# Language Reference Manual

March 11, 1997

*Software Version*: 2.6

*Part Number*: DOC00002

NOMADIC
TECHNOLOGIES, INC.

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

## Where can I get help ?

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

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

## Where can I get software ?

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

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

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

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

(Password information given on request to Nomadic Customers)

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

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

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

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

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

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

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

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

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

# QUICK REFERENCE

## Robot Commands

### Communication Commands

connect_robot .................................. connects to a robot
disconnect_robot ...................... closes connection with a robot
conf_tm ......................... sets the timeout period of the robot
real_robot .............................. switches to real robot mode
simulated_robot .................. switches to simulated robot mode
quit_server ................................ causes the server to quit
tk .......... sends a character stream to the robot's voice synthesizer
special_request ...................... sends a user packet to the robot
init_send_buffer ................................. sets the send buffer
stuff_send_buffer_header ........... loads header into the send buffer
stuff— .............................. stuffs a — into the send buffer
init_receive_buffer ............... sets the initial point to extract data
extract_receive_buffer_header . extracts header from the receive buffer
extract— ....................... extracts a — from the receive buffer

### Motion Commands

arm_mv ............ moves arm and/or setup arm motion parameters
arm_ws ...................... waits for the specified arm axes to stop
arm_zr ............................ checks orientation or orients arm
pr ...................... moves the motors of the robot by a distance
vm .............................. moves the robot at given velocities
mv ................ moves the three axes of the robot independently
st ......................................... stops the robot's motors
ws .......................... waits for the stop of the robot's motors
lp ................................................... sets motor limp
zr ....................... aligns steering and turret zero with bumper

### Motion Parameters Setting Commands

dp ................................. defines the position of the robot
da ................................ defines the robot's steering angle
ac .................................... sets the robot's accelerations
sp .......................................... sets the robot's speeds

### Sensing Parameters Setting Commands

init_mask ...................................... resets sensor mask
ct .............................. sends the sensor mask to the robot
init_sensors ......... initializes sensor mask and sends it to the robot
conf_ir ........................ configures the infrared sensor system
conf_sn ........................... configures the sonar sensor system
conf_cp .............................. configures the compass system
conf_ls ........................... configures the laser ranging system
conf_sg .......... configures the line mode of the laser ranging system
posDataCheck gets sensors for which position information is requested

posDataRequests .. requests position information for sensory readings

### Motion Parameters Retrieving Commands

get_rc ............................ gets configuration data of the robot
get_rv .................................... gets velocities of the robot

### Sensory Data Retrieving Commands

get_ir .............................. gets the infrared data of the robot
get_sn ............................... gets the sonar data of the robot
get_bp ............................. gets the bumper data of the robot
get_cp ............................ gets the compass data of the robot
get_ls ............................... gets the laser data in point mode
get_sg ................................ gets the laser data in line mode
gs ................................ gets the current state of the robot
posBumperGet ......... gets the position information for the bumper
posCompassGet ....... gets the position information for the compass
posInfraredGet .. gets the position information for one infrared sensor
posInfraredRingGet gets the position information for the infrared ring
posLaserGet .............. gets the position information for the laser
posSonarGet ............. gets the position information for one sonar
posSonarRingGet .... gets the position information for the sonar ring
posTimeGet .. gets the time for the position information time stamps
voltCpuGet ............................ gets the CPU-battery voltage
voltMotorGet ......................... gets the motor-battery voltage

### Local Map Display Commands

draw_line ............................................... draws a line
draw_arc ............................................... draws an arc
draw_robot .......................................... draws a robot
get_robot_conf ........ gets interactively a point from robot's window

## World Commands

add_obstacle ...... adds an obstacle to the current robot environment
delete_obstacle deletes an obstacle from the current robot environment
move_obstacle ........... moves an obstacle in the robot environment
new_world ...................... clears all its obstacles from the map

# CONVENTIONS

NAME

< *Function name* >

PURPOSE

< *Purpose of the function* >

SYNTAX

< *C syntax of the function* >

ARGUMENTS

< *Type, meaning and range of the arguments, if any* >

RETURNED VALUE

< *Meaning of the returned value, if any* >

UPDATED GLOBALS

< *Updated global vectors, if any* >

DESCRIPTION

< *Description of the function* >

EXAMPLE(S)

< *Refer to one of the examples in Appendix B where the function is used* >

KNOWN BUGS

< *Known bugs, or limitations of the function for the current release* >

SEE ALSO

< *Related functions* >

# Chapter 1

# Program Instructions

NAME

**ac**

PURPOSE

It sets the translation, steering, and rotation accelerations of the robot.

SYNTAX

```
int ac (unsigned int t_ac, unsigned int s_ac, unsigned int r_ac)
```

ARGUMENTS

`int t_ac` – the translation acceleration in 1/10 inch/sec$^2$;
`int s_ac` – the steering acceleration in 1/10 degree/sec$^2$;
`int r_ac` – the turret acceleration in 1/10 degree/sec$^2$.

RETURNED VALUE

`1` – success;
`0` – otherwise.

UPDATED GLOBALS

`State` vector;
`Laser` vector.

DESCRIPTION

This function sets the translation, steering, and rotation accelerations of the robot to `t_ac`, `s_ac`, and `r_ac` respectively.

`t_ac` is a positive integer less than 300, initially set to 100;
`s_ac` is a positive integer less than 500, initially set to 200;
`r_ac` is a positive integer less than 500, initially set to 200.

*Note*: The defaults can be found in the file `motor.cfg` on the robot.

This function updates `State` and `Laser` according to the set `Smask`.

EXAMPLE

Motion

KNOWN BUGS

SEE ALSO

**sp**

NAME

**add_obstacle**

PURPOSE

It adds an obstacle to the current robot environment.

SYNTAX

`int add_obstacle (long obs[21])`

ARGUMENTS

`long obs[21]` – `obs[0]` specifies the number (no greater than 10) of vertices of the polygonal obstacle. `obs[1]` to `obs[20]` specify the $x$ and $y$ coordinates of the vertices, in counter-clockwise direction.

RETURNED VALUE

`1` – success;
`0` – otherwise.

UPDATED GLOBALS

DESCRIPTION

This function creates and adds an obstacle specified by `obs` to the current robot environment. Currently, an obstacle can have at most 10 vertices. `obs[0]` specifies the number of vertices of the polygonal obstacle. `obs[2i+1]` and `obs[2i+2]` specify the $i$th ($i = 0, ..., 9$) vertex of the polygon. The vertices of the polygon must be specified in counter-clockwise direction. The new obstacle appears in the world window of the graphic interface, if any, as soon as it is created.

EXAMPLE

World

KNOWN BUGS

SEE ALSO

**move_obstacle, delete_obstacle**

NAME

**arm_mv**

PURPOSE

Move and/or set up motion parameters

SYNTAX

`long arm_mv(long l_mode, long l_v, long g_mode, long g_v)`

ARGUMENTS

`long l_mode` – lift axis mode
`long l_v` – lift axis value
`long g_mode` – grip axis mode
`long g_v` – grip axis value

RETURNED VALUE

`0` – if successful
`-1` – if unsuccessful, or if arm is not currently oriented

DESCRIPTION

This is similar to the mv() routine for the base.
The valid modes are as follows:

**MV_IGNORE** ignore the information for that axis

**MV_VM** change to velocity mode. The corresponding `l_v` or `g_v` value specifies the speed of the axis in .1 in/sec. Since both axes have travel limits, this mode will automatically cause the motors to stop before hitting the limit switches. The client program can furthermore wait for the travel limit to be reached by issuing a `arm_ws()`.

**MV_PA** change to position absolute mode. See below for a description of the absolute positions of each axis. The corresponding `l_v` or `g_v` value specifies the absolute position in .1in.

**MV_PR** change to position relative mode. The corresponding `l_v` or `g_v` value specifies the relative position in .1in.

**MV_AC** set the acceleration for that axis. The corresponding `l_v` or `g_v` value specifies the acceleration in .1 in/sec$^2$.

**MV_SP** set the speed (velocity) for that axis. The corresponding `l_v` or `g_v` value specifies the speed in .1 in/sec.

**MV_MT** set the maximum torque limit for that axis. The corresponding `l_v` or `g_v` value ranges from 0 to 100 and specifies the relative percentage of maximum torque. Decreasing the maximum torque is useful for sensing collisions or for picking up objects.

The sign conventions and absolute positions for the arm are as follows:

- Lift in the up direction is positive travel.
- The grip axis is such that closing the gripper is positive travel.
- The lift axis in the contracted (center of travel) position is zero.
- The grip axis in the open (gripper forks apart) position is zero.
- The lift axis moves upward to a maximum absolute position of 227 .1 in and downward to a minimum of -188 .1 in.
- The grip axis opens to a maximum absolute position of 50 .1 in and closes to a minimum of 0 .1 in.

NAME

**arm_ws**

PURPOSE

Waits for specified arm axes to stop.

SYNTAX

`long arm_ws(short lift, short grip, long timeout, long *time_remain)`

ARGUMENTS

`short lift, short grip` – boolean variables that specify which axes to wait for stop, lift or grip

`long timeout` – how long to wait before timeout (in seconds)

`long *time_remain` – remaining time in the timeout is written here in milliseconds, if it returns before the timeout expires. This argument may be passed as NULL

RETURNED VALUE

`0` – if a stop or timeout occurred

`-1` – if an error occurred in any of the axes that are being waited upon

DESCRIPTION

This is similar to `ws()` routine for the base. It waits for the specified grip or lift axes to stop.

NAME

**arm_zr**

PURPOSE

Checks orientation or orients arm.

SYNTAX

`long arm_zr(short mode)`

ARGUMENTS

`short mode` – one of two zeroing modes

RETURNED VALUE

`0` – if successful

`-1` – if unsuccessful

DESCRIPTION

The possible modes are as follows:

**ZR_ORIENT** This mode will always cause the arm to orient itself with respect to the limit switches regardless of whether the arm is currently oriented or not.

**ZR_CHECK** This mode will cause the arm to orient itself with respect to the limit switches only if the arm is not currently oriented. Otherwise, it will not move the arm and return immediately. It is intended for `arm_zr(ZR_CHECK)` to be the first one-time call a process executes before moving the arm, which results in no wasted time if the arm is already oriented and results in the arm orienting itself if it is not.

NAME

**arm_ws**

PURPOSE

Waits for specified arm axes to stop.

SYNTAX

`long arm_ws(short lift, short grip, long timeout, long *time_remain)`

ARGUMENTS

`short lift, short grip` – boolean variables that specify which axes to wait for stop, lift or grip

`long timeout` – how long to wait before timeout (in seconds)

`long *time_remain` – remaining time in the timeout is written here in milliseconds, if it returns before the timeout expires. This argument may be passed as NULL

RETURNED VALUE

`0` – if a stop or timeout occurred

`-1` – if an error occurred in any of the axes that are being waited upon

DESCRIPTION

This is similar to `ws()` routine for the base. It waits for the specified grip or lift axes to stop.

