DATAMAN MOBILE COMMUNICATIONS LAB
Julio C. Navas and Tomasz Imielinski

Geographic Routing
User’s Guide
# Table of Contents

**Acknowledgements** .......................... 1

**Chapter 1: Overview and Execution** .......................... 3

  Components of the System .................................. 3

  Set-up and Running ...................................... 4

    Environment Variables .................................. 4

    Some Messaging Tips and Hints ......................... 4

    Ideal Full Configuration ................................ 6

    Typical Full Configuration ................................ 7

    Routing-Only Configuration ................................ 8

    GeoFiltering-Only Configuration ....................... 9

**GeoMail Client User Interface** ......................... 10

  Menus .................................................. 10

  User Interface Items .................................. 11

**Chapter 2: Configuration and Parameters** ................. 15

  **Router (geode)** .................................... 15

    Parameters ........................................... 15

    Configuration File .................................. 16

  **GeoNode (geonode)** ................................ 19

    Parameters ........................................... 19

    Configuration File .................................. 20

  **GeoHost (mhd)** .................................... 21

    Parameters ........................................... 21

  **GPS PCMCIA Card Monitor** (position) .................... 21

    Parameters ........................................... 22

  **GeoArp (geoarp)** .................................. 22

    Parameters ........................................... 22

    Configuration File .................................. 22

  **GeoMail Client (geomail)** ......................... 23

    Parameters ........................................... 23

    Configuration File .................................. 24

    Map Definition files ................................ 24

**Chapter 3: GeoAPI** ........................................ 27

  **Geographic Message Class** (GeoMsg) .................. 27

    Definition ........................................... 27

    Creation ............................................. 27

    Operations .......................................... 27

    Packet Manipulation .................................. 28

    Polygon Bounding Box Access ......................... 28

    Header Fields Access Functions ....................... 28

  **Geographic Socket Class** (GeoSocket) ................. 31

    Definition ........................................... 31

    Creation ............................................. 31

    Operations .......................................... 31

    Pan-Message Access Functions ......................... 31

    Socket End-point(s) Functions ......................... 32

    Send/Receive Packets using the GeoMesg class .......... 32

    GeoHost / GeoNode communication commands ............. 34

  **Shape Class** (Shape) ................................ 36

    Definition ........................................... 36

    Creation ............................................. 36

    Operations .......................................... 36

  **IP Packet Class** (packetinet) ......................... 37

    Definition ........................................... 37

    Creation ............................................. 37

    Operations .......................................... 38

    Conversion .......................................... 38

    Data Manipulation .................................... 38

    Moving within the buffer space ....................... 40

    Access functions .................................... 40

**Appendix A: 2D Geometry Classes** ....................... 43

  **Points** (point) ..................................... 43

    Definition ........................................... 43

    Creation ............................................. 43
Acknowledgements

Thanks to all and in-between.

A debt of gratitude is owed to many who, together, have made this project possible.

A hearty thanks also goes to Rob Ruth at DARPA who provided much in the way of encouragement, thoughtful and insightful questions, and, of course, the funding that made this whole project possible.

Thanks also go to Jim Freeberseyser at the ARO for believing in us from the beginning and making the original decision to grant us the GloMo funding.

Michael Melnicki created the original GeoHost daemon implementation and the original library calls that dealt with the interactions between the GeoHost and client applications.

Vasilios Daskalopoulos created the original version of the GeoNode implementation.

The classes, data structures, and methods from the LEDA Class Library from the Max Planck Institute in Saarbrucken, Germany, served as a base for the complex computational geometry techniques necessary for geographical routing.

The Socket++ Socket Class Library (Version: 17Oct95 1.10) was originally created by Gnanasekaran Swaminathan. After many modifications and extensions, it became the base for the geographic socket library.

This research work was supported in part by DARPA under contract numbers DAAH04-95-1-0596 and DAAG55-97-1-0322, NSF grant numbers CCR 95-09620, IRIS 95-09816 and Sponsors of WINLAB.
Overview and Execution

What it is . . . . and how to make it go!

Components of the System

The system is composed of four main components: GeoHosts, GeoAPI, GeoNodes, and GeoRouters.

The GeoAPI is the set of software library routines that allow a programmer to create applications that can send and receive geographic messages.

The GeoHost is located on all computer hosts that are capable of receiving and sending geographic messages. Its role is to notify all client processes about the availability of geographic messages, the host computer’s current geographic location, the address of the local GeoNode, and the address of the local GeoRouter. If a GPS device is present in the mobile, the GeoHost will monitor that device and continually update its notion of the
mobile host’s location. Additionally, with the added accuracy of the GPS device, the GeoHost can further filter the messages that it receives and only accept those that directly correspond to its location. This may be necessary in those cases where the wireless base station covers a large geographical area.

A GeoNode is a buffer for messages with lifetimes. The main function of the GeoNode is to store incoming geographic messages (which have lifetimes greater than zero) for the duration of their lifetimes and to periodically multicast them on all of the subnets or wireless cells to which it is attached. Each subnet and each wireless cell will have at most one GeoNode. The sender of the message specifies the lifetime of a geographic message. Message lifetimes may be necessary because the receivers of geographic messages may be mobile and may possibly arrive at the message destination some time after the geographic message first arrives.

Since, most likely, there will be several geographic messages residing in a GeoNode at one time, the multicasting of the various messages will be scheduled. The scheduling algorithm will take into account the size of the message, the priority of the message, and the speed of the subnet’s transport medium. Clients wishing to receive geographic messages would then tune in to the appropriate multicast group to receive them.

Geographic routers (GeoRouter) are in charge of moving a geographic message from a sender to a receiver. GeoRouters are essentially routers which are geographically aware. Each router is charged with performing geographic routing functions for those networks to which it is directly attached. GeoRouters keep track of the geographic area that they service (called its service area) by calculating the union of the geographic areas covered by the networks attached to it. Its service area is represented as a single simple closed polygon whose vertices are denoted by geographic coordinates. GeoRouters build their routing tables by exchanging service area polygons.

**Set-up and Running**

**Environment Variables**

First and foremost, define and place the following environment variables into your `.profile` or `.login` file:

- **ROUTER_PORT** – port number of the geographic router. This port number can be found in the router’s `geode.conf` configuration file.

- **ROUTER_CONTROL_PORT** – geographic router port number for control messages. This variable is for future research efforts and not currently used – BUT it still must be defined!

- **ROUTER_ADDRESS** – IP address of the host where the geographic router is located.

If you are using the dynamically-linked version which relies on shared libraries, then you must also define and place the following:

- **LD_LIBRARY_PATH** – absolute path to the directory where the shared libraries are stored.

**Some Messaging Tips and Hints**

Keep in mind the following:
• When sending a geographic message from the GeoMail client, the following message fields must be filled in: Life, Port, MCast, Message Text Area. Also, a destination shape must have been drawn on the map display.

• The GeoMail clients bind to port 15000 in order to receive geographic message packets.

• The GeoArp agents bind to port 16000 in order to receive GeoArp query packets.

• Currently, there is no way to have the GeoMail clients join or leave multicast groups.

• When sending a message with no lifetime, set the multicast address to 0.0.0.0 and set the geographic port number to 15000.

• When sending a message with a non-zero lifetime, the multicast address can be set to any valid multicast address. The geographic port number can be set to any valid user-level port number except for 15000 and 16000.

• The router is ready to route geographic packets once it has reported that it has calculated the PEER AREA and (if GeoNodes are being used) the SERVICE AREA.

• The system automatically configures itself quicker if the executables are started in this order:

  1. mbd
  2. geonode
  3. geode
  4. geoarp
  5. geomail
Ideal Full Configuration

In an ideal full geographic routing system configuration, the geographic router (GeoRouter) will execute on a computer that is connected to several networks at the same time. See Figure 2. Each network would have a single GeoNode server. Any host with geographic clients would have a copy of the GeoHost software.

The required command line parameters for the *mhd* and the *geonode* are as follows:

- `mhd -d -g -n`
- `geonode -d -n`

**Note**

The GeoNode, GeoHost, GeoArp, and GeoMail client can all be located on the same computer host.
Since the majority of computers are connected to only one network, the more typical full geographic routing system configuration is shown in Figure 3. In this case, each network will have a GeoRouter execute on a computer that is connected to it. Since the GeoRouters use multicast with a TTL of 2 to find each other, routers on adjacent networks will automatically discover each other. If the routers are spaced further apart, however, then, using the configuration file, either a software tunnel can be established between them or the RIP TTL value can be increased.

The required command line parameters for the *mhd* and the *geonode* are as follows:

- mhd –d –g –n
- geonode –d –n

**Note**

The GeoRouter, GeoNode, GeoHost, GeoArp, and GeoMail client can all be located on the same computer host.
Routing-Only Configuration

If GeoFiltering is not needed and no messages will have lifetimes, then the GeoNode and GeoHost software is not needed. This situation may be preferable because of the extra delay incurred by messages with lifetimes which need to pass through the GeoNode and GeoHost software.

The required command line parameters for the `geomail` program are as follows:

- `geomail -s`

**Note**

The GeoRouter, GeoArp, and GeoMail client can all be located on the same computer host.
GeoFiltering-Only Configuration

Figure 5 - Configuration for Using only GeoFiltering

In a GeoFiltering-only configuration, all of the host computers communicate via the same network link (either wired or wireless). See Figure 5. In this scenario, all of the GeoHosts would move around within the reach of a powerful transmission point – such as a satellite or large radio transmitter. The powerful transmission point is represented by the host containing the GeoNode software.

In this case, all of the geographic clients send their transmitted messages to the GeoNode. The GeoNode, in turn, will echo back the messages to the common network link. If the messages have a non-zero lifetime, then the GeoNode will buffer the messages for the duration of their lifetimes and periodically advertise and multicast them.

If a GPS device is not available, then the GeoHost can be “manually” moved around using the interactive location entry feature of the GeoHost mbd daemon. Using command-line parameters, each GeoHost is given an initial geographic location. Then, in order to manually change the location of the GeoHost, simply press the <ENTER> key and the mbd will respond with the current location and then wait for you to enter the new coordinates. The coordinates should be entered in as: <latitude> <longitude> (note the space in between!).

The required command line parameters for the mbd and the geonode are as follows: (note that the initial location being passed to the mbd is <longitude = -55, latitude = 40 >)

- mbd -d -g -b -o -55 -a 40 -n
- geonode -d -k -n

Note

In order for the GeoMail clients to reflect the new manually-entered geographic locations of the GeoHosts, make sure that Automatic Location Updates is turned on by selecting it from the Location Updates menu.
In order for geographic clients (such as GeoMail), to send all of their messages to the GeoNode, the following changes need to be made to the environment variables:

- **ROUTER_PORT = 5761** – this is the port number of the GeoNode reserved for incoming messages from GeoHosts.

- **ROUTER_ADDRESS** – IP address of the host where the GeoNode is located.

**GeoMail Client User Interface**

![GeoMail User Interface Snapshot](-figure6.png)

The GeoMail client allows users to send and receive geographic messages. The user interface consists of a message area on the left and a map display on the right. See Figure 6. The client will receive any message with a lifetime because it will automatically bind to the appropriate port and join the right multicast group. Messages without lifetimes, however, do not go through the GeoFiltering software and, therefore, must be sent to port 15000 in order for the client to receive it.

