been reached for the axis. This should never happen for the Lift axis, but the Grip axis when fully closed depresses the gripper's negative limit switch.

  `N_LIFT_MOTION_ERROR`: indicates that the motion was unable to complete. This is often due to an obstacle in the desired path of the axis.

  `N_LIFT_ESTOP`: indicates that the motion is not possible because one or more of the emergency stop switches is depressed.

  `N_LIFT_CANNOT_MOVE`: indicates that the motion is not possible because the Lift Mechanism is not zeroed or not deployed.

- `DesiredPosition`: specifies the desired endpoint position of the axis in units of 0.1 mm.. The position is either relative to the current position or absolute with respect to the zero position as specified by the Mode field.

- `DesiredVelocity`: specifies the desired velocity at which the axis should move in units of 0.1 mm/s.
- `DesiredAcceleration`: specifies the desired acceleration at which the axis should move in units of 0.1 mm/s^2.
- `MaxMotor`: specifies the percentage of available torque to use for the axis. The possible values range between 0 and 100, where 100 specifies the default of 100% available torque. For example, this is useful to set for the gripper when picking up fragile objects.
- `Position`: provides the current absolute position of the axis in units of 0.1mm.
- `Velocity`: provides the current velocity of the axis in units of 0.1mm/s.

EXAMPLES

N_SetLift.c

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

#define SCHEDULER_HOSTNAME “fatboy”
#define SCHEDULER_PORT 7073

int main()
{
  struct N_RobotState *state;
  struct N_LiftController *lcont;

  N_InitializeClient(SCHEDULER_HOSTNAME, SCHEDULER_PORT);
  N_ConnectRobot(1);

  /* zero lift -- but only if necessary */
  N_ZeroLift(1, FALSE);

  /* Lift Mechanism must be deployed in order to move */
  N_DeployLift(1);
  /* wait for lift to deploy */
  do
  {
     state = N_GetLift(1);
    lcont=&(state->LiftController);
  }
  while (lcont->InProgress);
  lcont->Axis[N_GRIP].MaxMotor=50;        /* set the gripper torque to 50% */
  lcont->Axis[N_GRIP].Mode=N_LIFT_POSITION_ABSOLUTE; /* move the gripper to
the specified absolute position */
  lcont->Axis[N_GRIP].DesiredVelocty=300;              /* move at 30 mm/s */
  lcont->Axis[N_GRIP].DesiredAcceleration=300;  /* accelerate at 30 mm/s/s */
  lcont->Axis[N_GRIP].DesiredPosition=0; /* move the gripper to the zero posi-
tion (absolute) -- closed */
  N_SetLift();

  lcont->Axis[N_GRIP].DataActive=TRUE;         /* turn on data active flag */
  /* wait until finished */
  do
  {
     N_GetLift(1);
```

```
  }
  while (lcont->Axis[N_GRIP].Velocity != 0);
  N_DisconnectRobot(1);
  exit(0);
}
```

SEE ALSO

N_GetLift

N_SETSONARCONFIGURATION

NAME

N_SetSonarConfiguration

PURPOSE

To set the configuration of the sonar proximity sensors.

SYNTAX

```
int N_SetSonarConfiguration(long RobotID);
```

ARGUMENTS

`long RobotID` - robot identification number.

RETURNED VALUE

`N_NO_ERROR` - success.
`N_NO_ROBOT_FOUND` - robot not found.
`N_CONNECTION_FAILED` - the socket was disconnected since the last client call.

UPDATED GLOBALS

`N_RobotState`

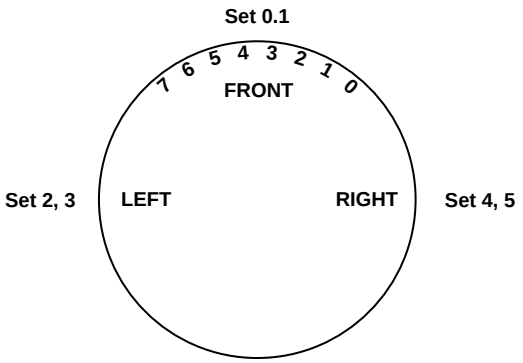
DESCRIPTION

This function configures the sonar sensor proximity sensors the values currently in the `N_RobotState` Structure.

The sonar controller structure holds all the configuration data that is valid for all the sonar sets on the robot. Sonar sets are groups of sonar that act together. For instance, a firing order can be defined among the sonar of a given set. The sonar controller structure has an array of `N_SonarSet` structures, each describing one particular set.

The Nomad 200 has one set of 16 sonar, going counterclockwise from 0 to 15, with 0 in front.

The XR4000 has three doors that go counterclockwise and there are two (sonar / infrared / bumper) sets per door for a total of six sets:



Set 0 = top set on front door (sensor #0–7)
Set 1 = bottom set on front door (sensor #0–7)
Set 2 = top set on left door (sensor #0–7)
Set 3 = bottom set on left door (sensor #0–7)
Set 4 = top set on right door (sensor #0–7)
Set 5 = bottom set on right door (sensor #0–7)