NAME

**conf_cp**

PURPOSE

It configures the compass system.

SYNTAX

`int conf_cp (unsigned int mode)`

ARGUMENTS

`int mode` – specifies compass status: 0 = off ; 1 = on; 2 = calibrate.

RETURNED VALUE

`1` – success;
`0` – otherwise.

UPDATED GLOBALS

`State` vector;
`Laser` vector.

DESCRIPTION

This function configures the compass system.

If `mode` is set to 1, the compass will be activated.

If `mode` is set to 0, the compass will be shut off.

If `mode` is set to 2, the compass system will self-calibrate (this will cause the robot to rotate slowly for several minutes). After calibration of the compass system, the turret will no longer in the same orientation as before the calibration.

This function updates `State` and `Laser` according to the set `Smask`.

EXAMPLE

Sensing

KNOWN BUGS

SEE ALSO

**gs, get_cp**

NAME

**conf_ir**

PURPOSE

It configures the infrared sensor system.

SYNTAX

```
int conf_ir (unsigned int dependency, unsigned int order[16])
```

ARGUMENTS

`unsigned int dependency` – percentage dependency of the current returned reading on the previous returned reading, between 0 and 10: 0 = no dependency, 10 = full dependency;
`unsigned int order[16]` – firing sequence of the infrared (#0 .. #15).

RETURNED VALUE

`1` – success;
`0` – otherwise.

UPDATED GLOBALS

`State` vector;
`Laser` vector.

DESCRIPTION

This function configures the infrared sensor system. The parameter `dependency` specifies the percentage dependency of the current returned infrared reading on the previous returned reading. `dependency` should be set to between 0 and 10, representing independency to full dependency. The returned value is computed as:

$$\frac{dependency * history + (10 - dependency) * reading}{10}$$

history being the average of all the values returned so far. The parameter `order` specifies the firing sequence of the infrared sensors(#0, ..., #15). The infrared specified in `order[i]` will be fired before that specified in `order[i+1]`. The infrared sensors that are not specified in the order list will not be active. You can terminate the order list by a '255'. The infrared sensors are numbered counter-clockwise, the front one being the first (opposite to the Emergency Stop button).

This function updates `State` and `Laser` according to the set `Smask`.

EXAMPLE

Sensing

KNOWN BUGS

SEE ALSO

**gs, get_ir**

NAME

**conf_ls**

PURPOSE

It configures the laser ranging system.

SYNTAX

`int conf_ls (unsigned int mode, unsigned int threshold, unsigned int width, unsigned int num_data, unsigned int processing)`

ARGUMENTS

`unsigned int mode` – mode of the data coming back:

first bit: laser on/off;
second bit: point/line mode;
third to fifth bits: returned data types:

000 = peak pixel,
001 = rise pixel,
010 = fall pixel,
011 = magnitude,
100 = distance;

sixth bit: data integrity checking.

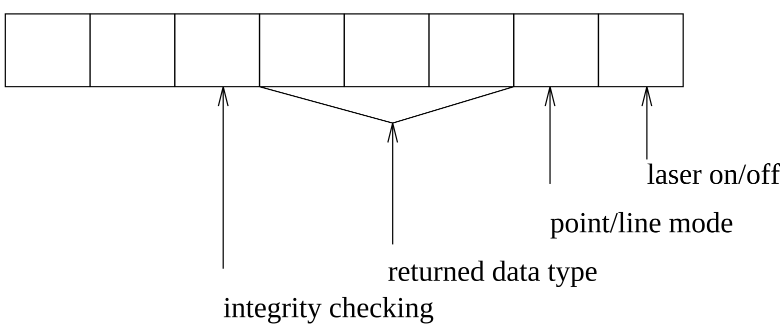


Figure 1.1: The Bits in the Mode Byte

`unsigned int threshold` – inverted acceptable brightness of the laser line;
`unsigned int width` – acceptable width in terms of number of pixels that are brighter than the set threshold;
`unsigned int num_data` – number of sampling points;
`unsigned int processing` – number of neighboring pixels for averaging.

RETURNED VALUE

`1` – success;
`0` – otherwise.

UPDATED GLOBALS

`State` vector;
`Laser` vector.

DESCRIPTION

This function configures the laser ranging system.

The parameter `mode` specifies the on-board processing mode of the laser sensor data which determines the mode of the data coming back. The first bit of the mode byte specifies the on/off of the laser system, the second bit specifies point/line mode (point mode if 1, line mode if 0), third to fifth bits specify the returned data types: (peak pixel if 0, rise pixel if 1, fall pixel if 2, magnitude if 3, and distance if 4). The sixth bit specifies data integrity checking (check if 1).
A range of 0xFFFF will be returned if the data failed the test.

In line mode, the peak pixel is automatically used. In point mode, the interpretation of the returned values depends on the value of the bits 3-5 of the `mode`: 0 to 3 will give pixels (therefore not using the calibration values), 4 will give distances (using the calibration values for the conversion).

The following table shows the valid mode values in point mode, with the type of the returned data. Integrity checking is on in each case (substract 32 to the mode value to remove integrity checking).

| mode | returned value |
|---:|---|
| 35 | peak pixel |
| 39 | rise pixel |
| 43 | fall pixel |
| 47 | magnitude |
| 51 | distance |

The values 33-37-41-45-49 are valid settings for the line mode. The peak pixel is always used.
The conversion pixel→distance uses calibration values. The computations for line mode are done on-board (using `laser.cal`) whereas the computations for point mode are done in the server (using the values of `robot.setup`). Therefore, the calibration numbers in these two files should be identical. These numbers are the result of the `calibration` utility provided with the Laser Range Finder.

DESCRIPTION

The parameter `threshold` specifies the inverted acceptable brightness of the laser light.

The parameter `width` specifies the acceptable peak width in terms of number of pixels that are brighter than the set threshold.

The parameter `num_data` specifies the number of sampling points.

The parameter `processing` specifies additional data for the on-board processing. For point mode, it specifies the half window size (in terms of number of pixels) used to extract the median.

`conf_ls (51, 10, 20, 16, 4)` is a recommended good initial setting.

This function updates `State` and `Laser` according to the set `Smask`.

EXAMPLE

Sensing

KNOWN BUGS

SEE ALSO

**gs, get_ls, conf_sg**

NAME

**conf_sg**

PURPOSE

It configures the line mode of the laser ranging system.

SYNTAX

```
int conf_sg (unsigned int threshold, unsigned int min_points,
unsigned int gap)
```

ARGUMENTS

`unsigned int threshold` – specifies the threshold value for least-square fitting. When the error term grows above the threshold, the line segment will be broken.
`unsigned int min_points` – specifies the acceptable number of points to form a line segment;
`unsigned int gap` – specifies the acceptable "gap" between two segments while they can still be treated as one (in 1/10 inches).

RETURNED VALUE

`1` – success;
`0` – otherwise.

UPDATED GLOBALS

`State` vector;
`Laser` vector.

DESCRIPTION

This function configures the line mode of the laser ranging system. Initially, `conf_sg(50, 4, 30)` is called during laser setup. The parameter `threshold` specifies the threshold value for least-square fitting. When the error term grows above this threshold, the line segment will be broken. The parameter `min_points` specifies the acceptable minimum number of points to form a line segment. The parameter `gap` specifies the acceptable gap between two segments while they can still be treated as one.

This function updates `State` and `Laser` according to the set `Smask`.

EXAMPLE

Sensing

KNOWN BUGS

SEE ALSO

**gs, get_sg, conf_ls**

NAME

**conf_sn**

PURPOSE

It configures the sonar sensor system.

SYNTAX

`int conf_sn (int firerate, int order[16])`

ARGUMENTS

`int firerate` – firing rate of the sonar;
`int order[16]` – firing sequence of the sonar (#0 .. #15).

RETURNED VALUE

`1` – success;
`0` – otherwise.

UPDATED GLOBALS

`State` vector;
`Laser` vector.

DESCRIPTION

This function configures the sonar sensor system. The parameter `firerate` specifies the rate the sonars are fired (i.e.the time between two sonars are fired) at 4 milli-second time intervals. `firerate` should be set between 0 and 255. Note that `firerate` starts after the end of the processing of the previous sonar, which in turns depends on the time it takes for the sound to come back.

The parameter `order` specifies the firing sequence of the sonar sensors(#0, ..., #15). The sonar specified in `order[i]` will be fired before that specified in `order[i+1]`. The sonar sensors that are not specified in the order list will not be active. You can terminate the order list by a '255'. The sonar sensors are numbered counterclockwise, the front one being the first (opposite to the Emergency Stop button).

This function updates `State` and `Laser` according to the set `Smask`.

EXAMPLE

Sensing

KNOWN BUGS

SEE ALSO

**gs, get_sn**

NAME

**conf_tm**

PURPOSE

It sets the timeout period of the robot (in seconds).

SYNTAX

`int conf_tm (int timeout)`

ARGUMENTS

`int timeout` – timeout period in seconds. If 0, the timeout will be disabled. Maximum: 255 seconds

RETURNED VALUE

`1` – success;
`0` – otherwise.

UPDATED GLOBALS

`State` vector;
`Laser` vector.

DESCRIPTION

This function sets the timeout period of the robot (in seconds), 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 the robot does not receive commands from the host.

This function updates `State` and `Laser` according to the set `Smask`.

EXAMPLE

Motion

KNOWN BUGS

SEE ALSO

NAME

**connect_robot**

PURPOSE

It requests the server to connect to the robot.

SYNTAX

`int connect_robot (long robot_id)`

ARGUMENTS

`long robot_id` – robot's identification number.

RETURNED VALUE

`robot_id` – if connection is successful;
`0` – otherwise.

UPDATED GLOBALS

DESCRIPTION

This function requests a connection to the robot with robot_id in the server. In order to talk to the server, you must set `SERVER_MACHINE_NAME` (a `char` array of 80) and `SERV_TCP_PORT` (an `int`) properly in advance if you do not want to use the default values for them. The default value of `SERVER_MACHINE_NAME` is an empty string, which means that the current machine is the server; the default value of `SERV_TCP_PORT` is 7019. A `robot_id` will typically be created by the Nserver. If a robot with `robot_id` exists, a connection is established with that robot. If no robot exists with `robot_id`, no connection is established. Once the connection is established, the subsequent commands are directed to that robot. Before your program sends any command to a robot, it must connect to it.

EXAMPLE

Motion

KNOWN BUGS

SEE ALSO

**disconnect_robot**

NAME

**ct**

PURPOSE

It sends the sensor mask to the robot.

SYNTAX

`int ct (void)`

ARGUMENTS

RETURNED VALUE

`1` – success;
`0` – otherwise.

UPDATED GLOBALS

`State` vector;
`Laser` vector.

DESCRIPTION

The function sends the sensor mask `Smask` to the robot so that only the desirable sensor data are sent back. You must change the global variable `Smask` to the desired communication mask before calling this function.

This function updates `State` and `Laser` according to the set `Smask`.