The client will display the text of a message that is received and will show the destination polygon for that message in red on the map display. Messages which are advertised by the GeoNode but which do not correspond to the GeoHost’s current location merely have their destination polygon drawn on the map display in green. When the user draws a destination polygon for a message that will be sent in the future, it is drawn on the map display in light blue.

Currently only black-and-white static maps can be loaded and drawn in the map display.

**Menus**

- **File** - operations on static map files
- Open - open a static map file
- Save - not implemented yet
- Exit - exit the program

• Mouse Actions - what occurs when the mouse button is pressed over the map display
  • User - change the user’s coordinates to the selected location
  • Draw Polygon - begin drawing a destination polygon
  • Zoom-in - zoom-in with the map center at the selected location
  • Zoom-out - zoom-in with the map center at the selected location

• Polygon Type - select the type of destination shape
  • Point - destination shape is a point
  • Circle - destination shape is a circle (default)
  • Polygon - destination shape is a polygon

• Coordinates - not implemented yet
  • Show User - not implemented yet
  • Show Map Center - not implemented yet

• Auto Update - should the map display continually update the user’s position?
  • Automatically Update User Position – update the user’s location every second. This is most useful for Geographic message receivers.
  • Do Not Update User Position – do not update the user’s location. This is most useful for Geographic message senders if they are changing maps often.

User Interface Items
• Buttons along the top - These are convenience buttons which execute a command
  • Open - open a static map file
  • Save - not implemented
  • Send Mail - send a geographic message with the specified destination shape,
lifetime, port number, multicast address, and message text.

- GeoARP - send a GeoARP location query to everyone within the map display area
- GeoARP Polygon - send a GeoARP location query to everyone within the specified destination shape.

- Message Text Area - Displays the information contained in a geographic message
- To: - filled in when a message arrives. Displays the geographic location of the sender of the message.
- From: - shows the destination shape of the message
- Subj: - optional subject line for the message
- Life: - lifetime for the message
- Port: - destination port number for the message
- Mcast: - destination multicast address for the message
- Urgent - indicates that the message is urgent and, if it has a lifetime, it should be periodically multicast more often.
- Message Text Area - shows the text of the geographic message

- Map area - Show the host’s location, local geographic message destination polygons, and black-and-white maps.

- Map Display - where the maps, destination shapes, etc. are shown.
- Map Movement - change what is shown on the map display
  - <, >, ^, v - move the map west, east, north, or south one screen
  - in - zoom-in at the current map center
  - out - zoom-out at the current map center
- Center - shows the current location of the mobile host
- Mouse - shows the location of the mouse cursor on the map display
• Message Line along the bottom - Gives the user updated information. Look here for an explanations about the user interface control that the cursor is touching. Also, look here for instructions on how to draw destination polygons.
Configuration and Parameters

Making the software jump through hoops.

The operation of the geographic routing software is easily configured through run-time parameters or configuration files. This section details all of the options available for tweaking the system.

Router (geode)

The geographic router executable is called geode. Ostensibly, geode stands for “GEOgraphic DaEmon.” However, in nature, a geode is a round homely-looking rock that, once opened, reveals a brilliant inner beauty in the form of multi-hued quartz crystals. Therefore, I felt that it was an appropriate name for the geographic router executable.

Parameters

The run-time parameters that the geographic router uses are as follows:

- **d** - turn on debug mode
- **n <name>** - name to give the program
- **c <file>** - name of the error file
- **c <file>** - name of the configuration file
- **h** - print out a help message
- **p <number>** - (not fully implemented) port number to listen on for network management
- **t <number>** - timeout (sec) for tunnel init
- **r <number>** - seconds between routing table refreshes
- **u <number>** - seconds between cache prunes
• a <number>  - max cache item age

**Configuration File**
The `geode` configuration file is meant to be similar to the MSDOS style .ini files. It has the following format:

```ini
# This is a comment
[Section]
display = 1
string = hello
abc
```

In this example, the section “Test” is declared to have the variables “display”, “string”, and “abc”. The “display” variable is set to the value 1, the “string” variable is set to the string “hello”, and the “abc” variable is set to `True`. Lines that begin with the symbol `#` are comment lines.

The `geode` configuration file has the following sections and variables. The variables are shown with suggested or example values.

• [Program]
  ▪ ErrorFile = geode.err  - output file for all of the warning and error messages

• [IterativeServer]
  ▪ Port = 10022  - port number for future network management software

• [Geode]
  ▪ Name = Hobbiton Node  - name of this router
  ▪ Timeout_Max = 10  - number of times to see if a GeoNode is still up
  ▪ Shape = circle -55 40 5  - service area of this router
  ▪ Max_Cache_Age = 240  - maximum age of a cache item before deletion
  ▪ Cache_Prune_Period = 180  - how often to check cache items for old age
  ▪ Table_Refresh_Period = 31  - how often to ping connect GeoNodes
  ▪ Table_Update_Period = 32  - how often to send out GeoNode discovery queries
  ▪ Tunnel_Init_Timeout = 20  - time that a GeoNode has to send a tunnel ack

• [Router]
  ▪ TTL = 2  - TTL of directly-multicast geographic packets
  ▪ Port = 12019  - port number to send/receive geographic packets
• **Send_Space = 130048** - size of router socket send buffer (bytes)

• **Recv_Space = 130048** - size of router socket receiver buffer (bytes)

• **[Rip]**

  • **TTL = 2** - TTL for GeoRIP routing table updates/queries

  • **Port = 11019** - port number for GeoRIP control messages

  • **Multicast = 225.0.0.0** - multicast address for GeoRIP communication

  • **Supplier = true** - if true, then send out routing table info.

  • **Act_As_Gateway = true** - if true, then forward packets

  • **Look_For_Interfaces = true** - discover the local network interfaces

  • **Send_Space = 130048** - size of GeoRIP socket send buffer (bytes)

  • **Recv_Space = 130048** - size of GeoRIP socket receive buffer (bytes)

  • **HopCnt_Infinity = 16** - TTL for infinity

  • **MaxPacketSize = 512** - max size of routing table update packets

  • **Timer_Rate = 30** - how often to integrate routing table changes (sec)

  • **Supply_Interval = 30** - how often to multicast routing table updates (sec).

  • **Check_Interval = 60** - how often to check network interfaces for changes (sec)

  • **Min_WaitTime = 2** - If changes occur between updates, dynamic updates

  • **Max_WaitTime = 5** - containing only changes may be sent. When these are

    - sent, a timer is set for a random value between

    - MIN_WAITTIME and MAX_WAITTIME, and no

    - additional dynamic updates are sent until the timer

    - expires.

  • **Expire_Time = 180** - Every update of a routing entry forces an entry’s timer to

  • **Garbage_Time = 240** - be reset. After EXPIRE_TIME without updates, the

    - entry is marked invalid, but held onto until
- GARBAGE_TIME so that others may see it
- "be deleted”.

- Interface update and creation.

The format is:

{net|host} X.X.X.X gateway X.X.X.X metric DD [remote|internal] [default | point x y | circle x y r | polygon num_pnts x y x y ...]

where:

- Net X.X.X.X - foreign network address (for software tunnels)
- Host X.X.X.X - foreign host address
- Gateway X.X.X.X - address of router gateway for foreign net/host
- Metric DD - metric number to put in routing table
- Internal - Update the info for a network interface
- Remote - remote net/host – OK to route through this node
- Default - use current router’s geographic service area
- Point | Circle | Polygon - specify the geographic service area of foreign net/host

Example of a software tunnel to network 128.6.25.0 through gateway 128.6.25.4 (where there is a geographic router) with a metric of 2 and a service area of a circle with center (-55,60) and radius 5.

- Gateway = net 128.6.25.0 gateway 128.6.25.4 metric 2 remote circle -55 60 5

Example of an update to local network interface 128.6.5.54 with a new metric of 1 and a new service area of a circle with center (-55,60) and radius 5.

- Gateway = net 128.6.5.0 gateway 128.6.5.54 metric 1 internal circle -55 60 5

- [Controller]
  - Tunnels to GeoNodes:
• Tunnel = Computer Science Node - name of tunnel
• Local.Addr = 128.6.157.143 - local end-point
• Remote.Addr = 128.6.5.53 - remote end-point
• Port = 5762 - remote end-point port for data packets
• IGeoMP.Port = 5760 - remote end-point port for control packets
• Rank = child - signifies a GeoNode
• Service.Area = circle -60.0 60.0 1 - service area of GeoNode

-GeoNode discovery queries:
• Multicast = child multicast group - name of discovery multicast group
• Multicast.Addr = 224.1.1.11 - multicast group address to send queries on
• Rank = child - signifies queries for GeoNodes
• Port = 5760 - port GeoNodes are listening on
• TTL = 2 - TTL for multicast queries

**GeoNode (geonode)**

The GeoNode executable is called *geonode*.

**Parameters**

The run-time parameters that the *geonode* uses are as follows:

- `-d` : Debug - print all info possible
- `-k` : Keep Local - all incoming geographic messages are buffered
- `-m` : Monitor - print controlling function names
- `-n` : Do NOT Use ICMP - always use this!
- `-t` number : time to live for advertisements
- `-i` address : interface to use for sending message advertisements
- `-I` address : interface to use for listening for control messages from router
Configuration File
The configuration file, called *geonode.config*, contains parameters according to the following format:

Alarm interval
Advertise interval
multicast address for control messages from the router
First of multicast address block
Last of multicast address block
Number of addresses to follow, i
Address 0
Address 1
...
Address i-1
Service area latitude
Service area longitude
Service area radius

The alarm and advertise intervals are in seconds and specify the period of the alarm and the period at which to send out geographic message advertisements with the message information in them respectively.

The multicast address will be the group the GeoNode subscribes to and sends/receives control messages on. This multicast group should also correspond to what the GeoNode’s peers are subscribed to.

Following this are two more multicast addresses delimiting the beginning and ending of the block of multicast groups that will be used to send the actual buffered geographic messages on. Each message is allocated a multicast address from this block when it is buffered and it is then sent out on this address at the period assigned for as long as the message's lifetime.

Next in the configuration file is a number specifying the number of addresses to follow. This number of addresses are read in and specify the different IP addresses for the base station which will be included in the advertisement messages to the GeoHosts. Since the GeoNode may have more than one connection to different subnets, it must advertise all these addresses to the hosts in its service area. The preferences field for each of these addresses in the advertisement message are currently set to zero.

Finally, after these addresses come three floating point numbers indicating the GeoNode's latitude, longitude and radius of service respectively. The circle described by these numbers specifies the base station's service area. This information is used in the advertisement messages and in response to service area queries from the router.

An example of a *geonode.config* follows:

rivendell
First ever "geonode" Rutgers University DCS, Hill Center, August 1996
1
2
224.1.1.11
230.0.0.0
231.0.0.0
1
128.6.157.143
40.0
-55.0
5.0
**GeoHost (mhd)**

The name of the GeoHose executable is *mhd*.