```
struct N_SonarController
{
  N_CONST unsigned int SonarSetCount;
  struct N_SonarSet SonarSet[N_MAX_SONAR_SET_COUNT];
  BOOL SonarPaused;
};
```

The configuration data used globally by all the sonar sets on the robot are:

- `SonarSetCount`: the number of sonar sets in the controller.

The `N_SonarSet` structure is defined as follows:

```
struct N_SonarSet
{
  unsigned int FiringOrder[N_MAX_SONAR_COUNT + 1];
  long FiringDelay;
  long BlankingInterval;
  BOOL DataActive;
  BOOL TimeStampActive;
  N_CONST unsigned int SonarCount;
  struct N_Sonar Sonar[N_MAX_SONAR_COUNT];
};
```

It contains an array `Sonar` of sonar structures (one per sonar transducer in the set), plus a number of configuration parameters. The configuration parameters are:

- `FiringOrder`: an array of sonar indices terminated by `N_END_SONAR_FIRING_ORDER` if the length of the array is less than the `SonarCount`.
- `FiringDelay`: delay in milliseconds between two consecutive sonar firings.
- `BlankingInterval`: the time in milliseconds to wait after a sonar sensor has fired before the sensor begins to listen. **This is currently not implemented on the XR4000 Release 1.0.**
- `DataActive`: set to a TRUE value if the sonar data is to be updated for this set.
- `TimeStampActive`: set to a TRUE value if the time (time of acquisition of the sonar range) is to be updated for this set.
- `SonarCount`: number of sonars in this set.

The `N_Sonar` structure contains the sensor readings:

```
struct N_Sonar
{
  long Reading;
  unsigned long TimeStamp;
};
```

The values defined for each sonar are:

- `Reading`: the distance measurement of the sensor in millimeters.
- `TimeStamp`: the time that the measurement took place in milliseconds.

EXAMPLES

N_SetSonarConfiguration.c

```
#include <stdio.h>
#include “Nclient.h”

#define SCHEDULER_HOSTNAME “fatboy”
#define SCHEDULER_PORT 7073

int main()
{
  struct N_RobotState *state;
  struct N_SonarSet *sonar_set;
  short i;

  N_InitializeClient(SCHEDULER_HOSTNAME, SCHEDULER_PORT);
  N_ConnectRobot(1);

  state = N_GetRobotState(1);
  sonar_set = &state->SonarController.SonarSet[0];
  /* Always do this to initialize the local state structure
   * with correct data */
  N_GetSonarConfiguration(1);

  sonar_set->DataActive = TRUE;
  sonar_set->TimeStampActive = TRUE;

  sonar_set->FiringOrder[0] = 0;
  sonar_set->FiringOrder[1] = 2;
  sonar_set->FiringOrder[2] = 4;
  sonar_set->FiringOrder[3] = N_END_SONAR_FIRING_ORDER;
  N_SetSonarConfiguration(1);

  printf(“Sonar Set 0 set to\n”);
  printf(“Data Active: %s Time Stamp Active: %s\n”,
         sonar_set->DataActive ? “Yes” : “No”,
         sonar_set->TimeStampActive ? “Yes” : “No”);
  printf(“Firing order: “);
  i = 0;
  while (sonar_set->FiringOrder[i]!=N_END_SONAR_FIRING_ORDER)
  {
    printf(“%d “, sonar_set->FiringOrder[i]);
    i+=1;
  }
  N_DisconnectRobot(1);
  exit(0)
}
```

SEE ALSO

N_GetSonarConfiguration

N_SETTIMER

NAME

N_SetTimer

PURPOSE

To set the timeout period of the robot.

SYNTAX

```
int N_SetTimer(long RobotID);
```

ARGUMENTS

`long RobotID` - robot identification number.

RETURNED VALUE

`N_NO_ERROR` - success;
`N_ROBOT_NOT_FOUND` - robot not found;

UPDATED GLOBALS

`N_RobotState`

DESCRIPTION

This function sets the timeout period of the robot, such that if the robot has not received a command from the host for more than the timeout period, it will abort its current motion. This is a safety measure to prevent the robot from continuing its motion without control if for some reason (crash, communication problem…) the robot does not receive any command at all.