EXAMPLE

Sensing

KNOWN BUGS

SEE ALSO

NAME

**da**

PURPOSE

It defines the robot's steering and turret angles.

SYNTAX

`int da (int th, int tu)`

ARGUMENTS

`int th` – the steering orientation;
`int tu` – the turret orientation.

RETURNED VALUE

`1` – success;
`0` – otherwise.

UPDATED GLOBALS

`State` vector;
`Laser` vector.

DESCRIPTION

This function defines the robot's steering angle to be `th` and the turret angle to be `tu`. It has no effect on the robot's position. In the simulation mode, the encoder robot and the real robot will be given this configuration. In the real robot mode, the angles are reset without affecting the robot position, and without real motion: The robot internal counters for angles are reset to this value.

This function updates `State` and `Laser` according to the set `Smask`.

EXAMPLE

World

KNOWN BUGS

SEE ALSO

**dp**

NAME

**delete_obstacle**

PURPOSE

It deletes an obstacle from the current robot environment.

SYNTAX

`int delete_obstacle (long obs[21])`

ARGUMENTS

`long obs[21]` – `obs[0]` specifies the number (no greater than 10) of vertices of the polygonal obstacle. `obs[1]` to `obs[20]` specify the $x$ and $y$ coordinates of the vertices, in counter-clockwise direction.

RETURNED VALUE

`1` – success;
`0` – otherwise.

UPDATED GLOBALS

DESCRIPTION

This function deletes an obstacle specified by obs from the current robot environment. The obstacle to delete is identified first by the number of vertices, second by the coordinates. Currently, an obstacle can have at most 10 vertices. `obs[0]` specifies the number of vertices of the polygonal obstacle. `obs[2i+1]` and `obs[2i+2]` specify the $i$th ($i = 0, ..., 9$) vertex of the polygon. The vertices of the polygon must be specified in counter-clockwise direction.

EXAMPLE

World

KNOWN BUGS

SEE ALSO

**add_obstacle, move_obstacle**

NAME

**disconnect_robot**

PURPOSE

It requests the server to close the connection with the robot.

SYNTAX

`int disconnect_robot (long robot_id)`

ARGUMENTS

`long robot_id` – robot's identification number.

RETURNED VALUE

`1` – success;

`0` – otherwise.

UPDATED GLOBALS

DESCRIPTION

This function requests the server to close the connection with robot of `robot_id`.

EXAMPLE

Motion

KNOWN BUGS

SEE ALSO

**connect_robot**

NAME

**dp**

PURPOSE

It defines the position of the robot.

SYNTAX

`int dp (long x, long y)`

ARGUMENTS

`long x, y` – the position coordinates.

RETURNED VALUE

`1` – success;
`0` – otherwise.

UPDATED GLOBALS

`State` vector;
`Laser` vector.

DESCRIPTION

This function defines the robot's position to (`x, y`). It has no effect on the robot's steering and turret coordinates. In the simulation mode, the encoder robot and the real robot will be given this configuration. In the real robot mode, the angles are reset without affecting the robot position, and without real motion: The robot internal counters for angles are reset to this value.

This function updates `State` and `Laser` according to the set `Smask`.

EXAMPLE

World

KNOWN BUGS

SEE ALSO

**da**

NAME

**draw_arc**

PURPOSE

It allows the client to draw an arc, a part of an ellipse, on the robot window.

SYNTAX

```
int draw_arc (long x0, long y0, long w, long h, int th1,
int th2, int mode)
```

ARGUMENTS

`long x0, y0` – specify the upper left corner of the rectangle bounding the ellipse in tens of inches in world coordinates;
`long w` – width of the bounding rectangle in tens of inches;
`long h` – height of the bounding rectangle in tens of inches;
`int th1, th2` – specify the angular range of the arc in tens of degree;
`int mode` – drawing mode

= 1: `BlackPixel` using `GXcopy`;
= 2: `BlackPixel` using `GXxor`;
> 2: color using `GXxor`.

RETURNED VALUE

`1` – success;
`0` – otherwise.

UPDATED GLOBALS

DESCRIPTION

This function allows the client to draw an arc which is a part of an ellipse in the robot window of the currently connected robot. (`x0, y0`) specifies the upper_left corner of the bounding box of the ellipse and (`w, h`) specifies the width and height of the bounding box. (`th1, th2`) specifies the angular range of the arc. If `mode` = 1, the drawing is done in black using GXcopy. If `mode` = 2, the drawing is done in black using `GXxor`. If `mode` > 2, the drawing is done in color using `GXxor`.

EXAMPLE

World

KNOWN BUGS

SEE ALSO

**draw_line, draw_robot**

NAME

**draw_line**

PURPOSE

It allows the client to draw a line.

SYNTAX

`int draw_line (long x1, long y1, long x2, long y2, int mode)`

ARGUMENTS

`long x1, y1` – starting point of the line, tens of inches in world coordinates;
`long x2, y2` – ending point of the line, tens of inches in world coordinates;
`int mode` – drawing mode

= 1: `BlackPixel` using `GXcopy`;
= 2: `BlackPixel` using `GXxor`;
> 2: color using `GXxor`.

RETURNED VALUE

`1` – success;
`0` – otherwise.

UPDATED GLOBALS

DESCRIPTION

This function allows the client to draw a line from (`x1, y1`) to (`x2, y2`) in the robot window of the currently connected robot. If `mode` = 1, the drawing is done in black using `GXcopy`. If `mode` = 2, the drawing is done in black using `GXxor`. If `mode` > 2, the drawing is done in color using `GXxor`.

EXAMPLE

World

KNOWN BUGS

SEE ALSO

**draw_arc, draw_robot**

NAME

**draw_robot**

PURPOSE

It allows the client to draw the shape of a robot.

SYNTAX

`int draw_robot (long x, long y, int th, int tu, int mode)`

ARGUMENTS

`long x, y` – $x$-$y$ position of the robot;
`int th, tu` – steering and turret orientation of the robot;
`int mode` – drawing mode

- = 1: `BlackPixel` using `GXxor`.
- = 2: `BlackPixel` using `GXxor` — a small arrow is drawn at the center of the robot using `GXcopy`.
- = 3: `BlackPixel` using `GXcopy`.
- > 3: color using `GXxor`.

RETURNED VALUE

`1` – success;
`0` – otherwise.

UPDATED GLOBALS

DESCRIPTION

This function allows the client to draw a robot at configuration (`x, y, th, tu`), using the world coordinates. If `mode` = 1, the robot is drawn in black using `GXxor` (using this mode you can erase the trace of robot). If `mode` = 2, the robot is drawn in black using `GXxor` and in addition, a small arrow is drawn at the center of the robot using `GXcopy` (using this mode you can leave a trace of small arrow). If `mode` = 3, the robot is drawn in black using `GXcopy`. If `mode` > 3, the drawing is done in color using `GXxor`.

EXAMPLE

World

KNOWN BUGS

SEE ALSO

**draw_arc, draw_line**

NAME

**extract_receive_buffer_header**

PURPOSE

It extracts the header from the receive buffer.

SYNTAX

`int extract_receive_buffer_header (int *length, unsigned char *serial_number, unsigned char *packet_type, unsigned char *buffer)`

ARGUMENTS

`int *length` – returns the number of chars in the buffer;
`unsigned char serial_number` – returns the serial number assigned to the packet;
`unsigned char packet_type` – returns the type number assigned to the packet;
`unsigned char *buffer` – the receive buffer.

RETURNED VALUE

`*length` – the number of chars in the receive buffer.

UPDATED GLOBALS

DESCRIPTION

This function extracts the header information `length` (the `index`), `serial_number`, and `packet_type` from the beginning of the receive buffer. It is not available in simulation.

EXAMPLE

Packet

KNOWN BUGS

SEE ALSO

**init_receive_buffer, extract????**

NAME

**extract—**

PURPOSE

**extractchar** extracts a one-byte char from the receive buffer.
**extract2byteint** extracts a short int(2 bytes) from the receive buffer.
**extract4byteint** extracts a long int(4 bytes) from the receive buffer.
**extractuchar** extracts an unsigned char from the receive buffer.
**extract2byteuint** extracts an unsigned short int(2 bytes) from the receive buffer.
**extract4byteuint** extracts an unsigned long int(4 bytes) from the receive buffer.
**extractdouble** extracts a double(8 bytes) from the receive buffer.

SYNTAX

```
char extractchar(unsigned char *buffer, int *index)
short extract2byteint(unsigned char *buffer, int *index)
long extract4byteint(unsigned char *buffer, int *index)
unsigned char extractuchar( unsigned char *buffer, int *index)
unsigned short extract2byteuint(unsigned char *buffer, int
*index)
unsigned long extract4byteuint(unsigned char *buffer, int
*index)
double extractdouble(unsigned char *buffer, int *index)
```

ARGUMENTS

`unsigned char *buffer` – the receive buffer which holds the data;
`int *index` – the receive buffer index which will be incremented to reflect the position of the next place of data to be extracted.

RETURNED VALUE

– All the functions return the extracted data of the corresponding type.

UPDATED GLOBALS

DESCRIPTION

All these functions are basically doing the same thing, that is, extracting data of specific types from the receive buffer. They are not available in simulation.

EXAMPLE

Packet

KNOWN BUGS

SEE ALSO

**stuff—**

NAME

**get_bp**

PURPOSE

It gets the bumper data.

SYNTAX

`int get_bp (void)`

ARGUMENTS

RETURNED VALUE

`1` – success;
`0` – otherwise.

UPDATED GLOBALS

`State` vector.

DESCRIPTION

This function gets the bumper data independently of `Smask`. However, data is valid only if the bumper is active (as specified by the previous `conf_bp` function call).

This function updates the `State` vector (state `STATE_BUMPER`).

EXAMPLE

Sensing

KNOWN BUGS

SEE ALSO

**gs**

NAME

**get_cp**

PURPOSE

It gets the compass data of the robot.

SYNTAX

`int get_cp (void)`

ARGUMENTS

RETURNED VALUE

`1` – success;
`0` – otherwise.

UPDATED GLOBALS

`State` vector.

DESCRIPTION

This function gets the compass data of the robot independently of `Smask`. However, data is valid only if the compass is active (as specified by the previous `conf_cp` function call). The orientation is given in 1/10 of degree with magnetic north being 0.

This function updates the `State` vector (state `STATE_COMPASS`).

EXAMPLE

Sensing

KNOWN BUGS

SEE ALSO

**conf_cp, gs**

NAME

**get_ir**

PURPOSE

It gets the infrared data.

SYNTAX

`int get_ir (void)`

ARGUMENTS

RETURNED VALUE

`1` – success;
`0` – otherwise.

UPDATED GLOBALS

`State` vector.

DESCRIPTION

This function gets the infrared data, independently of `Smask`. However, only the active infrared sensor (as specified by the previous `conf_ir` function call) readings are valid.

This function updates the `State` vector (states `STATE_IR_0` to `STATE_IR_15`).