**Parameters**

- `-d` : Debug - print all received packet info
- `-g` : No GPS card installed - Accept messages intended for any geopolygon
- `-b` : Do not update the GeoHost's current geographic position using the GeoNode's geographic position (for GeoFiltering demos)
- `-o <longitude>` : Set the GeoHost's initial longitude
- `-a <latitude>` : Set the GeoHost's initial latitude
- `-n` : Do not use ICMP - just regular multicast – always use this!
- `-l <host name>` : Use only a this host's geonode (need to use -p too)
- `-p <port num>` : geonode's port number (used with -l)
- `-h` : Help message
- `-i address` : interface to use for listening for advertisement from GeoNodes

**GPS PCMCIA Card Monitor (position)**

The name of the GPS PCMCIA Card Monitor executable is *position*.

In order to be aware of it's current position at all times, the GeoHost creates a TCP socket to the GPS card position daemon (*position*). The *position* daemon can accept connections from several applications at once, and once a second, reports the mobile host's current position to all connections. The *position* daemon uses serial I/O through one of the systems COM ports (usually /dev/cua0) to communicate with the Trimble PCMCIA GPS card. The *position* daemon finds it's current position by querying the GPS card once a second. The PCMCIA card responds with several reports, of which the daemon chooses the report corresponding to the GPS card's current latitude and longitude. Also, the GPS card periodically reports 'Health Messages' which inform the daemon of the GPS card's current status. Occasionally these messages will report errors due to poor operating conditions (ie. poor line-of-sight visibility to the overhead satellites, ionospheric interference, etc...). Most of the source code within the position daemon to interpret the output of the card was taken from Trimble Navigation's DOS based GPS card utilities 'GPSTOOLS'.

The GPS card begins tracking satellites as soon as the serial interface is opened by the position daemon. This tracking takes anywhere from 5 to 15 minutes. The serial interface must remain open as long as the card is to be
used, hence the position daemon only closes the port on exiting. As soon as the serial port is closed, the GPS card loses all information about the satellites, and therefore loses its current position.

**Parameters**

- `-f /dev/cua0` : use the given COM port to interface with the Trimble Navigation GPS Receiver.

**GeoArp (geoarp)**

The name of the GeoArp executable is called *geoarp*.

**Parameters**

The run-time parameters that the geographic router uses are as follows:

- `d` : turn on debug mode
- `n <name>` : name to give the program
- `c <file>` : name of the error file
- `c <file>` : name of the configuration file
- `h` : print out a help message
- `p <number>` : (not fully implemented) port number to listen on for network management

**Configuration File**

The *geoarp* configuration file is meant to be similar to the MSDOS style .ini files. It has the following format:

```
# This is a comment
[Section]
display = 1
string = hello
abc
```

In this example, the section “Test” is declared to have the variables “display”, “string”, and “abc”. The “display” variable is set to the value 1, the “string” variable is set to the string “hello”, and the “abc” variable is set to *True*. Lines that begin with the symbol # are comment lines.

The *geoarp* configuration file has the following sections and variables. The variables are shown with suggested or example values.

- `[Program]`
- ErrorFile = geoarp.err - output file for all of the warning and error messages

- [IterativeServer]
  - Port = 10022 - port number for future network management software

- [GeoArpAgent]
  - GeoPort = 16000 - port number to listen on for geoarp queries

- For each item that should be sent when a geoarp query is received:
  - Title = DataMan Lab - title of item (will appear on geomail client map display)
  - Shape = circle - type of polygon to draw (only circles for now)
  - Longitude = -55 - longitude of circle
  - Latitude = 40 - latitude of circle
  - Radius = 5 - radius of circle
  - Url = http://www.cs.rutgers.edu/~dataman/ - item web address
  - Description = Mobile Communications Lab - item description

**GeoMail Client (geomail)**

The name of the GeoMail Client is *geomail*.

**Parameters**

The run-time parameters that the geographic router uses are as follows:

- d - turn on debug mode
- s - tells the client that there is NO GeoHost
- n <name> - name to give the program
- c <file> - name of the error file
- f <port> - port number the GeoArp agent is listening to for geoarp queries.
- h - print out a help message
**Configuration File**

The `geomail` client does not itself use a configuration file. However, each static map has a configuration file so the client knows how to adjust its map display. Each map is really an X-Windows bitmap that has a suffix of “.xbm”. Each map configuration file should have the same name as the X-Windows bitmap file but it should have an extension of “.map”.

**Map Definition files**

The `geomail` map configuration file is meant to be similar to the MSDOS style .ini files. It has the following format:

```
# This is a comment
[Section]
display = 1
string = hello
abc
```

In this example, the section “Test” is declared to have the variables “display”, “string”, and “abc”. The “display” variable is set to the value 1, the “string” variable is set to the string “hello”, and the “abc” variable is set to `True`. Lines that begin with the symbol `#` are comment lines.

The `geomail` map configuration file has the following sections and variables. The variables are shown with suggested or example values. Note: each static map has to be square!

- **[Geographic]**
  - `Center_Latitude = 37.86725` - latitude of the map’s center
  - `Center_Longitude = -122.29729` - longitude of the map’s center
  - `Degree_Width = 0.250` - width of the map in degrees
  - `Degree_Height = 0.250` - height of the map in degrees

- **[Image]**
  - `Image_File_Name = /home/navas/geo/demo/v2/client/berkeley.xbm` - absolute address of the X-Windows bitmap file containing the map picture itself.
  - `Pixel_Width = 400` - width of the map picture in pixels
  - `Pixel_Height = 400` - height of the map picture in pixels

An example map configuration file follows:

```
#
# berkeley.map
#
# configuration file for the UC Berkeley Map
```
# Julio C. Navas
# Jan. 27 1997
#
[Geographic]
Center_Latitude = 37.86725
Center_Longitude = -122.29729
Degree_Width = 0.250
Degree_Height = 0.250

[Image]
Image_File_Name = /home/navas/geo/demo/v2/client/berkeley.xbm
Pixel_Width = 400
Pixel_Height = 400
GeoAPI

*Geographic Message Application Programming Interface*

The GeoAPI allows programmers to access the functionality of the geographic messaging system. The API is a C++ library which is divided into two main classes: GeoMesg and GeoSocket. The GeoMesg class allows a programmer to create and manipulate geographic messages. The GeoSocket class allows programmers to create and manipulate IP sockets which can send and receive geographic messages.

**Geographic Message Class (GeoMesg)**

**Definition**
The GeoMesg Class is derived from the *packetinet* class. Internally it contains two buffer areas. The first buffer, called the class buffer, contains the header and data information as distinct class objects and in host byte order. The second buffer, called the packet buffer, contains the header and data information as an encoded packet in network-byte order. The header fields access functions and the polygon bounding-box functions all operate on the class buffer. The packet manipulation functions handle the transfer of information between the two buffers.

**Creation**

```c
GeoMesg()
```
This routine will create a geographic message class object and instantiate all internal fields to their default values. In particular, it will create an internal packet buffer of size 4096 bytes. Geographic Messages that are created or received must be able to fit within this buffer size.

```c
GeoMesg( const GeoMesg& gm )
```
A new geographic message class object will be created and instantiated to the values of the internal fields of GeoMesg object *gm*. Completely separate copies of any internal buffers will be made.

```c
GeoMesg( int sz, unsigned char *pkt = NULL )
```
This routine will create a geographic message class object and, if the parameter *pkt* is set to *NULL*, then all internal fields will be instantiated to their default values and it will create an internal packet buffer of size *sz* bytes. Geographic Messages that are created or received must be able to fit within this buffer size. However, if the parameter *pkt* is non-*NULL*, then it is assumed to point to a buffer
of size \( s \). This buffer will now become the internal packet buffer and an attempt is made to parse the contents of the message (if any).

**Operations**

**Packet Manipulation**

bool MessageParse ()

After a packet is received through a geographic socket, this routine is used to decode the header information from the packet buffer into the class buffer. However, the destination polygon information is not extracted. The header fields are now accessible using the Header Field Access routines. Returns *True* if successful and *False* if not.

bool ShapeParse()

After a packet is received through a geographic socket, this routine is used to decode the destination polygon information from the packet header. The destination polygon is now accessible using the GetShape() routine. Returns *True* if successful and *False* if not.

bool MessageCreate()

Uses the information entered using the Header Field Access routines to encode a new geographic packet. Returns *True* if successful and *False* if not.

char * ExtractRawPolygon( int & size )

Returns a *char* pointer to the internal packet buffer space. The pointer points to the beginning of the encoded destination polygon information. The size in bytes of the encoded polygon is placed in the *size* parameter.

**Polygon Bounding Box Access**

Point& GetBottomLeft()

This routine will access the polygon information in the class buffer and return the bottom-left point of the best-fit bounding rectangle around the polygon.

Point& SetBottomLeft( Point & op )

This routine will access the polygon information in the class buffer and will set and return the bottom-left point of the best-fit bounding rectangle around the polygon.

Point& GetTopRight()

This routine will access the polygon information in the class buffer and return the top-right point of the best-fit bounding rectangle around the polygon.

Point& SetTopRight( Point & op )

This routine will access the polygon information in the class buffer and will set and return the top-right point of the best-fit bounding rectangle around the polygon.

**Header Fields Access Functions**

char * GetData( int *size = NULL )

This routine will access the data information in the class buffer and return a *char* pointer to the raw data. If the *size* parameter is non-*NULL*, then the size in bytes of the data is returned.
char * SetData( char *d, int size )
This routine will allocate internal class buffer space to hold and store the data pointed to by parameter d of size in bytes size. A pointer to the internal data buffer space is returned.

u_char GetVersion()
This routine will access the geographic packet header version field in the class buffer and return the version number.

u_char SetVersion( char l )
This routine will set the geographic packet header version field in the class buffer to the value of the parameter l and will return the new version number. The default version number is three.

u_char GetPriority()
This routine will access the priority field in the class buffer and return the priority number. Priorities are currently not taken into account when making routing decisions.

u_char SetPriority( char l )
This routine will set the priority field in the class buffer to the value of the parameter l and will return the new priority number. The default priority number is zero. Priorities start from zero and increase to 255. Larger priorities have precedence over lower priorities. Priorities are currently not taken into account when making routing decisions.

u_char GetFlags()
This routine will access the flag field in the class buffer and return its value.

u_char SetFlags( char l )
This routine will set the flags field in the class buffer to the value of the parameter l and will return the new flags value. The default flags value is zero. Currently defined flag values are:

- ROUTER_FIRST_MESSAGE – obsolete – States that this is the first message seen for a particular polygon.
- ROUTER_PRUNE – for internal router use only – Used to tell routers to prune a path from a routing tree.
- URGENT – Tells GeoNodes on the end-points that this message is “urgent” and should be frequently transmitted.
- FOR_GEONODE – Tells the GeoNodes on the end-points that his message is meant for the GeoNode itself.

ShapeType GetType()
This routine will access the polygon type field in the class buffer and return its value.