```
struct N_Timer
{
  long Timeout;
  long Time;
};
```

`Timeout` is the timer threshold, after which the base motors will be turned off in units of milliseconds. For safety reasons, the user cannot set the `Timeout` parameter to exceed 1500 milliseconds.

`Time` is the robot's master reference clock and represents the amount of time the robot has been powered up. Both parameters are in units of milliseconds.

EXAMPLES

`N_SetTimeout.c`

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

#define SCHEDULER_HOSTNAME "fatboy"
#define SCHEDULER_PORT 7073
int main()
{
  struct N_RobotState *state;
  struct N_Timer *timer;
```

```
  short i;

  N_InitializeClient(SCHEDULER_HOSTNAME, SCHEDULER_PORT);
  N_ConnectRobot(1);

  state = N_GetRobotState(1);
  timer = &(state->Timer);

  timer->Timeout = 10000;

  N_SetTimer(1);

  printf("Timeout set to %d milliseconds\n", timer->Timeout);

  N_DisconnectRobot(1);
  exit(0);
}
```

SEE ALSO

N_GetTimer

N_SPEAK

NAME

N_Speak

PURPOSE

To download a string of text for the speech card to speak.

SYNTAX

```
int N_Speak(long RobotID, char *Text);
```

ARGUMENTS

`unsigned long RobotID` - robot identification number.
`char *Text` - text string the robot will speak.

RETURNED VALUE

`N_NO_ERROR` - success.
`N_ROBOT_NOT_FOUND` - robot not found.
`N_CONNECTION_FAILED` - the socket was disconnected since the last client call

UPDATED GLOBALS

None

DESCRIPTION

This function sends a text stream in characters to the robot's voice synthesizer (if available) to let the robot speak. Please refer to the voice synthesizer documentation for information about speech control characters.

EXAMPLES

`N_Speak.c`

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

#define SCHEDULER_HOSTNAME "fatboy"
#define SCHEDULER_PORT 7073

int main()
{
  struct N_RobotState *state;
  struct N_Timer *timer;
  short i;

  N_InitializeClient(SCHEDULER_HOSTNAME, SCHEDULER_PORT);
  N_ConnectRobot(1);

  N_Speak(1, "Hello World");

  N_DisconnectRobot(1);
  exit(0);
}
```

SEE ALSO

None

N_ZEROLIFT

NAME

N_ZeroLift

PURPOSE

To "zero" the Lift Mechanism so the Lift Controller knows the position of the axes. If the Lift Mechanism is not zeroed, motion commands initiated by N_SetLift() will not execute.

SYNTAX

```
int N_ZeroLift(long RobotID, BOOL Force);
```

ARGUMENTS

long RobotID - robot identification number.

BOOL Force - a boolean value that if passed as TRUE will cause the Lift Mechanism to perform the zeroing motions each time it is called, otherwise it will perform the zeroing motions once per boot cycle of the robot.

RETURNED VALUE

N_NO_ERROR - success.
N_ROBOT_NOT_FOUND - robot not found.
N_CONNECTION_FAILED - the socket was disconnected since the last client call.

UPDATED GLOBALS

N_RobotState

DESCRIPTION

When the robot is turned on, the position of the lift mechanism's axes are not known. In order to control the lift mechanism, the axis positions must be determined. This is done by "zeroing" the lift mechanism with a call to N_ZeroLift(). Only one call the N_ZeroLift is required for each boot cycle of the robot. If the Lift Mechanism is not zeroed, motion commands initiated by N_SetLift() will not execute.

EXAMPLES

N_ZeroLift.c

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

#define SCHEDULER_HOSTNAME "fatboy"
#define SCHEDULER_PORT 7073

int main()
{
  struct N_RobotState *state;
  struct N_LiftController *lcont;
  N_InitializeClient(SCHEDULER_HOSTNAME, SCHEDULER_PORT);
  N_ConnectRobot(1);

  /* do not force -- only zero if necessary */
  N_ZeroLift(1, FALSE);
```

```
  /* wait for lift to deploy */
  do
  {
     state = N_GetRobotState(1);
    lcont=&(state->LiftController);
   }
  while (lcont->InProgress);

  N_DisconnectRobot(1);
  exit(0);
}
```

SEE ALSO

N_SetLift