EXAMPLE

Sensing

KNOWN BUGS

SEE ALSO

**conf_ir, gs**

NAME

**get_ls**

PURPOSE

It gets the laser data in *point* mode.

SYNTAX

`int get_ls (void)`

ARGUMENTS

RETURNED VALUE

`1` – success;
`0` – otherwise.

UPDATED GLOBALS

`Laser` vector.

DESCRIPTION

This function gets the laser data, in point mode, independently of `Smask`. The data is determined by the previous `conf_ls` function call. If `conf_ls` was used to set mode to point mode(33), `get_ls` will temporarily set it to 51.

This function updates the `Laser` vector.

EXAMPLE

Sensing

KNOWN BUGS

SEE ALSO

**gs**

NAME

**get_rc**

PURPOSE

It gets the configuration data of the robot.

SYNTAX

`int get_rc (void)`

ARGUMENTS

RETURNED VALUE

`1` – success;
`0` – otherwise.

UPDATED GLOBALS

`State` vector.

DESCRIPTION

This function gets the configuration of the robot including its integrated `x, y, th,` and `tu` coordinates, independent of `Smask`.

This function updates the `State` vector (states `STATE_CONF_X, STATE_CONF_Y, STATE_CONF_STEER, STATE_CONF_TURRET`).

EXAMPLE

Sensing

KNOWN BUGS

SEE ALSO

**get_rv, gs**

NAME

**get_robot_conf**

PURPOSE

It interactively gets a point from robot's window.

SYNTAX

`int get_robot_conf(long *conf)`

ARGUMENTS

`long *conf` – an array of 4 long integers; the configuration of the robot is returned in this array.

RETURNED VALUE

`1` – success;
`0` – otherwise.

UPDATED GLOBALS

DESCRIPTION

This function interactively gets a robot's configuration. When called, a robot shape, augmented with synchronization bars and handles appear in the MAP window:

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

To finish, click in the gray label: bars and handles disappear. The configuration is stored in the array given as argument as follows:

`conf[0]`: X position in tens of inches
`conf[1]`: Y position in tens of inches
`conf[2]`: Steering in tens of degrees
`conf[3]`: Turret orientation in tens of degree

EXAMPLE

World

KNOWN BUGS

SEE ALSO

NAME

**get_rv**

PURPOSE

It gets the translation, steering, turret velocities of the robot.

SYNTAX

```
int get_rv (void)
```

ARGUMENTS

RETURNED VALUE

`1` – success;
`0` – otherwise.

UPDATED GLOBALS

`State` vector.

DESCRIPTION

This function gets the velocities of the robot including its translation, steering and turret rotation velocities, independently of `Smask`.

This function updates the `State` vector (state `STATE_VEL_TRANS`, `STATE_VEL_STEER`, `STATE_VEL_TURRET`).

EXAMPLE

Sensing

KNOWN BUGS

SEE ALSO

**get_rc, gs**

NAME

**get_sg**

PURPOSE

It gets the laser data in *line* mode.

SYNTAX

`int get_sg (void)`

ARGUMENTS

RETURNED VALUE

`1` – success;
`0` – otherwise.

UPDATED GLOBALS

`Laser` vector.

DESCRIPTION

This function gets the laser data, in line mode, independent of `Smask`. The data is determined by the previous `conf_ls` and `conf_sg` function calls.

This function updates the `Laser` vector.

EXAMPLE

Sensing

KNOWN BUGS

SEE ALSO

**conf_ls, conf_sg, gs**

NAME

**get_sn**

PURPOSE

It gets the sonar data.

SYNTAX

`int get_sn (void)`

ARGUMENTS

RETURNED VALUE

`1` – success;
`0` – otherwise.

UPDATED GLOBALS

`State` vector.

DESCRIPTION

This function gets the sonar data, independent of `Smask`. However, only the active sonar sensor (as specified by the previous `conf_sn` function call) readings are valid.

This function updates the `State` vector(states `STATE_SONAR_0` to `STATE_SONAR_15`).

EXAMPLE

Sensing

KNOWN BUGS

SEE ALSO

**conf_sn**, **gs**

NAME

`gs`

PURPOSE

It gets the current state of the robot according to the mask `Smask`.

SYNTAX

```
int gs (void)
```

ARGUMENTS

RETURNED VALUE

`1` – success;
`0` – otherwise.

UPDATED GLOBALS

`State` vector;
`Laser` vector.

DESCRIPTION

This function gets the current state of the robot according to the mask of the communication channel. It simply updates `State` and `Laser`.

`State` and `Smask` values:

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

`Laser` values:

| Mode | Laser[0] | Notes |
|---|---|---|
| Pixel | Number of pixels | 482 pixels maximum |
| Point | Number of points | x-y coordinates in tens of inches, relative to the robot's turret center<br>482 points (964 numbers) maximum |
| Line | Number of lines | x-y coordinates of lines endpoints, in tens of inches, relative to the robot's turret center<br>241 lines maximum |

EXAMPLE

Sensing

KNOWN BUGS

SEE ALSO

**ct, init_sensors, init_mask**

NAME

**init_mask**

PURPOSE

It sets the sensor mask and sends the mask to the robot.

SYNTAX

`void init_mask(void)`

ARGUMENTS

RETURNED VALUE

UPDATED GLOBALS

`Smask` vector.

DESCRIPTION

This function initializes the sensor mask, `Smask`, to 1 everywhere, meaning that every type of data will be returned.

EXAMPLE

Sensing

KNOWN BUGS

SEE ALSO

**ct, init_sensors**

NAME

**init_receive_buffer**

PURPOSE

It sets the index to 4, the point extract data.

SYNTAX

```
int init_receive_buffer(int *index)
```

ARGUMENTS

`int *index` – the buffer index.

RETURNED VALUE

`4` – the last value of *index.

UPDATED GLOBALS

DESCRIPTION

This function sets the index the point at which data should begin to be extracted. It is not available in simulation.

EXAMPLE

Packet

KNOWN BUGS

SEE ALSO

**init_send_buffer, extract_receive_buffer, extract????**

NAME

**init_send_buffer**

PURPOSE

It sets the send buffer.

SYNTAX

```
int init_send_buffer(int *index)
```

ARGUMENTS

`int *index` – the buffer index.

RETURNED VALUE

`4` – the last value of *index.

UPDATED GLOBALS

DESCRIPTION

This function sets the index to the point at which data should be inserted. It is not available in simulation.

EXAMPLE

Packet

KNOWN BUGS

SEE ALSO

**init_receive_buffer, stuff_send_buffer, stuff????**

NAME

**init_sensors**

PURPOSE

It initializes the sensor mask and sends it to the robot.

SYNTAX

`void init_sensors(void)`

ARGUMENTS

RETURNED VALUE

UPDATED GLOBALS

`Smask` vector.

DESCRIPTION

This function initializes the sensor mask, `Smask`, and sends it to the robot. It has no effect on the sensors.

EXAMPLE

Sensing

KNOWN BUGS

SEE ALSO

**ct, init_mask**

NAME

**lp**

PURPOSE

It sets the motor limp.

SYNTAX

```
int lp (void)
```

ARGUMENTS

RETURNED VALUE

`1` – success;
`0` – otherwise.

UPDATED GLOBALS

`State` vector;
`Laser` vector.

DESCRIPTION

This function stops all motors of the robot, that is, the robot will not hold its position; the old accelerations will be restored after the call to this function. This function will return without waiting for the stop to complete. Note that this function might not produce the desired effect if the accelerations are too small.

This function updates `State` and `Laser` according to the set `Smask`.

EXAMPLE

Motion

KNOWN BUGS

SEE ALSO

**st**

NAME

**move_obstacle**

PURPOSE

It moves an obstacle in the robot environment.

SYNTAX

`int move_obstacle (long obs[21], long dx, long dy)`

ARGUMENTS

`long obs[21]` – `obs[0]` specifies the number (no greater than 10) of vertices of the polygonal obstacle. `obs[1]` to `obs[20]` specify the $x$ and $y$ coordinates of the vertices, in counter-clockwise direction.
`long dx, dy` – the $x$ and $y$ distances to translate the obstacle.

RETURNED VALUE

`1` – success;
`0` – otherwise.

UPDATED GLOBALS

DESCRIPTION

This function moves the obstacle specified by `obs` by the distance of `dx` along the $x$-axis and `dy` along the $y$-axis. `Obs[21]` will be modified to reflect the move.

EXAMPLE

World

KNOWN BUGS

SEE ALSO

**create_obstacle, delete_obstacle**

NAME

**mv**

PURPOSE

Move the three axes of the robot independently.

SYNTAX

`int mv(int t_mode, int t_mv, int s_mode, int s_mv, int r_mode, int r_mv)`

ARGUMENTS

`int t_mode` – the control law for the tranlational axis
`int t_mv` – the motion value for the translational axis
`int s_mode` – the control law for the steering axis
`int s_mv` – the motion value for the steering axis
`int r_mode` – the control law for the turret axis
`int r_mv` – the motion value for the turret axis

RETURNED VALUE

`TRUE` – the command was executed successfully
`FALSE` – the command could not be executed or wrong arguments

UPDATED GLOBALS

`State` vector;
`Laser` vector.

DESCRIPTION

The motion commands `vm` (velocity move) and `pr` (position relative) allow to move all of robot's axes by specifying either a velocity or a relative position, respectively. The motion command `mv` (move) allows to specify modes of motion control for each of the axes independently. The values that specify the modes are defined in `Nclient.h`:

`MV_VM`: specifies velocity mode similar to `vm`
`MV_PR`: specifies position mode similar to `pr`
`MV_IGNORE`: ignore the information for this axis
`MV_LP`: set this axis limp
`MV_SP`: set the speed for this axis
`MV_AC`: set the acceleration for this axis

The mode arguments `t_mode`, `s_mode`, and `r_mode` define how the corresponding values `t_mv`, `s_mv`, and `r_mv` are interpreted. In velocity mode they are treated like the arguments to `vm`, as velocities. If position mode is specified they will be interpreted as positions relative to the current configuration. Specifying mode `MV_IGNORE` for an axis will result in that axis to remain in its current state. Refer to the documentation of `pr` and `vm` for detailed information on the value arguments for the corresponding mode.
*Example:* `mv ( MV_VM, 200, MV_PR, 100, MV_IGNORE, MV_IGNORE )` will cause the robot to translate in velocity mode at a velocity of 20 inch per second, to steer 10 degrees, and to continue the previously specified turret motion (if a turret command was issued prior to the `mv`).

EXAMPLE

Motion

KNOWN BUGS

SEE ALSO

**pr, vm**

NAME

**new_world**

PURPOSE

It deletes all obstacles in the current robot world.

SYNTAX

`int new_world(void)`

ARGUMENTS

RETURNED VALUE

`1` – success;
`0` – otherwise.

UPDATED GLOBALS

DESCRIPTION

This function deletes all obstacles in the current robot world.