ShapeType SetType( ShapeType l )
This routine will set the polygon type field in the class buffer to the value of the parameter l and will return the new type value. The default type value is NoType. The polygon type information is automatically set when the SetShape() or MessageParse() routines are used. Currently defined type values are:
- NoType
- Point
- Circle
- Polygon

u_short GetPort()
This routine will access the class buffer and return the geographic packet port number that the geographic message is destined for.

u_short SetPort( short l )
This routine will access the class buffer and will set the geographic packet port number to the value of the parameter l. The new port number is returned. Any standard IP port number can be used.

u_short GetLifetime();
This routine will access the class buffer and return the lifetime in seconds of the geographic message.

u_short SetLifetime( short l );
This routine will access the class buffer and will set the lifetime in seconds of the geographic message to the value of the parameter l. The new lifetime is returned. The default lifetime is zero.

struct in_addr& GetDestAddr()
This routine will access the class buffer and return an IP internet address structure containing the destination IP multicast group address of the geographic message.

struct in_addr& SetDestAddr( struct in_addr ia )
This routine will access the class buffer and will set the IP multicast group address of the geographic message to the value of the parameter ia. The new group address is returned. The default address is 0.0.0.0 which tells the end-point geographic routers to deliver the geographic message to everyone by broadcasting it. In order to target a subset of the receivers in the destination geographic area, set this field to the IP multicast group address to which everyone in the desired subset of receivers is a member.

struct in_addr& GetSenderAddr()
This routine will access the class buffer and return an IP internet address structure containing the IP address of the sender of the geographic message.

struct in_addr& SetSenderAddr( struct in_addr ia )
This routine will access the class buffer and will set the IP sender address of the geographic message to the value of the parameter ia. This field is automatically set by the GeoSocket class.

Shape* GetShape()
If the internal Shape field is set to some value (say, after using SetShape() or ShapeParse()), then this value will be returned. Otherwise, it will call ShapeParse() and return its result.

Shape* SetShape( Shape& sh )
The internal Shape field will be set to the value of sh and this new value will be returned.
void Clear()
This will clear all internal fields and set them to their default values.

**Geographic Socket Class (GeoSocket)**

**Definition**
The GeoSocket class is derived from the  *io_sockinet* class. The GeoSocket class is designed to allow users to send and received geographic messages. Geographic messages essentially act as IP datagrams with a geographic region as the destination instead of an IP address. Note, however, that GeoSocket does not yet employ io_sockinet's streaming capabilities.

When a GeoSocket object is created, it first determines the IP address of the host. This host address will be inserted into the Sender Address field of all out-going geographic messages. Then, the object determines the IP address and the port number of the geographic router. It first checks to see if the following environment variables are set (all of them must have values):

- ROUTER_PORT – port number of the geographic router.
- ROUTER_CONTROL_PORT – geographic router port number for control messages. This variable is for future research efforts and is currently ignored.
- ROUTER_ADDRESS – IP address of the geographic router.

If these environment variables are not set, then the GeoHost daemon, called  *mhd*, is consulted about the router’s location. If this query fails, then the GeoSocket object cannot send messages but it still will be able to receive.

**Creation**

GeoSocket ()
A new GeoSocket object will be created that is not bound to any port number and not targeted to any geographic destination.

GeoSocket (const sockbuf& sb);
A new GeoSocket object is created. This new object will use the same underlying IP socket as *sb*. The object will not be targeted at any geographic destination.

**Operations**

**Pan-Message Access Functions**

u_short GetPort()
If this GeoSocket object has been targeted to a specific geographic destination (using the Connect() routine), then this routine will return the destination port number. Otherwise, it will return zero.

struct in_addr& GetDestAddr()
If this GeoSocket object has been targeted to a specific geographic destination (using the Connect() routine), then this routine will return the destination IP multicast group address. Otherwise, it will return the address 0.0.0.0.
struct in_addr& GetSenderAddr()
    This routine will return the current host computer’s IP address.

Shape* GetShape()
If this GeoSocket object has been targeted to a specific geographic destination (using the Connect() routine), then this routine will return the destination polygon. Otherwise, it will return NULL.

Socket End-point(s) Functions

bool Connect( Shape & sh, short portnum )
The user can specify a target geographic region for all messages sent by this GeoSocket object. This is similar to using the connect() call on a datagram socket. The destination polygon of all out-going messages will be set to sh, the destination port number will be set to portnum, and the destination multicast address will be set to 0.0.0.0 (deliver to everyone in the destination area). Return true if successful and return false if not.

bool Connect( Shape & sh, short portnum, in_addr& maddr )
The user can specify a target geographic region for all messages sent by this GeoSocket object. This is similar to using the connect() call on a datagram socket. The destination polygon of all out-going messages will be set to sh, the destination port number will be set to portnum, and the destination multicast address will be set to maddr. Return true if successful and return false if not.

bool Bind()
Bind the geographic socket to an available port number. Return true if successful and return false if not.

bool Bind( short portnum )
Bind the geographic socket to the port number portnum. Return true if successful and return false if not.

bool MulticastJoin( in_addr maddr )
Join the multicast group maddr. Return true if successful and return false if not.

bool MulticastDrop( in_addr maddr )
Leave the multicast group maddr. Return true if successful and return false if not.

Send/Receive Packets using the GeoMesg class

int Read ( GeoMesg& pi )
Read from the geographic socket and place any received information into pi. The packet data will be extracted and made ready for access using the GeoMesg GetData() call. If successful, the number of bytes received is returned. If there is no more information being sent (equivalent to an end-of-file), then EOF is returned. If an error occurs, then -1 is returned.

int Recv ( GeoMesg& pi )
Read from the geographic socket and place any received information into pi. The packet data will be extracted and made ready for access using the GeoMesg GetData() call. If successful, the number of bytes received is returned. If there is no more information being sent (equivalent to an end-of-file), then EOF is returned. If an error occurs, then -1 is returned.

int RecvFrom( GeoMesg& pi )
Read from the geographic socket and place any received information into \( pi \). The \( \text{GeoMsg\ MessageParset() \ routine \ will \ be \ automatically \ called \ on \ pi} \). The destination polygon, port, and multicast address will be stored in \( pi \). If successful, the number of bytes received is returned. If there is no more information being sent (equivalent to an end-of-file), then \( EOF \) is returned. If an error occurs, then -1 is returned.

```c
int Read ( char *buf, int size )

int Recv ( char *buf, int size )

int RecvFrom( Shape & sh,
               u_short& portnum,
               in_addr& maddr,
               char *buf,
               int size )
```

Read from the geographic socket and place any received information into the buffer \( buf \) of size \( size \). If successful, the number of bytes received is returned. If there is no more information being sent (equivalent to an end-of-file), then \( EOF \) is returned. If an error occurs, then -1 is returned.

```c
int Write ( GeoMesg& pi )

int Send ( GeoMesg& pi )

int SendTo( GeoMesg& pi )
```

Using the values stored in this \( \text{GeoSocket} \), this routine sets the sender address, destination polygon, destination port, and destination multicast address fields. It then calls the \( \text{GeoMsg::MessageCreate() \ routine \ on \ pi} \), and sends the message to the local router.

Note: This routine presumes that the \( \text{Connect()} \) routine has been called and that the location of the local geographic router is known.

```c
int SendTo( GeoMesg& pi )
```

Using the values stored in the \( \text{GeoMesg} \) parameter \( pi \), this routine sets the sender address, destination polygon, destination port, and destination multicast address fields. It then calls the \( \text{GeoMsg::MessageCreate() \ routine \ on \ pi} \), and sends the message to the local router.

Note: This routine presumes that the \( \text{Connect()} \) routine has been called and that the location of the local geographic router is known.

```c
int Read ( char *buf, int size )
```

Read from the geographic socket and place any received information into the buffer \( buf \) of size \( size \). If successful, the number of bytes received is returned. If there is no more information being sent (equivalent to an end-of-file), then \( EOF \) is returned. If an error occurs, then -1 is returned.

```c
int Recv ( char *buf, int size )
```

Read from the geographic socket and place any received information into the buffer \( buf \) of size \( size \). If successful, the number of bytes received is returned. If there is no more information being sent (equivalent to an end-of-file), then \( EOF \) is returned. If an error occurs, then -1 is returned.
int Write ( char *buf, int size )
Using the values stored in this GeoSocket, this routine sets the sender address, destination polygon, destination port, and destination multicast address fields. It then sends the message stored in buf of size size bytes to the local router.

Note: This routine presumes that the Connect() routine has been called and that the location of the local geographic router is known.

int Send ( char *buf, int size )
Using the values stored in this GeoSocket, this routine sets the sender address, destination polygon, destination port, and destination multicast address fields. It then sends the message stored in buf of size size bytes to the local router.

Note: This routine presumes that the Connect() routine has been called and that the location of the local geographic router is known.

int SendTo( Shape & sh,
            u_short & portnum,
            in_addr& maddr,
            char *buf,
            int size )
Using the values stored in the parameters sb, portnum, and maddr, this routine sets the sender address, destination polygon, destination port, and destination multicast address fields, respectively. It then sends the message to the local router.

Note: This routine presumes that the location of the local geographic router is known.

GeoHost / GeoNode communication commands
list< msg_info > GetAllMsgsInfo()
Contact the local GeoHost and have it respond by sending a list of the meta-data (destination polygon and multicast address) of all of the geographic messages that the local GeoNode is buffering. The structure msg_info is defined as follows:

struct msg_info
{
    in_addr multi_addr;
    u_short port;
    u_char type;
    Shape sh;
};

where multi_addr is the multicast address where the message is periodically multicast by the geonode, port is the destination port number, type is the type of polygon, and sb is the destination polygon.

msg_info GetSpecMsgInfo( in_addr addr )
Contact the local GeoHost and have it send the meta-data of the message identified by addr.

point GetPos()
Contact the local GeoHost and retrieve the current position in longitude and latitude. If the host computer is connected to a GPS device, then the GeoHost will query that device and extract from it
the current geographic location. Otherwise, the GeoHost will extract the current location from the GeoNode advertisements, which contain the geographic location and service area for the GeoNode.

```cpp
bool RecvSpecGeoMsg( GeoMesg& gm,
    in_addr& multi_addr,
    u_short portnum )
```

Download the specific geographic message defined by the multicast group `multi_addr` and the port number `portnum`. This routine will join the group and bind to the port and then wait for the geonode to multicast the message. The message is returned in `gm`. The value `true` is returned if successful and `false` is returned if not.

```cpp
int RecvGeoMsg( GeoMesg& gm )
```

For this library call, several steps are internally taken to receive a datagram. First, the GeoHost is queried for the list of all message advertisements that are currently available for the mobile host's position. The list is then searched for message advertisements which contain a port number equal to the desired port number specified in a preceding `Bind()` call. For any advertisement having the desired port, this routine joins the appropriate multicast group in order to receive the datagram. Then it waits for either more updates from the mhd or a datagram to arrive. As soon as a datagram arrives, it is placed into the parameter `gm`. Subsequent `RecvGeoMsg()` calls can be used to receive remaining datagrams. Information is kept in the `GeoSocket` to prevent the `RecvGeoMsg()` calls from receiving the same datagram twice. Upon success, the number of bytes received is returned. On failure, a -1 is returned.

```cpp
iosockinet* RecvFromGeoStart()
```

`RecvFromGeoStart()`, `HandleDaemonMessage()`, and `RecvFromGeoEnd()` work are designed to work together as a non-blocking version of `RecvGeoMsg()`. `RecvFromGeoStart()` will create an `iosockinet` object and use it to contact the GeoHost and query it for the list of all message advertisements that are currently available for the mobile host's position. Since the GeoHost may not respond immediately, at this point only a pointer to the `iosockinet` object is returned. The user could now use a `select()` or `poll()` call to determine whether the any messages are pending for the `iosockinet` object or this `GeoSocket` object.

```cpp
bool HandleDaemonMessage();
```

`RecvFromGeoStart()`, `HandleDaemonMessage()`, and `RecvFromGeoEnd()` work are designed to work together as a non-blocking version of `RecvGeoMsg()`. This routine should be called when a `select()` or `poll()` system call determines that the GeoHost has sent meta-data to the `iosockinet` object. The meta-data is then searched for message advertisements which contain a port number equal to the desired port number specified in a preceding `Bind()` call. For any advertisement having the desired port, this routine joins the appropriate multicast group. If any problems occur, `false` is returned. Otherwise, `true` is returned.

```cpp
int RecvFromGeoEnd( GeoMesg& gm );
```

`RecvFromGeoStart()`, `HandleDaemonMessage()`, and `RecvFromGeoEnd()` work are designed to work together as a non-blocking version of `RecvGeoMsg()`. This routine should be called when a `select()` or `poll()` system call determines that the GeoNode has sent a message to this `GeoSocket` object. As soon as a datagram arrives, it is placed into the parameter `gm`. Subsequent `RecvFromGeoEnd()` calls can be used to receive remaining datagrams. Information is kept in the `GeoSocket` to prevent the `RecvFromGeoEnd()` calls from receiving the same datagram twice. Upon success, the number of bytes received is returned. On failure, a -1 is returned.
Shape Class (Shape)

Definition
The Shape class is a container class that allows for the various types of polygons to be handled in a more abstract manner. Also, for convenience, all of the routines that act upon polygons, such as intersection, can be made readily available as methods of this class. A Shape class contains at most one polygon. The operations =, ==, ! =, and identical are defined on Shape objects.

Creation

Shape()
Create an empty Shape object and make it of type NoType.

Shape( const Shape& s )
Create a Shape object and make it a duplicate of Shape object s.

Shape( point& npt )
Create a Shape object, make it of type Point, and copy into it the information from parameter npt.

Shape( circle& nc )
Create a Shape object, make it of type Circle, and copy into it the information from parameter nc.

Shape( polygon& np )
Create a Shape object, make it of type Polygon, and copy into it the information from parameter np.

Shape( double x, double y )
Create a Shape object, make it of type Point, and internally store the point information (x, y).

Shape( double x, double y, double r ); // circle
Create a Shape object, make it of type Circle, and internally store the circle information as center = (x, y) and radius = r.

Shape( const list<point> & lpt ); // polygon
Create a Shape object, make it of type Polygon, and internally store the polygon information as the list of points lpt.

Operations

IntersectContainType IntersectOrContain( Shape& sh )
Intersect the current Shape with the parameter sh. Return one of the following IntersectContainType:

- NoIntersectContain – no intersection at all
- Intersect – the polygons intersect
- Contain – one polygon completely contains the other

// shape info
list<point> GetPoints()
Return a list of points containing all of the points in the Shape object. One of the following will be returned depending on the Shape type:

- **NoType** – an empty list
- **Point** – a list containing one point
- **Circle** – a list containing ten points from equidistant positions on the circle’s perimeter.
- **Polygon** – a list containing all of the vertices on the polygon’s perimeter.

```c
ShapeType GetShapeType()
```

Returns the type of the internally stored shape. The type ShapeType can have one of the following values:

- **NoType**
- **Point**
- **Circle**
- **Polygon**

```c
point& GetBottomLeft()
```

Return the bottom-left point of the best-fit bounding rectangle.

```c
point& GetTopRight()
```

Return the top-right point of the best-fit bounding rectangle.

```c
point& GetPoint()
```

If the Shape object’s type is a Point, then a reference to the internal point object is returned.

```c
circle& GetCircle()
```

If the Shape object’s type is a Circle, then a reference to the internal circle object is returned.

```c
polygon& GetPolygon()
```

If the Shape object’s type is a Polygon, then a reference to the internal polygon object is returned.

## IP Packet Class (packetinet)

### Definition

The packetinet class is derived from the socketaddr class. packetinet is an abstract data type for IP packets. An IP packet is essentially a socket address (struct sockaddr_in) plus data. The packetinet class allows users to create and manipulate packets and their contents.

### Creation

```c
packetinet ()
```
Create a \texttt{packetinet} with an empty buffer of default size (4096 bytes) and a destination address of IP address = \texttt{INADDR\_ANY}, family = \texttt{AF\_INET}, and port = 0.

\begin{verbatim}
packetinet (const packetinet& pi )
Create a \texttt{packetinet} that is a duplicate of parameter \texttt{pi}.

packetinet ( int sz, unsigned char *pkt = NULL );
Create a \texttt{packetinet} with an empty buffer of size \texttt{sz} and a destination address of IP address = \texttt{INADDR\_ANY}, family = \texttt{AF\_INET}, and port = 0. If \texttt{pkt} is non-\texttt{NULL}, then make it the internal packet buffer.

packetinet (const char* host_name,
           const char* service_name,
           const char* protocol_name="tcp");
Create a \texttt{packetinet} with an empty buffer of default size (4096 bytes) and a destination address of IP address = \texttt{host\_name}, family = \texttt{AF\_INET}, and port = \texttt{service\_name}. The parameter \texttt{protocol} is used to help resolve the port number.

packetinet ( int sz,
           unsigned char *pkt,
           const char* host_name,
           const char* service_name,
           const char* protocol_name="tcp");
This is a combination of the previous two routines.

packetinet(const socketinternet sina);
Create a \texttt{packetinet} with an empty buffer of default size (4096 bytes) and a destination address equal to \texttt{sina}.

\textbf{Operations}
\textbf{Conversion}
\begin{verbatim}
operator void* () const
Return a \texttt{void*} pointer to the internal packet data buffer.

operator char* () const
Return a \texttt{char*} pointer to the internal packet data buffer.

operator const void* () const
Return a \texttt{const void*} pointer to the internal packet data buffer.

operator const char* () const
Return a \texttt{const char*} pointer to the internal packet data buffer.

operator bool () const
Return \texttt{true} if there is still unread data in the packet data buffer. Return \texttt{false} otherwise.
\end{verbatim}

\textbf{Data Manipulation}
\begin{verbatim}
void ExpandBuffer( int newloc )
Doubles the internal packet data buffer space.
\end{verbatim}
void Insert( char *buf, u_int size )
Directly insert size number of bytes from buf into the packet data buffer starting at the current position. Advance the current position pointer size bytes.

void CharFill( unsigned char val )
Insert the type char parameter val in network-byte order into the packet data buffer starting at the current position. Advance the current position pointer sizeof(char) bytes.

void ShortFill( unsigned short val )
Insert the type short parameter val in network-byte order into the packet data buffer starting at the current position. Advance the current position pointer sizeof(short) bytes.

void LongFill( unsigned long val )
Insert the type long parameter val in network-byte order into the packet data buffer starting at the current position. Advance the current position pointer sizeof(long) bytes.

void DoubleFill( double val );
Insert the type double parameter val in network-byte order into the packet data buffer starting at the current position. Advance the current position pointer sizeof(double) bytes.

void ChunkFill( char *stuff, int stuffsize );
Insert the data pointed to by stuff of size stuffsize into the packet data buffer starting at the current position. Advance the current position pointer by stuffsize bytes. The size of the data inserted is prepended to the data for easy retrieval later.

void InetAddrFill( struct in_addr val );
Insert the IP address parameter val in network-byte order into the packet data buffer starting at the current position. Advance the current position pointer sizeof(struct in_addr) bytes. Note: IP addresses should always be stored in network-byte order. This routine is essentially an alias for Insert( &val, sizeof(struct in_addr) ).

void Extract( char *buf, u_int size ) const;
Directly extract size number of bytes into buf from the packet data buffer starting at the current position. Advance the current position pointer size bytes.

unsigned char ReadChar() const;
Extract and return a value of type char in host-byte order from the packet data buffer starting at the current position. Advance the current position pointer sizeof(char) bytes.

unsigned short ReadShort() const;
Extract and return a value of type short in host-byte order from the packet data buffer starting at the current position. Advance the current position pointer sizeof(short) bytes.

unsigned long ReadLong() const;
Extract and return a value of type long in host-byte order from the packet data buffer starting at the current position. Advance the current position pointer sizeof(long) bytes.

double ReadDouble() const;
Extract and return a value of type double in host-byte order from the packet data buffer starting at the current position. Advance the current position pointer sizeof(double) bytes.
char * ReadChunk( int *chunksize = NULL ) const;
   Extract and return a char * to a chunk of data from the packet data buffer starting at the current
   position. Since the size of the chunk of data precedes the stored data chunk, retrieval is
   straightforward. If the parameter chunksize is non-NULL, then return the size of the chunk in it.
   Advance the current position pointer past the chunk of data.

struct in_addr ReadInetAddress() const;
   Extract and return an IP address in network-byte order from the packet data buffer starting at the
   current position. Advance the current position pointer sizeof( struct in_addr ) bytes. Note: IP
   addresses should always be stored in network-byte order. This routine is essentially an alias for the
   Extract() routine.

Moving within the buffer space
int   Seek( int newloc )
   Go to absolute position newloc within the packet data buffer. Positions numbers start at zero.

int   SeekRel( int newloc )
   Go to position (current position + newloc) within the packet data buffer.

int   Begin()
   Go to the beginning of the packet data buffer.

int   End()
   Go to one byte past the end of the information entered into the packet data buffer.

void   Erase()
   Clear the packet data buffer.

Access functions
int   InfoSize() const
   Return the size in bytes of the amount of information in the packet data buffer.

int   SetInfoSize( int s ) const
   Set the size in bytes of the amount of information in the packet data buffer to be the value s.

int   BufferSize() const
   Return the total size in bytes of the packet data buffer.

int   GetCurrentLoc()
   Return the current position in the packet data buffer as the number of bytes from the beginning of
   the buffer.

unsigned char *GetCurrentLocPtr()
   Return a char * pointer to the internal packet data buffer at the current position.

unsigned char *GetBuffer()
   Return a char * pointer to the packet data buffer.

unsigned char *SetBuffer( int sz, unsigned char *pkt = NULL );
Create an empty packet data buffer of size $s_\infty$. If $pkt$ is non-$NULL$, then make it the internal packet buffer. Any previous buffer is deleted.
2D Geometry Classes

Leveraging LEDA.

The Library of Efficient Data types and Algorithms (LEDA) from the Max Planck Institute in Saarbrucken, Germany, was used extensively as the base for much of the computational geometry coding in the router and in the GeoAPI. For completeness, those classes which are used by the GeoAPI are described here. The class descriptions shown here are copied from the LEDA user manual.