EXAMPLE

World

KNOWN BUGS

SEE ALSO

**add_obstacle, delete_obstacle**

NAME

**place_robot**

PURPOSE

It places the robot at a certain position.

SYNTAX

```
int place_robot (long x, long y, int th, int tu)
```

ARGUMENTS

`long x, y` – the $x$-$y$ position of the desired robot configuration;
`int th, tu` – the steering and turret orientation of the desired robot configuration.

RETURNED VALUE

`1` – success;
`0` – otherwise.

UPDATED GLOBALS

DESCRIPTION

This function places the robot at position (`x y`), steering at `th` and turret at `tu`. In simulation mode, it will place both the Encoder-robot and the Actual-robot at this configuration. In real robot mode, it will reset the robot's counters to the new values.

EXAMPLE

World

KNOWN BUGS

SEE ALSO

**dp, da**

NAME

**posBumperGet**

PURPOSE

Copy the latest position information for the bumper into a structure provided by the user.

SYNTAX

`int posBumperGet ( PosData *posData )`

ARGUMENTS

`PosData *posData` – a pointer to a PosData structure to be filled with the position information for the bumper.

RETURNED VALUE

`TRUE` – always.

UPDATED GLOBALS

None.

DESCRIPTION

If the user has requested the attachment of position information to the bumper readings by calling `posDataRequest`, this procedure will copy the position attachment for the specified sensor into the structure that is passed as an argument and defined in `Nclient.h`.
*Note:* The position attachment to the bumper is updated only when the bumper is hit.
This command is only available if the EPROM on the Intellisys 100 board is version 2.6.0 or higher. The welcome message of robotd and of the telnet-session to the robot will print out the version of the EPROM.

EXAMPLE

Position Attachment

KNOWN BUGS

SEE ALSO

**posDataRequest, posDataCheck, pos[Sensor]Get**

NAME

**posCompassGet**

PURPOSE

Copy the latest position information for the compass into a structure provided by the user.

SYNTAX

```
int posCompassGet ( PosData *posData )
```

ARGUMENTS

`PosData *posData` – a pointer to a PosData structure to be filled with the position information for the compass.

RETURNED VALUE

`TRUE` – always.

UPDATED GLOBALS

None.

DESCRIPTION

If the user has requested the attachment of position information to the compass reading by calling `posDataRequest`, this procedure will copy the position attachment for the specified sensor into the structure that is passed as an argument and defined in `Nclient.h`.

*Note:* The compass is updated at a rate of 10 Hz.

This command is only available if the EPROM on the Intellisys 100 board is version 2.6.0 or higher. The welcome message of robotd and of the telnet-session to the robot will print out the version of the EPROM.

EXAMPLE

Position Attachment

KNOWN BUGS

SEE ALSO

**posDataRequest, posDataCheck, pos[Sensor]Get**

NAME

**posDataCheck**

PURPOSE

Determine for which sensors the attachment of position data is currently requested.

SYNTAX

`int posDataCheck ( void )`

ARGUMENTS

RETURNED VALUE

`int` – the mask that was the argument to the previous call to posDataRequest

UPDATED GLOBALS

None.

DESCRIPTION

The return value of this function allows the user to determine for which of the sensors the attachment of position data is currently requested. It returns the value that was previously passed to posDataRequest. The user can inquire the status of the position attachment with the macros defined in `Nclient.h`:

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

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

The `printf` statement is only executed if the position attachment is requested for either the infrared sensors or the sonars, or both.
This command is only available if the EPROM on the Intellisys 100 board is version 2.6.0 or higher. The welcome message of robotd and of the telnet-session to the robot will print out the version of the EPROM.

EXAMPLE

Position Attachment

KNOWN BUGS

SEE ALSO

**posDataRequest, pos[Sensor]Get**

NAME

**posDataRequest**

PURPOSE

Request position information to be attached to sensor readings.

SYNTAX

`int posDataRequest ( int posRequest )`

ARGUMENTS

`int posRequest` – a mask in which bits are set for the sensors

RETURNED VALUE

`TRUE` – the argument was a legal mask
`FALSE` – otherwise

UPDATED GLOBALS

`Mask[0]`

DESCRIPTION

With this command the user can request the attachment of position information to sensors readings. Since sensor readings are not instantaneous and the robot's position or the environment can change significantly between acquisition and processing of the sensory data, the sensor reading might be inaccurate when processed. By attaching the position of the robot and a time stamp to every sensor reading, the user is able to account for this source of error.
In `Nclient.h` values are defined for each of the sensors. To request the attachment of position information to sensor readings the logical disjunct of these values is used as an argument. After this procedure has been called the position attachment will be updated automatically until it is turned off again by passing zero as an argument. For example, `posDataRequest ( POS_SONAR | POS_BUMPER )` will request the position information for the sonars and the bumper.
This command is only available if the EPROM on the Intellisys 100 board is version 2.6.0 or higher. The welcome message of robotd and of the telnet-session to the robot will print out the version of the EPROM.

EXAMPLE

Position Attachment

KNOWN BUGS

SEE ALSO

**posDataCheck, pos[Sensor]Get**

NAME

**posInfraredGet**

PURPOSE

Copy the latest position information for one infrared sensor into a structure provided by the user.

SYNTAX

```
int posInfraredGet ( PosData *posData, int infraredNumber )
```

ARGUMENTS

`PosData *posData` – a pointer to a PosData structure to be filled with the position information for the specified infrared sensor.
`int infraredNumber` – the number of the infrared sensor (use defined values from `Nclient.h`)

RETURNED VALUE

`TRUE` – always.

UPDATED GLOBALS

None.

DESCRIPTION

If the user has requested the attachment of position information to the infrared readings by calling `posDataRequest`, this procedure will copy the position attachment for the specified sensor into the structure that is passed as an argument and defined in `Nclient.h`. For example, `posInfraredGet ( POS_IR_1, &posData )` will fill the variable `PosData posData` with the position information for the first infrared sensor.
This command is only available if the EPROM on the Intellisys 100 board is version 2.6.0 or higher. The welcome message of robotd and of the telnet-session to the robot will print out the version of the EPROM.

EXAMPLE

Position Attachment

KNOWN BUGS

SEE ALSO

**posDataRequest, posDataCheck, pos[Sensor]Get**

NAME

**posInfraredRingGet**

PURPOSE

Copy the latest position information for the infrared ring into a structure provided by the user.

SYNTAX

`int posInfraredRingGet ( PosData posDataArray[16] )`

ARGUMENTS

`PosData posData[16]` – an array of sixteen `PosData` structures that upon return of the procedure is filled with the position information for the sixteen infrared sensors.

RETURNED VALUE

`TRUE` – always.

UPDATED GLOBALS

None.

DESCRIPTION

If the user has requested the attachment of position information to the infrared readings by calling `posDataRequest`, this procedure will copy the position attachment for the current readings into the structures that are passed as an argument and defined in `Nclient.h`. For example, `posInfraredRingGet ( posDataArray )` will fill the array defined as `PosData posDataArray[16]` with the position information for all infrared sensors.
This command is only available if the EPROM on the Intellisys 100 board is version 2.6.0 or higher. The welcome message of robotd and of the telnet-session to the robot will print out the version of the EPROM.

EXAMPLE

Position Attachment

KNOWN BUGS

SEE ALSO

**posDataRequest, posDataCheck, pos[Sensor]Get**

NAME

**posLaserGet**

PURPOSE

Copy the latest position information for the laser into a structure provided by the user.

SYNTAX

```
int posLaserGet ( PosData *posData )
```

ARGUMENTS

`PosData *posData` – a pointer to a PosData structure that upon return of the procedure is filled with the position information for the laser.

RETURNED VALUE

`TRUE` – always.

UPDATED GLOBALS

The `PosData` structure passed by the user.

DESCRIPTION

If the user has requested the attachment of position information to the laser reading by calling `posDataRequest`, this procedure will copy the position attachment for the specified sensor into the structure that is passed as an argument and defined in `Nclient.h`.
*Note:* The laser reading is updated at a rate of 30 Hz.
This command is only available if the EPROM on the Intellisys 100 board is version 2.6.0 or higher. The welcome message of robotd and of the telnet-session to the robot will print out the version of the EPROM.

EXAMPLE

Position Attachment

KNOWN BUGS

SEE ALSO

**posDataRequest, posDataCheck, pos[Sensor]Get**

NAME

**posSonarGet**

PURPOSE

Copy the latest position information for one sonar into a structure provided by the user.

SYNTAX

`int posSonarGet ( PosData *posData, int sonarNumber )`

ARGUMENTS

`PosData *posData` – a pointer to a PosData structure to be filled with the position information for the specified sonar.
`int sonarNumber` – the number of the sonar (use defined values from `Nclient.h`)

RETURNED VALUE

`TRUE` – always.

UPDATED GLOBALS

None.

DESCRIPTION

If the user has requested the attachment of position information to the sonar readings by calling `posDataRequest`, this procedure will copy the position attachment for the specified sensor into the structure that is passed as an argument and defined in `Nclient.h`. For example, `posSonarGet ( POS_SONAR_1, &posData )` will fill the variable `PosData posData` with the position information for the first sonar sensor.
This command is only available if the EPROM on the Intellisys 100 board is version 2.6.0 or higher. The welcome message of robotd and of the telnet-session to the robot will print out the version of the EPROM.

EXAMPLE

Position Attachment

KNOWN BUGS

SEE ALSO

**posDataRequest, posDataCheck, pos[Sensor]Get**

NAME

**posSonarRingGet**

PURPOSE

Copy the latest position information for the sonar ring into a structure provided by the user.

SYNTAX

`int posSonarRingGet ( PosData posDataArray[16] )`

ARGUMENTS

`PosData posData[16]` – an array of sixteen `PosData` structures to be filled with the position information for the sixteen sonars.

RETURNED VALUE

`TRUE` – always.

UPDATED GLOBALS

None.

DESCRIPTION

If the user has requested the attachment of position information to the sonar readings by calling `posDataRequest`, this procedure will copy the position attachment for the current readings into the structures that are passed as an argument and defined in `Nclient.h`. For example, `posSonarRingGet ( posDataArray )` will fill the array defined as `PosData posDataArray[16]` with the position information for all sonars.
This command is only available if the EPROM on the Intellisys 100 board is version 2.6.0 or higher. The welcome message of robotd and of the telnet-session to the robot will print out the version of the EPROM.

EXAMPLE

Position Attachment

KNOWN BUGS

SEE ALSO

**posDataRequest, posDataCheck, pos[Sensor]Get**

NAME

**posTimeGet**

PURPOSE

Get the current time for the position attachment.

SYNTAX

`int posTimeGet ( void )`

ARGUMENTS

RETURNED VALUE

`int` – The time of the Intellisys 100 sensor-board in ticks.

UPDATED GLOBALS

None.

DESCRIPTION

If the user has requested the attachment of position information to some sensor reading by calling `posDataRequest`, this procedure will return the current time in ticks of the Intellisys 100 sensor-board. The timestamps of the configuration and the position attachment are in Intellisys 100 time. The clock starts when the robot is turned on and rolls over every 19.54 hours.

*Note:* The value returned by this function does not necessarily correspond to the current time on the Intellisys 100. The internal representation of the Intellisys 100 time is updated by each robot command that updates the state vector and by commands that read the sensor readings of infrared sensors, sonars, bumper, or compass. After lengthy computations or after reading input from the console this value will not correspond to the actual Intellisys 100 time. However, by issuing, for example, a `gs()` command the user can force the time value to be updated.

This command is only available if the EPROM on the Intellisys 100 board is version 2.6.0 or higher. The welcome message of robotd and of the telnet-session to the robot will print out the version of the EPROM.

EXAMPLE

Position Attachment

KNOWN BUGS

SEE ALSO

**posDataRequest, posDataCheck, pos[Sensor]Get**

NAME

**pr**

PURPOSE

It moves the motors of the robot by a distance, using the speed set by `sp()`.

SYNTAX

`int pr (int tpr, int spr, int rpr)`

ARGUMENTS

`int tpr` – translational step in 1/10 inches, within [-32000, 32000];
`int spr` – steering step in 1/10 degrees, within [-32000, 32000];
`int rpr` – rotational step in 1/10 degrees, within [-32000, 32000].

RETURNED VALUE

`1` – success;
`0` – otherwise.

UPDATED GLOBALS

`State` vector;
`Laser` vector.

DESCRIPTION

This function moves the robot's translational, steering, and rotational motors by (`tpr`/10) inches, (`spr`/10) degrees, and (`rpr`/10) degrees, respectively at the speeds specified by the previous function call to `sp(tsp, ssp, rsp)` and at accelerations by the previous call to `ac(...)`. The three parameters specify the relative distances for the three motors: translation, steering, and turret. All the three motors move concurrently if their speeds are not set to zero and the distances to be travelled are not zero. Depending on the timeout period set (by function `conf_tm (timeout)` ), the motion may terminate before the robot has moved the desired distance.

*Important note*: The robot controller cannot execute a `pr` instruction directly after a `vm`. Any `pr` command directly following a `vm` command *will not* be executed.
To execute `pr`, you have to get the robot to a complete stop, as in the following example:

```
vm(100,0,0);
sleep(2); /* move for 2 seconds */
st(); /* stop the robot */
ws(); /* wait for stop to be complete */
pr(0,900,0); /* the pr can now be executed */
```

This function updates `State` and `Laser` according to the set `Smask`.

EXAMPLE

Motion

KNOWN BUGS

SEE ALSO

**vm, mv**

NAME

**quit_server**

PURPOSE

It causes the server to quit.

SYNTAX

```
int quit_server (void)
```

ARGUMENTS

RETURNED VALUE

`1` – success;
`0` – failure.

UPDATED GLOBALS

DESCRIPTION

This function causes the server to quit assuming this feature is enabled in the setup file of the server.

EXAMPLE

Motion

KNOWN BUGS

This function works only with ONE client program; it has the side effect of killing other clients connected to the server and cannot get the returned value 1.

SEE ALSO

NAME

**real_robot**

PURPOSE

It switches the server to the `real_robot` mode.

SYNTAX

```
int real_robot (void)
```

ARGUMENTS

RETURNED VALUE

`1` – success;
`0` – otherwise.

UPDATED GLOBALS

DESCRIPTION

This function switches the server to the `real_robot` mode. All the commands will be directed to the real robot.

CAUTION: Make sure that the robot is in a safe position before switching to this mode.

EXAMPLE

Motion

KNOWN BUGS

You cannot call this function when the server is already in the `real_robot` mode.

SEE ALSO

NAME

**server_is_running**

PURPOSE

It queries the server to see if it is up and running.

SYNTAX

`int server_is_running (void)`

ARGUMENTS

RETURNED VALUE

`1` – the server is running;
`0` – the server is not running.

UPDATED GLOBALS

DESCRIPTION

This function queries the server to see if it is up and running.

EXAMPLE

Motion

KNOWN BUGS

SEE ALSO

**quit_server**

NAME

**simulated_robot**

PURPOSE

It switches the server to the `simulated_robot` mode.

SYNTAX

```
int simulated_robot (void)
```

ARGUMENTS

RETURNED VALUE

`1` – success;
`0` – otherwise.

UPDATED GLOBALS

DESCRIPTION

This function switches the server to the `simulated_robot` mode.

EXAMPLE

Motion

KNOWN BUGS

You cannot call this function when the server is already in the `simulated_robot` mode.

SEE ALSO

**real_robot**

NAME

**sp**

PURPOSE

It sets the translation, steering, and rotation speeds of the robot.

SYNTAX

`int sp (unsigned int tsp, unsigned int ssp, unsigned int rsp)`

ARGUMENTS

`int tsp` – the translation speed in 1/10 inch/sec, within [0, 200].
`int ssp` – the steering speed in 1/10 degree/sec, within [0, 450].
`int rsp` – the turret speed in 1/10 degree/sec, within [0, 450].

RETURNED VALUE

`1` – success;
`0` – otherwise.

UPDATED GLOBALS

`State` vector;
`Laser` vector.

DESCRIPTION

This function sets the translation, steering, and rotation speeds of the robot to `tsp, ssp`, and `rsp` respectively. The speeds are initially set to 200, 300 and 300 for tsp, ssp, and rsp respectively. *Note*: The defaults can be found in the file `motor.cfg` on the robot or in the file `robot.setup` on the simulated robot.

This function updates `State` and `Laser` according to the set `Smask`.

EXAMPLE

Motion

KNOWN BUGS

SEE ALSO

**ac**

NAME

**special_request**

PURPOSE

It sends a user packet to the robot and waits for response.

SYNTAX

```
int special_request (char *user_send_buffer, char
*user_receive_buffer)
```

ARGUMENTS

`char *user_send_buffer` – stores data to be sent to the robot;

`char *user_receive_buffer` – stores data received from the robot.

RETURNED VALUE

`1` – success;

`0` – otherwise.

UPDATED GLOBALS

DESCRIPTION

This function sends a user packet (stored in `user_send_buffer`) to the robot and waits for the robot's response (which will be stored in `user_receive_buffer`). The maximum packet size is 1k bytes. This function is to be used for implementing user defined communication between the robot and the host.

EXAMPLE

Packet

KNOWN BUGS

SEE ALSO

**init_send_buffer, stuff_send_buffer, stuff????**

**init_receive_buffer, extract_receive_buffer, extract????**

NAME

**st**

PURPOSE

It stops the motion of the robot.

SYNTAX

`int st (void)`

ARGUMENTS

RETURNED VALUE

`1` – success;
`0` – otherwise.

UPDATED GLOBALS

`State` vector;
`Laser` vector.

DESCRIPTION

This function brings the robot to a *controlled* stop with appropriate accelerations and holds its current position. If acceleration = 0, the robot will NOT stop.

This function updates `State` and `Laser` according to the set `Smask`.

EXAMPLE

Motion

KNOWN BUGS

SEE ALSO

**lp, ws**

NAME

**stuff_send_buffer_header**

PURPOSE

It loads the header into the send buffer.

SYNTAX

`int stuff_send_buffer_header (short index, unsigned char serial_number, unsigned char packet_type, unsigned char *buffer)`

ARGUMENTS

`short index` – the buffer index which holds the number of chars in the buffer;
`unsigned char serial_number` – the serial number to be assigned to the packet;
`unsigned char packet_type` – the type number to be assigned to the packet;
`unsigned char *buffer` – the send buffer.

RETURNED VALUE

`index` – the same as the parameter.

UPDATED GLOBALS

DESCRIPTION

This function loads the header information length (the `index`),`serial_number`, and `packet_type` into the beginning of the buffer. It should be called after the data has been stuffed. It is not available in simulation.

EXAMPLE

Packet

KNOWN BUGS

SEE ALSO

**all the communication functions**

NAME

**stuff—**

PURPOSE

**stuffchar** stuffs a one-byte char into the send buffer.
**stuff2byteint** stuffs a short int(2 bytes) into the send buffer.
**stuff4byteint** stuffs a long int(4 bytes) into the send buffer.
**stuffuchar** stuffs an unsigned char into the send buffer.
**stuff2byteuint** stuffs an unsigned short int(2 bytes) into the send buffer.
**stuff4byteuint** stuffs an unsigned long int(4 bytes) into the send buffer.
**stuffdouble** stuffs a double(8 bytes) into the send buffer.

SYNTAX

```
int stuffchar(char data, unsigned char *buffer, int *index)
int stuff2byteint(short data, unsigned char *buffer, int *index)
int stuff4byteint(long data, unsigned char *buffer, int *index)
int stuffuchar(unsigned char data, unsigned char *buffer, int
*index)
int stuff2byteuint(unsigned short data, unsigned char *buffer,
int *index)
int stuff4byteuint(unsigned long data, unsigned char *buffer,
int *index)
int stuffdouble(double data, unsigned char *buffer, int *index)
```

ARGUMENTS

`char data` – the char to be stuffed.
`short data` – the value which will be split apart and stuffed bytewise into the send buffer.
`long data` – the value which will be split apart and stuffed bytewise into the send buffer.
`unsigned char data` – the char to be stuffed.
`unsigned short data` – the value which will be split apart and stuffed bytewise into the send buffer.
`unsigned long data` – the value which will be split apart and stuffed bytewise into the send buffer.
`double data` – the value which will be split apart and stuffed bytewise into the send buffer.
`unsigned char *buffer` – the send buffer.
`int *index` – the buffer index which will be incremented to reflect the bytes stuffed into the buffer.

RETURNED VALUE

– All the functions return the last value of `*index`.

UPDATED GLOBALS

DESCRIPTION

All these functions are basically doing the same thing, that is, putting data *bytewise* into the send buffer.

EXAMPLE

Packet

KNOWN BUGS

SEE ALSO

**extract**—

NAME

**tk**

PURPOSE

It sends a character stream to the robot's voice synthesizer.

SYNTAX

`int tk (char *talk_stream)`

ARGUMENTS

`char *talk_stream` – the character stream to be sent to the synthesizer.

RETURNED VALUE

`1` – success;
`0` – otherwise.

UPDATED GLOBALS

`State` vector;
`Laser` vector.

DESCRIPTION

This function sends a talk stream in characters to the robot's voice synthesizer to let the robot talk. This function updates `State` and `Laser` according to the set `Smask`.

EXAMPLE

Motion

KNOWN BUGS

It does not accept non-printable chars.

SEE ALSO

NAME

**voltCpuGet**

PURPOSE

Determine the voltage supplied to the CPU.

SYNTAX

`float voltCpuGet ( void )`

ARGUMENTS

RETURNED VALUE

`float` – the voltage of the CPU battery

UPDATED GLOBALS

None.

DESCRIPTION

This command returns the voltage that is supplied to the CPU. The returned value does not necessarily coincide with the voltage of the battery. If the voltage of the CPU reaches 10.8 volts the batteries have to be recharged, otherwise damage to the batteries will result.
*Note:* This command is only available if the EPROM on the Intellisys 100 board is version 2.6.0 or higher. The welcome message of robotd and of the telnet-session to the robot will print out the version of the EPROM.