LEDA is available via www from http://www.mpi-sb.mpg.de/LEDA. The distribution contains all sources, installation instructions, technical reports, and the user manual. LEDA is not in the public domain, but can be used freely for research and teaching. Information on a commercial license is available from leda@mpi-sb.mpg.de.

There are also other sources of information. On the LEDA web page you can get the online html version of the manual. There are preliminary chapters of the forthcoming LEDA book available and there are also documentation reports covering internal topics of LEDA and the implementation of diverse data types.

Points (point)

Definition
An instance of the data type point is a point in the two-dimensional plane R^2. We use (x,y) to denote a point with first (or x-) coordinate x and second (or y-) coordinate y.

Creation

<table>
<thead>
<tr>
<th>point</th>
<th>p;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Point</td>
<td>p(double x, double y);</td>
</tr>
<tr>
<td>point</td>
<td>p(vector v);</td>
</tr>
</tbody>
</table>

introduces a variable p of type point initialized to the point (0,0).
introduces a variable p of type point initialized to the point (x,y).
introduces a variable p of type point initialized to the point (v[0],v[1]).

Precondition: v.dim() = 2.
Operations

double p.xcoord() returns the first coordinate of p.
double p.ycoord() returns the second coordinate of p.
vector p.to_vector() returns the vector.
double p.sqr_dist(point q) returns the square of the Euclidean distance between p and q.
double p.xdist(point q) returns the horizontal distance between p and q.
double p.ydist(point q) returns the vertical distance between p and q.
double p.distance(point q) returns the Euclidean distance between p and q.
double p.distance() returns the distance between p and (0,0).
double p.angle(point q, point r) returns the angle between p and r.
point p.translate_by_angle (double alpha, double d) returns p translated in direction alpha by distance d. The direction is given by its angle with a right oriented horizontal ray.
point p.translate(double dx, double dy) returns p translated by vector (dx,dy).
point p.translate(vector v) returns p+v, i.e., p translated by vector v.
  Precondition: v.dim() = 2.
point p + vector v returns p translated by vector v.
point p - vector v returns p translated by vector -v.
point p.rotate(point q, double a) returns p rotated about q by angle a.
point p.rotate(double a) returns p.rotate(point(0,0), a).
point p.rotate90(point q) returns p rotated about q by an angle of 90 degrees.
point p.rotate90() returns p.rotate90(point(0,0)).
point p.reflect(point q, point r) returns p reflected across the straight line passing through q and r.
point p.reflect(point q) returns p reflected across point q.
vector p - q returns the difference vector of the coordinates.
ostream& p - q returns the difference vector of the coordinates.
ostream& O << p writes p to output stream O.
istream& p >> I reads the coordinates of p (two double numbers) from input stream I.
point center(point a, point b) returns the center of a and b, i.e.,
Point midpoint(point a, point b) returns the center of a and b.
Int orientation(point a, point b, point c) computes the orientation of points a, b, and c as the sign of the determinant
  i.e., it returns +1 if point c lies left of the directed line through a and b, 0 if a,b, and c are collinear, and -1 otherwise.
double area(point a, point b, point c) computes the signed area of the triangle determined by a,b,c,
bool collinear(point a, point b, point c)
    returns true if points a, b, c are collinear, i.e.,
    orientation(a, b, c) = 0, and false otherwise.

bool right_turn(point a, point b, point c)
    returns true if points a, b, c form a right turn, i.e.,
    orientation(a, b, c) > 0, and false otherwise.

bool left_turn(point a, point b, point c)
    returns true if points a, b, c form a left turn, i.e.,
    orientation(a, b, c) < 0, and false otherwise.

int side_of_circle (point a, point b, point c, point d)
    returns +1 if point d lies left of the directed circle through
    points a, b, and c, 0 if a,b,c, and d are cocircular, and -1
    otherwise.

bool incircle(point a, point b, point c, point d)
    returns true if point d lies in the interior of the circle through
    points a, b, and c, and false otherwise.

bool outcircle(point a, point b, point c, point d)
    returns true if point d lies outside of the circle through points
    a, b, and c, and false otherwise.

bool cocircular(point a, point b, point c, point d)
    returns true if points a, b, c, and d are cocircular.

Circles (circle)

Definition
An instance C of the data type circle is an oriented circle in the plane passing through three points p_1, p_2, p_3. The orientation of C is equal to the orientation of the three defining points, i.e. orientation(p_1, p_2, p_3). If = 1 C is the empty circle with center p_1. If p_1, p_2, p_3 are collinear C is a straight line passing through p_1, p_2 and p_3 in this order and the center of C is undefined.

Creation

circle C(point a, point b, point c);
    introduces a variable C of type circle. C is initialized to the
    oriented circle through points a, b, and c.

circle C(point a, point b);
    counter-clockwise oriented circle with center a passing
    through b.

circle C(point a);
    introduces a variable C of type circle. C is initialized to the
    trivial circle with center a.

circle C;
    introduces a variable C of type circle. C is initialized to the
    trivial circle with center (0,0).

circle C(point c, double r);
    circle with center c and radius r with positive (i.e. counter-
    clockwise) orientation.
circle C(double x, double y, double r);
introduces a variable C of type circle. C is initialized to the
circle with center (x,y) and radius r with positive (i.e. counter-
clockwise) orientation.

**Operations**

**point** C.center() returns the center of C.

*Precondition:* The orientation of C is not 0.

**double** C.radius() returns the radius of C.

*Precondition:* The orientation of C is not 0.

**point** C.point1() returns p_1.

**point** C.point2() returns p_2.

**point** C.point3() returns p_3.

**point** C.point_on_circle(double alpha, double =0)
returns a point p on C with angle of alpha.

**bool** C.is_degenerate() returns true if the defining points are collinear.

**bool** C.is_trivial() returns true if C has radius zero.

**int** C.orientation() returns the orientation of C.

**int** C.side_of(point p) MISSING.

**bool** C.inside(point p) returns true if p lies inside of C, false otherwise.

**bool** C.outside(point p) returns !C.inside(p).

**bool** C.contains(point p) returns true if p lies on C, false otherwise.

**circle** C.translate_by_angle(double a, double d)
returns C translated in direction a by distance d.

**circle** C.translate(double dx, double dy)
returns C translated by vector (dx,dy).

**circle** C.translate(vector v) returns C translated by vector v.

**circle** C + vector v returns C translated by vector v.

**circle** C - vector v returns C translated by vector -v.

**circle** C.rotate(point q, double a)
returns C rotated about point q by angle a.

**circle** C.rotate(double a) returns C rotated about the origin q by angle a.

**circle** C.rotate90(point q) returns C rotated about q by an angle of 90 degrees.

**circle** C.reflect(point p, point q)
returns C reflected across the straight line passing through p and q.

**circle** C.reflect(point p) returns C reflected across point p.

**list<point>** C.intersection(circle D) returns C <intersection> D as a list of points.

**list<point>** C.intersection(line l) returns C <intersection> l as a list of points.

**list<point>** C.intersection(segment s) returns C <intersection> s as a list of points.

**segment** C.left_tangent(point p) returns the line segment starting in p tangent to C and left of p.

**segment** C.right_tangent(point p) returns the line segment starting in p tangent to C and right of segment [p,C.center()].

**double** C.distance(point p) returns the distance between C and p (negative if p inside C).
double C.distance(line l)  
returns the distance between C and l (negative if l intersects C).

double C.distance(circle D)  
returns the distance between C and D (negative if D intersects C).

**Polygons (polygon)**

**Definition**
An instance P of the data type polygon is a simple polygon in the two-dimensional plane defined by the sequence of its vertices. The number of vertices is called the size of P. A polygon with empty vertex sequence is called empty.

**Creation**

```cpp
polygon P(list<point> pl, bool check=true);
introduces a variable P of type polygon. P is initialized to the polygon with vertex sequence pl.
Precondition: The vertices in pl define a simple polygon (checked if check == true).
```

```cpp
polygon P
introduces a variable P of type polygon. P is initialized to the empty polygon.
```

**Operations**

```cpp
list<point> P.vertices()  
returns the sequence of vertices of P in counterclockwise ordering.
```

```cpp
list<segment> P.edges()  
returns the sequence of bounding segments of P in counterclockwise ordering.
```

```cpp
list<point> P.intersection(segment s)
\returns P <intersection> s as a list of points.
```

```cpp
list<point> P.intersection(line l)
\returns P <intersection> l as a list of points.
\returns P <union> Q as a list of polygons. The first polygon in the list gives the outer boundary of the contour of the union. Possibly following polygons define the inner boundaries (holes) of the contour (holes).
```

```cpp
list<polygon> P.unite(polygon Q)
```

```cpp
list<polygon> P.intersection(polygon Q)
\returns P <intersection> Q as a list of polygons.
```

```cpp
bool P.inside(point p)
\returns true if p lies inside of P and false otherwise.
```

```cpp
bool P.outside(point p)
\returns true if p lies outside of P and false otherwise.
```

```cpp
double P.area()
\returns the area of P.
```

```cpp
double P.translate_by_angle(double a, double d)
\returns P translated in direction a by distance d.
```

```cpp
double P.translate(double dx, double dy)
\returns P translated by vector (dx,dy).
```

```cpp
double P.translate(vector v)
\returns P translated by vector v.
```

```cpp
double P + vector v  
returns P translated by vector v.
```

```cpp
double P - vector v  
returns P translated by vector -v.
```
polygon P.rotate(point q, double a) returns P rotated by angle a about point q.
polygon P.rotate(double a) returns P rotated by angle a about the origin.
polygon P.rotate90(point q) returns P rotated about q by angle of 90 degrees.
polygon P.reflect(point p, point q) returns P reflected across the straight line passing through p and q.
polygon P.reflect(point p) returns P reflected across point p.
int P.size() returns the size of P.
bool P.empty() returns true if P is empty, false otherwise.

Iterations Macros
forall_vertices(v,P) "the vertices of P are successively assigned to point v"

forall_segments(s,P) "the edges of P are successively assigned to segment s"

Segments (segment)

Definition
An instance s of the data type segment is a directed straight line segment in the two-dimensional plane, i.e., a straight line segment [p,q] connecting two points p,q in \( \mathbb{R}^2 \). p is called the source or start point and q is called the target or end point of s. The length of s is the Euclidean distance between p and q. If p = q s is called empty. We use line(s) to denote a straight line containing s. The angle between a right oriented horizontal ray and s is called the direction of s.

Creation
segment s(point p, point q); introduces a variable s of type segment. s is initialized to the segment (p,q)
segment s(point p, vector v); introduces a variable s of type segment. s is initialized to the segment (p,p+v).
Precondition: v.dim() = 2.
segment s(double x1, double y1, double x2, double y2); introduces a variable s of type segment. s is initialized to the segment \([x_1,y_1], (x_2,y_2)\].
segment s(point p, double alpha, double length); introduces a variable s of type segment. s is initialized to the segment with start point p, direction alpha, and length length.
segment s; introduces a variable s of type segment. s is initialized to the empty segment.