EXAMPLE

KNOWN BUGS

SEE ALSO

**voltMotorGet**

NAME

**voltMotorGet**

PURPOSE

Determine the voltage of the motor batteries.

SYNTAX

`float voltMotorGet ( void )`

ARGUMENTS

RETURNED VALUE

`float` – the voltage of the motor batteries

UPDATED GLOBALS

None.

DESCRIPTION

This command returns the voltage of the motor batteries. If this value reaches 10.8 volts the batteries have to be recharged, otherwise damage to the batteries will result.

*Note:* This command is only available if the EPROM on the Intellisys 100 board is version 2.6.0 or higher. The welcome message of robotd and of the telnet-session to the robot will print out the version of the EPROM.

EXAMPLE

KNOWN BUGS

SEE ALSO

**voltCpuGet**

NAME

**vm**

PURPOSE

It moves the robot according to the velocities specified by its parameters.

SYNTAX

`int vm (int tv, int sv, int rv)`

ARGUMENTS

`int tv` – the desired translation velocity in 1/10 inch/sec, within [-240,240];

`int sv` – the desired steering velocity in 1/10 degree/sec, within [-450,450];

`int rv` – the desired turret velocity in 1/10 degree/sec, within [-450,450].

RETURNED VALUE

`1` – success;
`0` – otherwise.

UPDATED GLOBALS

`State` vector;
`Laser` vector.

DESCRIPTION

This function moves the robot at translational velocity `tv`, steering velocity `sv`, and rotational velocity `rv`. `tv` is an integer between -240 and 240 (in units of 0.1 inches/sec). `sv` and `rv` are both integers between -450 and 450 (in units of 0.1 degrees/sec).

The robot will continue to move at these velocities until either it receives another command or it receives no command after timeout (in which case it will stop its motion).

*Important note*: The robot controller cannot execute a `pr` instruction directly after a `vm`. Any `pr` command directly following a `vm` command *will not* be executed.

To execute `pr`, you have to get the robot to a complete stop, as in the following example:

```
vm(100,0,0);
sleep(2) /* move for 2 seconds */
st(); /* stop the robot */
ws(1,0,0,10); /* wait for stop to be complete */
pr(0,900,0); /* the pr can now be executed */
```

This function updates `State` and `Laser` according to the set `Smask`.

EXAMPLE

Motion

KNOWN BUGS

SEE ALSO

**pr, mv**

NAME

**ws**

PURPOSE

It waits for the stop of the motors of the robot.

SYNTAX

`int ws (unsigned char w_t, unsigned char w_s, unsigned char w_r, unsigned char timeout)`

ARGUMENTS

`unsigned char w_t, w_s, w_r` – These three parameters specify which axis or combination of axes (translation, steering, and turret) to wait: 1 for wait and 0 for not;

`unsigned char timeout` – specifies how long (in seconds) to wait before timing out.

RETURNED VALUE

`1` – success;

`0` – otherwise.

UPDATED GLOBALS

`State` vector;

`Laser` vector.

DESCRIPTION

The function waits for the stop of the motors of the robot. Which motor(s) to wait depends on which of the parameters: `w_t`, `w_s`, and `w_r`, are set. This function is intended to be used in conjunction with `pr`() or `st` to detect the desired motion has finished.

*Note*: contrary to the standard behavior, this command will only return after the motors of the robot have actually stopped (usually commands return immediately regardless of whether the robot has completed the desired action or not).

This function updates `State` and `Laser` according to the set `Smask`.

EXAMPLE

Motion

KNOWN BUGS

SEE ALSO

**st**

NAME

**zr**

PURPOSE

It zeroes the robot – aligns steering and turret with bumper zero.

SYNTAX

`int zr (void)`

ARGUMENTS

RETURNED VALUE

`1` – success;
`0` – otherwise.

UPDATED GLOBALS

`State` vector;
`Laser` vector.

DESCRIPTION

This function automatically aligns the forward direction of the steering wheels and the turret with the forward direction of the bumper and sets the aligned steering and turret angles as 0 as well as the position coordinates $(x, y)$ to $(0, 0)$. It will wait 1 to 1.5 minutes for zeroing to complete.

*Note*: contrary to the standard behavior, this command will only return after the zeroing has been completed (usually commands return immediately regardless of whether the robot has completed the desired action or not).

This function updates `State` and `Laser` according to the set `Smask`.

EXAMPLE

Motion

KNOWN BUGS

SEE ALSO

# Chapter 2

# Programming Examples

## 2.1 Motion example

```
/*
 * This program will connect to the robot, configure locomotion,
 * and move the robot using various commands.
 * It assumes that a server is running and connects to it
 *
 * To compile: gcc -o motiontest motion.c Nclient-linux.o -DSIMULATION=1   OR
 *             gcc -o motiontest motion.c Nclient-sparc.o -DSIMULATION=1
 *
 */
#include "Nclient.h"

#define ROBOT_ID 1 /* Currently, only robot #1 allowed */

#ifndef SIMULATION
#define SIMULATION 1
#endif

int main()


  /* Connection */

  SERV_TCP_PORT = 7019; /* Matches the number given in world.setup */
  strcpy(SERVER_MACHINE_NAME, "masai"); /* The machine the server is running on */

  if (!connect_robot(ROBOT_ID))

    printf("Connexion to robot failed\n");
    return(0);


  if (SIMULATION)
    simulated_robot(); /* Commands will be sent to simulator */
  else
    real_robot; /* Commands will be sent to real robot (CAUTION!!) */

  conf_tm(2); /* Robot will stop if no command from the server in 2 seconds */


  printf("         ************************************************************\n");
  printf("         * NOMADIC HOST SOFTWARE ENVIRONMENT - MOTION DEMONSTRATION *\n");
  printf("         ************************************************************\n\n\n");

  /* Initialize the robot */

  printf("Zeroing...\n");

  zr(); /* Zero the robot */

  /* Initialize motion parameters */

  ac(200,300,300); /* translation, steering, turret accelerations in .1in/s2, .1deg/s2 */
  sp(100,200,100); /* translation, steering, turret speeds in .1in/s, .1deg/s */

  printf("Hit any key to translate blindly by 1000 tens of inch... will wait for the motion to stop\n");
```

```
getchar();

pr(1000,0,0); /* x, y, theta */
ws(1,0,0,20);  /* Wait for the motion to stop */

printf("Hit any key to steer blindly by 1800 tens of degree... will wait for the motion to stop\n");
getchar();

pr(0,1800,0); /* x, y, theta */
ws(0,1,0,50);  /* Wait for the motion to stop */

printf("Hit any key to rotate blindly by 900 tens of degree... will wait for the motion to stop\n");
getchar();

pr(0,0,900); /* x, y, theta */
ws(0,0,1,20);  /* Wait for the motion to stop */

printf("Hit any key to translate blindly by 1000 tens of inches, while rotating the turret at speed 300...will wait for the translation to stop\n");
getchar();
mv(MV_PR, 1000, MV_IGNORE, 0, MV_VM, 300);
ws(1,0,0,20);  /* Wait for the translation motion to stop */
vm(0,0,0);                    /* Stop everything */
ws(0,0,1,20);                 /* Wait for the turret to stop */


printf("Hit any key to move in straight line until hitting something...\n");
if (SIMULATION)
  printf("Make sure that there is an obstacle in front of the robot, or create it NOW !\n");
 getchar();

gs(); /* Get state according to Smask */
while(!State[33]) /* Check bumpers */
  vm(100,0,0); /* If ok, velocity move at translation, steering, turret speed (vm updates State vector) */
st(); /* Stop the robot, hold the position */
sleep(2); /* Wait (ws would do as well) */

printf("Hit any key to backup a little bit...\n");
getchar();

pr(-500,0,0); /* position relative */
sleep(2);
get_rv(); /* Get current velocities, independently of the mask */
if (State[38])
  tk("ALERT! The robot did not stop in time !"); /* This robot talks !! */
lp(); /* Stop the robot without holding the position */

printf("End of demo, quitting the server...\n");

quit_server();          /* Kills the server; just disconnect_robot if the server is to be used again */
return(1);
```

## 2.2 Sensing example

```
/*
 * This program will connect to the robot, configure sensing,
 * get sensor data, print and draw it.
 * It assumes that a server is running and connects to it
 *
 * To compile: gcc -o sensingtest sensing.c Nclient-linux.o -DSIMULATION=1 -DALL_SENSORS=0 -lm   OR
 *             gcc -o sensingtest sensing.c Nclient-sparc.o -DSIMULATION=1 -DALL_SENSORS=0 -lm
 *
 */
#include <stdio.h>
#include <math.h>
#include "Nclient.h"

#define ROBOT_ID 1 /* Currently, only robot #1 allowed */

#define PI 3.1415

#ifndef SIMULATION
#define SIMULATION 1
#endif

#ifndef ALL_SENSORS
#define ALL_SENSORS 0
#endif

int main()


  int x0, y0;
  double x1,y1,x2,y2;
  double th, tu;
  int xw1, yw1, xw2, yw2;
  int LineNumber;
  int i;

  /* Sensing configuration */

  int ir_order[16] = 0,2,4,6,8,12,14,255; /* Use every other infrared sensor */
  int sn_order[16] = 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15; /* Use all sonars, in order */

  /* Connection */

  SERV_TCP_PORT = 7019; /* Matches the number given in world.setup */
  strcpy(SERVER_MACHINE_NAME, "masai"); /* The machine the server is running on */

  if (!connect_robot(ROBOT_ID))

    printf("Connection to robot failed\n");
    return(0);


  if (SIMULATION)
    simulated_robot(); /* Commands will be sent to simulator */
  else
    real_robot(); /* Commands will be sent to real robot (CAUTION!!) */
```

```
/* Initialize sensing parameters */

conf_ir(0, ir_order); /* Infrared: history dependence, order */
conf_sn(2, sn_order); /* Sonar: firing interval, order */
conf_cp(1); /* Compass: mode */
conf_ls(33, 10, 20, 20, 0); /* Laser: mode, threshold, width, sampling points, processing */
conf_sg(50, 2, 30); /* Laser line mode: threshold, min points, gap */

/* Initialize sensor mask */

if (ALL_SENSORS)
  init_sensors(); /* Set sensor mask to 1 everywhere, send it to robot */
else

  init_mask(); /* Set sensor mask to 1 everywhere */
  Smask[43]=0; /* Do not want compass */
  ct(); /* Send the mask to the robot */


/* Get sensor data */

printf("        *************************************************************\n");
printf("        * NOMADIC HOST SOFTWARE ENVIRONMENT - SENSING DEMONSTRATION *\n");
printf("        *************************************************************\n\n\n");

if (SIMULATION)

  printf("Make sure that you have obstacles around, or get some NOW\n");
  printf("Then hide map in robot window, and move the robot to some interesting place\n");

printf("Hit any key to get sensor data...\n");
getchar();

if (ALL_SENSORS)
  gs(); /* Get sensor data according to Smask */
else

  get_ir(); /* To get infrared data independently of Smask */
  get_sn(); /* To get sonar data independently of Smask */
  get_bp(); /* To get bumper data independently of Smask */
  get_cp(); /* To get compass data independently of Smask */
  get_ls(); /* To get laser independently of Smask */
  get_sg(); /* To get laser line data independently of Smask */
  get_rc(); /* To get configuration data independently of Smask */
  get_rv(); /* To get velocity data independently of Smask */
  get_cp();                   /* To get compass data independently of Smask */


/* Print configuration infrared, sonar, bumper data on screen */

printf("Infrared data:");
for (i=0;i<16;i++)
  printf("%d ", State[STATE_IR_0+i]);
```

```
printf("\n");

printf("Sonar data:");
for (i=0;i<16;i++)
  printf("%d ", State[STATE_SONAR_0+i]);

printf("\n");

printf("Bumper data:");

for ( i = 0 ; i < 20 ; i ++ )
  if (State[STATE_BUMPER] & ( 1L << i ) )
    printf("1");
  else
    printf("0");

printf("\n");

printf("X Pos: %d  Steer Pos: %d\n", State[STATE_CONF_X], State[STATE_CONF_STEER]);
printf("Y Pos: %d  Turret Pos: %d\n", State[STATE_CONF_Y], State[STATE_CONF_TURRET]);
printf("Trans Speed: %d Steer Speed: %d Turret Speed: %d\n", State[STATE_VEL_TRANS], State[STATE_VEL_STEER], State[STATE_VEL_TURRET]);


/* Draw configuration and laser data on the robot window */

x0 = State[STATE_CONF_X]; /* Current position */
y0 = State[STATE_CONF_Y];
th = (double)State[STATE_CONF_STEER]; /* Current steer */
tu = (double)State[STATE_CONF_TURRET] *PI/1800.0; /* Current turret orientation */

LineNumber = Laser[0];

for (i=1; i<4*LineNumber;i=i+4)

  /* Get endpoints coordinates */
  x1 = (double)Laser[i];
  y1 = (double)Laser[i+1];
  x2 = (double)Laser[i+2];
  y2 = (double)Laser[i+3];

  /* Map these endpoints in the world coordinates system */

  xw1 = x0 + (int)(cos(tu)*x1 - sin(tu)*y1);
  yw1 = y0 + (int)(sin(tu)*x1 + cos(tu)*y1);

  xw2 = x0 + (int)(cos(tu)*x2 - sin(tu)*y2);
  yw2 = y0 + (int)(sin(tu)*x2 + cos(tu)*y2);

  draw_line(xw1,yw1,xw2,yw2,3);   /* Coordinates, color */


printf("End of demonstration, disconnecting...\n");

disconnect_robot(ROBOT_ID);
return(1);
```

## 2.3 World example

```
/*
 * This program will connect to a robot, illustrate obstacle manipulation
 * functions, and interactively get a new position for the robot
 * It assumes that a server is running and connects to it
 *
 * To compile: gcc -o worldtest world.c Nclient-linux.o    OR
 *             gcc -o worldtest world.c Nclient-sparc.o
 *
 */

#include "Nclient.h"

#define ROBOT_ID 1 /* Currently, only robot #1 allowed */
#define RANGE 500

int main()

  /* Obstacle definition: number of vertices, coordinates */
  long obs[21]=3,400,-100,700,-100,500,500;
  long conf[4];
  int id;

  /* Connection */

  SERV_TCP_PORT = 7019; /* Matches the number given in world.setup */
  strcpy(SERVER_MACHINE_NAME, "masai"); /* The machine the server is running on */

  if (!connect_robot(ROBOT_ID))

    printf("Connexion to robot failed\n");
    return(0);


  printf("         ************************************************************************\n");
  printf("         * NOMADIC HOST SOFTWARE ENVIRONMENT - WORLD MANIPULATION DEMONSTRATION *\n");
  printf("         ************************************************************************\n\n\n");


  new_world(); /* Clear the map */

  printf("Hit any key to add a newly created obstacle\n");
  getchar();
  add_obstacle(obs); /* Add this obstacle */

  printf("Hit any key to translate this obstacle\n");
  getchar();
  move_obstacle(obs, 200, 200); /* Move it !! */

  printf("Hit any key to delete this obstacle\n");
  getchar();
  delete_obstacle(obs);

  printf("Position the robot in Robot window with the mouse: \n");
  printf("Click LEFT to drop the robot in place\n");
  printf("Drag handles to rotate/steer the robot\n");
```

```
printf("Click on sync bars to re-align steering and turret\n");
printf("Click on the gray square to exit\n");
get_robot_conf(conf);
printf("Setting position to %d, %d, steer to %d, turret to %d\n", conf[0], conf[1], conf[2], conf[3]);

draw_robot(conf[0], conf[1], conf[2], conf[3], 2); /* Draw the future position of the robot on the robot window */
draw_arc(conf[0]-RANGE, conf[1]+RANGE, 2*RANGE, 2*RANGE, 0, 3600, 2); /* Draw safety range */

printf("Hit any key to reset robot encoders\n");
getchar();

/* Reset robot encoders */
dp(conf[0], conf[1]); /* x, y */
da(conf[2], conf[3]); /* steering, turret */

printf("End of demo, disconnecting...\n");

disconnect_robot(ROBOT_ID);

return(1);
```

## 2.4 Packet example

```
/*
 * This program illustrates the use of special requests.
 * A first, very generic example simply sends numbers to the robot as
 * special requests.
 * The second example will get back the "raw" infrared data (a long integer)
 * by sending a special request.
 * Note that this raw data measures the reflected
 * intensity, thus is larger when the object is closer
 *
 * To compile: gcc -o packettest packet.c Nclient.o   OR
 *             gcc -o packettest packet.c Ndirect.o
 *
 * The packet returned will be the packet sent, except that
 * it is bytewise multiplied by two. Therefore you should
 * not expect the returned package to look like the one you
 * sent.
 */

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

#define ROBOT_ID 1

#define CHAR_TYPE        0
#define INT2_TYPE        1
#define UINT2_TYPE       2
#define INT4_TYPE        3
#define UINT4_TYPE       4
#define DOUBLE_TYPE      5
#define SPECIAL_IR_TYPE  10       /* This is the special request code */

int main( int argc, char **argv )

  short i, b_index, b_length;
  unsigned char serial_number;
  unsigned char packet_type;
  unsigned char user_send_buffer[1024];
  unsigned char user_receive_buffer[1024];

  /* Establish connection */

  SERV_TCP_PORT = 7019;                       /* Matches world.setup         */
  strcpy( SERVER_MACHINE_NAME, "masai"     ); /* machine for server(Nclient) */
  strcpy( ROBOT_MACHINE_NAME, "128.1.1.200"); /* name of the robot (Ndirect) */

  if ( !connect_robot( ROBOT_ID ) )

    printf( "Connection to robot failed\n" );
    return( 0 );


  printf( "*******************************************************\n"   );
  printf( "* NOMADIC HOST SOFTWARE ENVIRONMENT - USER PACKET DEMO  *\n"   );
  printf( "*******************************************************\n\n" );
```

```
  /*
   * First example: Send arbitrary numbers
   */

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

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

/* can be any one of the following :
  stuffchar(123, user_send_buffer, &b_index);
  stuff2byteint(-1234, user_send_buffer, &b_index);
  stuff2byteuint(1234, user_send_buffer, &b_index);
  stuff4byteint(-1234567, user_send_buffer, &b_index);
  stuff4byteuint(1234567, user_send_buffer, &b_index);
  stuffdouble(123.321, user_send_buffer, &b_index);
  stuffdouble(-123.321, user_send_buffer, &b_index);
*/

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

/* can be any one of the following :
  stuff_send_buffer_header(b_index,0,CHAR_TYPE,user_send_buffer);
  stuff_send_buffer_header(b_index,0,INT2_TYPE,user_send_buffer);
  stuff_send_buffer_header(b_index,0,UINT2_TYPE,user_send_buffer);
  stuff_send_buffer_header(b_index,0,INT4_TYPE,user_send_buffer);
  stuff_send_buffer_header(b_index,0,UINT4_TYPE,user_send_buffer);
  stuff_send_buffer_header(b_index,0,DOUBLE_TYPE,user_send_buffer);
  stuff_send_buffer_header(b_index,0,DOUBLE_TYPE,user_send_buffer);
*/

  /* send the buffer */
  printf( "Sending   : 0x10101010.\n" );
  special_request( user_send_buffer, user_receive_buffer );

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

/* can be any one of the following :
  printf ( "char: %d\n",   extractchar     ( user_receive_buffer, &b_index ));
  printf ( "short: %d\n",  extract2byteint ( user_receive_buffer, &b_index ));
  printf ( "ushort: %u\n", extract2byteuint( user_receive_buffer, &b_index ));
  printf ( "long: %ld\n",  extract4byteint ( user_receive_buffer, &b_index ));
  printf ( "ulong: %lu\n", extract4byteuint( user_receive_buffer, &b_index ));
  printf ( "double: %f\n", extractdouble   ( user_receive_buffer, &b_index ));
  printf ( "double: %f\n", extractdouble   ( user_receive_buffer, &b_index ));
*/

  /*
```

```
 * Second example: how to get back infrared data
 */

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

/* fill buffer with special request */
stuff_send_buffer_header ( b_index, 0, SPECIAL_IR_TYPE, user_send_buffer );
printf ( "Requesting IR data.\n" );

/* send buffer */
special_request ( user_send_buffer, user_receive_buffer );

/* process received_buffer */
init_receive_buffer ( &b_index );
extract_receive_buffer_header ( &b_length, &serial_number, &packet_type,
user_receive_buffer);
/* display data */
printf ( "Received IR data: \n" );
for ( i = 1; i <= 16; i++ )
  printf ( "%2d:  %lu \n", i,
  extract4byteuint( user_receive_buffer, &b_index ));
/*
 * Third example: send a buffer of chars
 */

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

/* fill buffer */
printf ( "Sending   : " );
for ( i = 0; i < 10; i++ )

  stuffchar ( i, user_send_buffer, &b_index);
  printf ( "%3d ", i );

printf ( "\n" );
stuff_send_buffer_header ( b_index, 0, CHAR_TYPE, user_send_buffer );

/* send buffer */
special_request ( user_send_buffer, user_receive_buffer );

/* process received_buffer */
init_receive_buffer ( &b_index );
extract_receive_buffer_header ( &b_length, &serial_number, &packet_type,
user_receive_buffer);
printf ( "Receiving : " );
for ( i = 0; i < b_length - 4; i++ )
  printf ( "%3d ", extractchar(user_receive_buffer, &b_index) );

printf("\n");
printf ( "End of demonstration.\n" );

disconnect_robot ( ROBOT_ID );
return ( 1 );
```