Operations
point s.source() returns the source point of segment s.
point s.target() returns the target point of segment s.
double s.xcoord1() returns the x-coordinate of s.source().
double s.xcoord2() returns the x-coordinate of s.target().

double s.ycoord1() returns the y-coordinate of s.source().

double s.ycoord2() returns the y-coordinate of s.target().

double s.dx() returns the xcoord2 - xcoord1.

double s.dy() returns the ycoord2 - ycoord1.

double s.slope() returns the slope of s.

Precondition: s is not vertical.

double s.sqr_length() returns the square of the length of s.

double s.length() returns the length of s.

double s.direction() returns the direction of s as an angle in the interval [0, 2pi).

double s.angle() returns s.direction().

double s.angle(segment t) returns the angle between s and t, i.e., t.direction() - s.direction().

bool s.is_trivial() returns true if s is trivial.

bool s.is_vertical() returns true iff s is vertical.

bool s.is_horizontal() returns true iff s is horizontal.

double s.x_proj(double y) returns p.xcoord(), where p in line(s) with p.ycoord() = y.

Precondition: s is not horizontal.

double s.y_proj(double x) returns p.ycoord(), where p in line(s) with p.xcoord() = x.

Precondition: s is not vertical.

double s.y_abs() returns the y-abscissa of line(s), i.e., s.y_proj(0).

Precondition: s is not vertical.

bool s.contains(point p) decides whether s contains p.

bool s.intersection(segment t) decides whether s and t intersect in one point.

bool s.intersection(segment t, point& p)

if s and t intersect in a single point this point is assigned to p and the result is true, otherwise the result is false.

bool s.intersection_of_lines(segment t, point& p)

if line(s) and line(t) intersect in a single point this point is assigned to p and the result is true, otherwise the result is false.

segment s.translate_by_angle(double alpha, double d)

returns s translated in direction alpha by distance d.

segment s.translate(double dx, double dy)

returns s translated by vector (dx, dy).

segment s.translate(vector v) returns s + v, i.e., s translated by vector v.

Precondition: v.dim() = 2.

segment s + vector v returns s translated by vector v.

segment s - vector v returns s translated by vector -v.

segment s.perpendicular(point p) returns the segment perpendicular to s with source p. and target on line(s).

double s.distance(point p) returns the Euclidean distance between p and s.

double s.distance() returns the Euclidean distance between (0,0) and s.

segment s.rotate(point q, double a)

returns s rotated about point q by angle a.

segment s.rotate(double alpha)

returns s.rotate(s.source(), alpha).
segment  \( s.\text{rotate90}(\text{point} \ q) \)  \( s.\text{rotate90}() \)  \( s.\text{reflect}(\text{point} \ p, \text{point} \ q) \)
returns \( s \) rotated about \( q \) by an angle of 90 degrees.
returns \( s.\text{rotate90}(s.\text{source}(), \ a) \).
returns \( s \) reflected across the straight line passing through \( p \) and \( q \).
returns \( s \) reflected across point \( p \).
writes \( s \) to output stream \( O \).
reads the coordinates of \( s \) (four double numbers) from input stream \( I \).

**Non-Member Functions**

\[
\begin{align*}
\text{int} & \quad \text{orientation}(\text{segment} \ s, \ \text{point} \ p) \\
& \quad \text{computes orientation}(s.\text{source}(), s.\text{target}(), p).
\end{align*}
\]

\[
\begin{align*}
\text{int} & \quad \text{cmp_slopes}(\text{segment} \ s_1, \ \text{segment} \ s_2) \\
& \quad \text{return} \ \text{compare(slope(s_1), slope(s_2))}.
\end{align*}
\]

\[
\begin{align*}
\text{int} & \quad \text{cmp_segments_at_xcoord}(\text{segment} \ s_1, \ \text{segment} \ s_2, \ \text{point} \ p) \\
& \quad \text{compares points l_1 <intersection> v and l_2 <intersection> v} \\
& \quad \text{where l_i is the line underlying segment s_i and v is the} \\
& \quad \text{vertical straight line passing through point p}.
\end{align*}
\]

\[
\begin{align*}
\text{bool} & \quad \text{parallel}(\text{segment} \ s_1, \ \text{segment} \ s_2) \\
& \quad \text{Returns true if } s_1 \text{ and } s_2 \text{ are parallel and false otherwise.}
\end{align*}
\]

---

**Real-Valued Vectors (vector)**

**Definition**
An instance of data type vector is a vector of variables of type double.

**Creation**

\[
\begin{align*}
\text{vector} & \quad \text{v;} \\
& \quad \text{creates an instance } v \text{ of type vector; } v \text{ is initialized to the} \\
& \quad \text{zero-dimensional vector.}
\end{align*}
\]

\[
\begin{align*}
\text{vector} & \quad v(\text{int} \ d); \\
& \quad \text{creates an instance } v \text{ of type vector; } v \text{ is initialized to the zero} \\
& \quad \text{vector of dimension } d.
\end{align*}
\]

\[
\begin{align*}
\text{vector} & \quad v(\text{double} \ a, \ \text{double} \ b); \\
& \quad \text{creates an instance } v \text{ of type vector; } v \text{ is initialized to the two-} \\
& \quad \text{dimensional vector } (a,b).
\end{align*}
\]

\[
\begin{align*}
\text{vector} & \quad v(\text{double} \ a, \ \text{double} \ b, \ \text{double} \ c); \\
& \quad \text{creates an instance } v \text{ of type vector; } v \text{ is initialized to the} \\
& \quad \text{three-dimensional vector } (a,b,c).
\end{align*}
\]

**Operations**

\[
\begin{align*}
\text{int} & \quad v.\text{dim}() \\
& \quad \text{returns the dimension of } v.
\end{align*}
\]

\[
\begin{align*}
\text{double} & \quad v[\text{int} \ i] \\
& \quad \text{returns } i\text{-th component of } v. \\
& \quad \text{Precondition: } 0 <= i <= v.\text{dim}()-1.
\end{align*}
\]

\[
\begin{align*}
\text{double} & \quad v.\text{sqr_length}() \\
& \quad \text{returns the square of the Euclidean length of } v.
\end{align*}
\]

\[
\begin{align*}
\text{double} & \quad v.\text{length}() \\
& \quad \text{returns the Euclidean length of } v.
\end{align*}
\]

\[
\begin{align*}
\text{vector} & \quad v.\text{norm}() \\
& \quad \text{returns } v \text{ normalized.}
\end{align*}
\]
double v.angle(vector w)     returns the angle between v and w.
vector v.rotate90()          returns the v rotated by 90 degrees.
Precondition: v.dim() = 2
vector v.rotate(double a)    returns the v rotated by an angle of a.
Precondition: v.dim() = 2
vector v + v1                Addition.
Precondition: v.dim() = v1.dim().
vector v - v1                Subtraction.
Precondition: v.dim() = v1.dim().
double v * v1                Scalar multiplication.
Precondition: v.dim() = v1.dim().
vector v * double r          Componentwise multiplication with double r.
bool v == w                  Test for equality.
bool v != w                  Test for inequality.
void v.print(ostream& O)     prints v componentwise to ostream O.
void v.print()               prints v to cout.
void v.read(istream& I)      reads d = v.dim() numbers from input stream I and writes
                            them into v[0] ... v[d-1].
void v.read()                reads v from cin.
ostream& ostream& O << v    writes v componentwise to the output stream O.
istream& istream& I >>     reads v componentwise from the input stream I.
vector& v

Implementation
Vectors are implemented by arrays of real numbers. All operations on a vector v take time O(v.dim()), except for dim and [ ] which take constant time. The space requirement is O(v.dim()). Be aware that the operations on vectors and matrices incur rounding errors and hence are not completely reliable. For example, if M is a matrix, b is a vector, and x is computed by x = M.solve(b) it is not necessarily true that the test b == M * b evaluates to true. The types integer_vector and integer_matrix provide exact linear algebra.

Linear Lists (list)

Definition
An instance L of the parameterized data type list<E> is a sequence of items (list_item). Each item in L contains an element of data type E, called the element type of L. The number of items in L is called the length of L. If L has length zero it is called the empty list. In the sequel <x> is used to denote a list item containing the element x and L[i] is used to denote the contents of list item i in L.

Creation
list<E> L;             creates an instance L of type list<E> and initializes it to the empty list.

Operations
Access Operations
int L.length()         returns the length of L.
int L.size() returns L.length().
bool L.empty() returns true if L is empty, false otherwise.
list_item L.first() returns the first item of L (nil if L is empty).
list_item L.last() returns the last item of L (nil if L is empty).
list_item L.get_item(int i) returns the item at position i (the first position is 0).
    Precondition: i < L.length().
list_item L.succ(list_item it) returns the successor item of item it, nil if it=L.last().
    Precondition: it is an item in L.
list_item L.pred(list_item it) returns the predecessor item of item it, nil if it=L.first().
    Precondition: it is an item in L.
list_item L.cyclic_succ(list_item it) returns the cyclic successor of item it, i.e., L.first() if it = L.last(), L.succ(it) otherwise.
list_item L.cyclic_pred(list_item it) returns the cyclic predecessor of item it, i.e, L.last() if it = L.first(), L.pred(it) otherwise.
list_item L.search(E x) returns the first item of L that contains x, nil if x is not an element of L.
    Precondition: compare has to be defined for type E.
void L.remove(E x) removes all items with contents x from L.
    Precondition: compare has to be defined for type E.
E L.contents(list_item it) returns the contents L[it] of item it.
    Precondition: it is an item in L.
E L.inf(list_item it) returns L.contents(it).
E L.front() returns the first element of L, i.e. the contents of L.first().
    Precondition: L is not empty.
E L.head() see L.front().
E L.back() returns the last element of L, i.e. the contents of L.last().
    Precondition: L is not empty.
E L.tail() see L.back().
int L.rank(E x) returns the rank of x in L, i.e. its first position in L as an integer from [1.. |L|] (0 if x is not in L).

**Update Operations**

list_item L.push(E x) adds a new item <x> at the front of L and returns it (L.insert(x,L.first(),before)).
list_item L.push_front(E x) see L.push(x).
list_item L.append(E x) appends a new item <x> to L and returns it (L.insert(x,L.last(),after)).
list_item L.push_back(E x) see L.append(x).
list_item L.insert(E x, list_item pos, int dir=after) inserts a new item <x> after (if dir=after) or before (if dir=before) item pos into L and returns it (here after and before are predefined constants).
    Precondition: it is an item in L.
E L.pop() deletes the first item from L and returns its contents.
    Precondition: L is not empty.
E
L.pop_front() see L.pop().
E
L.Pop() deletes the last item from L and returns its contents.
Precondition: L is not empty.
E
L.pop_back() see L.Pop().
void
L.erase(list_item it) deletes the item it from L.
Precondition: it is an item in L.
E
L.del_item(list_item it) deletes the item it from L and returns its contents L[it].
Precondition: it is an item in L.
E
L.del(list_item it) same as L.del_item(it).
void
L.move_to_front(list_item it) moves it to the front end of L.
void
L.move_to_rear(list_item it) moves it to the rear end of L.
void
L.assign(list_item it, E x) makes x the contents of item it.
Precondition: it is an item in L.
void
L.conc(list<E>& L1, int dir = after) appends (dir = after or prepends (dir = before) list L_-1 to list L and makes L_-1 the empty list.
Precondition: L != L_-1
void
L.swap(list<E>& L1) swaps lists of items of L and L1;
void
L.split(list_item it, list<E>& L1, list<E>& L2) splits L at item it into lists L1 and L2. More precisely, if it != nil and L = x_1,...,x_k-1,it,x_k+1,...,x_n then L1 = x_1,...,x_k-1 and L2 = it,x_k+1,...,x_n. If it = nil then L1 is made empty and L2 a copy of L. Finally L is made empty if it is not identical to L1 or L2.
Precondition: it is an item of L or nil.
void
L.split(list_item it, list<E>& L1, list<E>& L2, int dir) splits L at item it into lists L1 and L2. Item it becomes the first item of L2 if dir==0 and the last item of L1 otherwise. Precondition: it is an item of L.
void
L.sort(int (*cmp)(E, E)) sorts the items of L using the ordering defined by the compare function cmp : Ex E -> int, with

More precisely, if (in_1,...,in_n) and (out_1,...,out_n) denote the values of L before and after the call of sort, then cmp(L[out_i], L[out_j+i]) <= 0 for 1 <= j<n and there is a permutation pi of [1..n] such that out_i = in_pi_i for 1 <= i <= n.

void
L.sort() sorts the items of L using the default ordering of type E, i.e.,
the linear order defined by function int compare(const E&,
const E&).
void
L.unique() removes duplicates from L.
Precondition: L is sorted increasely according to the default
void L.unique(int (*cmp)(E, E))
removes duplicates from L.
Precondition: L is sorted increasing according to the ordering defined by cmp.

void L.merge(list<E>& l1)
merges the items of L and L1 using the default ordering of type E. The result is assigned to L and L1 is made empty.
Precondition: L and L1 are sorted increasing according to the default ordering of type E.

void L.merge(list<E>& l1, int (*cmp)(E, E))
merges the items of L and L1 using the ordering defined by cmp.

list_item L.min(int (*cmp)(E, E))
returns the item with the minimal contents with respect to the linear order defined by compare function cmp.

list_item L.min()
returns the item with the minimal contents with respect to the default linear order of type E.

list_item L.max(int (*cmp)(E, E))
returns the item with the maximal contents with respect to the linear order defined by compare function cmp.

list_item L.max()
returns the item with the maximal contents with respect to the default linear order of type E.

void L.apply(void (*f)(E& for all items <x> in L function f is called with argument x))
(passed by reference).

void L.reverse_items()
reverses the sequence of items of L.

void L.reverse_items(list_item it1, list_item it2)
reverses the sub-sequence it1,...,it2 of items of L.
Precondition: it1 = it2 or it1 appears before it2 in L.

void L.reverse()
reverses the sequence of entries of L.

void L.reverse(list_item it1, list_item it2)
reverses the sequence of entries L[it1] ... L[it2].

void L.permute()
randomly permutes the items of L.

void L.bucket_sort (int i, int j, int (*f)(E))
sorts the items of L using bucket sort, where f : E -> int with f(x) in [i,j] for all elements x of L. The sort is stable, i.e., if f(x)=f(y) and <x> is before <y> in L then <x> is before <y> after the sort.

void L.bucket_sort(int (*f)(E))
same as bucket_sort(i,j,f) where i and j are the minimal and maximal value of f(e) as e ranges over all elements of L.

void L.clear()
makes L the empty list.

Input and Output

void L.read(istream& I, char delim = (char)EOF)
reads a sequence of objects of type E terminated by the delimiter delim from the input stream I using operator>>(istream&,E&). L is made a list of appropriate length and the sequence is stored in L.

void L.read(char delim = 'calls L.read(cin, delim) to read L from the standard input stream cin.'
void L.read(string s, char delim = 'n')
    As above, but uses string s as a prompt.
void L.print(ostream& O, char space = ' ')
    prints the contents of list L to the output stream O using
    operator<<(ostream&, const E&) to print each element. The
    elements are separated by character space.
void L.print(char space = ' ')
    calls L.print(cout, space) to print L on the standard output
    stream cout.
void L.print(string s, char space = ' ')
    As above, but uses string s as a header.

Operators

list<E>& L = list L

list_item L += E x
list_item L[int i]
E& L[list_item it]

list_item L = list L1
then L is made a sequence of items y_1, y_2, ... , y_n
with L[y_i] = L1[x_i] for 1 <= i <= n.

Iterations Macros

forall_items(it,L) ``the items of L are successively assigned to it''
forall(x,L) ``the elements of L are successively assigned to x''

Implementation

The data type list is realized by doubly linked linear lists. All operations take constant time except for the
following operations: search and rank take linear time O(n), item(i) takes time O(i), bucket_sort takes
time O(n + j - i) and sort takes time O(n* c*log n) where c is the time complexity of the compare
function. n is always the current length of the list.
C++ Socket Classes

Classes to automate much of the IP bits, bytes, and nibbles.

C++ Class library (socket++) defines a family of C++ classes that can be used more effectively than directly calling the underlying low-level system functions. One distinct advantage of the socket++ is that it has the same interface as that of the istream so that the users can perform type-safe input output. See your local IOSStream library documentation for more information on istreams.

The Socket++ Socket Class Library (Version: 17Oct95 1.10) was originally created by Gnanasekaran Swaminathan. After many modifications and extensions, it became the base for the geographic socket library. The primary modifications include: new internal plumbing, new convenience socket option methods, support for ioctl() and fcntl(), support for multicast, support for the packet class, and support for interaction with network interfaces. Since the majority of the socket class interface stayed the same, what follows is a modified version of the original Socket++ Socket Class Library manual.

The streambuf counterpart of the socket++ is sockbuf. sockbuf is an endpoint for communication with yet another sockbuf or simply a socket descriptor. sockbuf has also methods that act as interfaces for most of the commonly used system calls that involve sockets. See section sockbuf Class, for more information on the socket buffer class.

For each communication domain, we derive a new class from sockbuf that has some additional methods that are specific to that domain. At present, only the inet domain is supported. sockinetbuf class defines inet domain of sockets. See section sockinetbuf Class, for inet sockets.

We also have a domain specific socket address class that is derived from a common base class called sockAddr. sockinetaddr class is used for inet domain addresses. For more information on address classes see section sockAddr Class and section sockinetaddr Class.

Note: sockAddr is not spelled sockaddr in order to prevent name clash with the struct sockaddr declared in `<sys/socket.h>'.

We noted earlier that the C++ socket class provides the same interface as the istream library. For example, in the internet domain, we have isockinet, osockinet, and iosockinet classes that are counterparts to istream, ostream, and istream classes of IOStream library. For more details on iosockstream classes see See section sockstream Classes.
**sockbuf Class**

sockbuf class is derived from streambuf class of the iostream library. You can simultaneously read and write into a sockbuf just like you can listen and talk through a telephone. To accomplish the above goal, we maintain two independent buffers for reading and writing.

**Constructors**

sockbuf constructors sets up an endpoint for communication. A sockbuf object so created can be read from and written to in linebuffered mode. To change mode, refer to streambuf class in your IOStream library.

**Note**: If you are using AT&T IOStream library, then the linebuffered mode is permanently turned off. Thus, you need to explicitly flush a socket stream. You can flush a socket stream buffer in one of the following four ways:

```cpp
// os is a socket ostream
os << "this is a test" << endl;
os << "this is a test\n" << flush;
os << "this is a test\n"; os.flush ();
os << "this is a test\n"; os->sync ();
```

sockbuf objects are created as follows where

- `s` and `so` are sockbuf objects
- `sd` is an integer which is a socket descriptor
- `af` and `proto` are integers which denote domain number and protocol number respectively
- `ty` is a sockbuf::type and must be one of sockbuf::sock_stream, sockbuf::sock_dgram, sockbuf::sock_raw, sockbuf::sock_rdm, and sockbuf::sock_seqpacket

```cpp
sockbuf s(sd);
sockbuf s;
```

Set socket descriptor of `s` to `sd` (defaults to -1). sockbuf destructor will close `sd`.

```cpp
sockbuf s(af, ty, proto);
```

Set socket descriptor of `s` to `::socket(af, int(ty), proto);`

```cpp
sockbuf so(s);
```

Set socket descriptor of `so` to the socket descriptor of `s`.

```cpp
s.open(ty, proto)
```

does nothing and returns simply 0, the null pointer to sockbuf.
s.is_open()

returns a non-zero number if the socket descriptor is open else return 0.

s = so;

return a reference s after assigning s with so.

**Destructor**

sockbuf::~sockbuf() flushes output and closes its socket if no other sockbuf is referencing it and _S_DELETE_DONT_CLOSE flag is not set. It also deletes its read and write buffers.

In what follows,

s is a sockbuf object

how is of type sockbuf::shuto how and must be one of sockbuf::shut_read, sockbuf::shut_write, and sockbuf::shut_readwrite

sockbuf::~sockbuf()

flushes output and closes its socket if no other sockbuf object is referencing it before deleting its read and write buffers. If the _S_DELETE_DONT_CLOSE flag is set, then the socket is not closed.

s.close()

closes the socket even if it is referenced by other sockbuf objects and _S_DELETE_DONT_CLOSE flag is set.

s.shutdown(how)

shuts down read if how is sockbuf::shut_read, shuts down write if how is sockbuf::shut_write, and shuts down both read and write if how is sockbuf::shut_readwrite.

**Reading and Writing**

sockbuf class offers several ways to read and write and tailors the behavior of several virtual functions of streambuf for socket communication.

In case of error, sockbuf::error(const char*) is called.

In what follows,

- s is a sockbuf object

- pi is a packetinet class object
- buf is buffer of type char*
- bufsz is an integer and is less than sizeof(buf)
- msgf is an integer and denotes the message flag
- sa is of type sockAddr
- msgh is a pointer to struct msghdr
- wp is an integer and denotes time in seconds
- c is a char

s.write(pi, bufsz)
returns an int which must be equal to bufsz if bufsz chars in the pi are written successfully. It returns 0 if there is nothing to write or if, in case of timeouts, the socket is not ready for write section Time Outs While Reading and Writing.

s.send(pi, msgf)

same as sockbuf::write described above but allows the user to control the transmission of messages using the message flag msgf. If msgf is sockbuf::msg_oob and the socket type of s is sockbuf::sock_stream, s sends the message in out-of-band mode. If msgf is sockbuf::msg_dontroute, s sends the outgoing packets without routing. If msgf is 0, which is the default case, sockbuf::send behaves exactly like sockbuf::write.

s.sendto(pi, msgf)

same as sockbuf::send but works on unconnected sockets. pi specifies the to address for the message.

s.read(pi, bufsz)
returns an int which is the number of chars read into the pi. In case of EOF, return EOF. Here, bufsz indicates the size of the buf. In case of timeouts, return 0 section Time Outs While Reading and Writing.

s.recv(pi, msgf)

same as sockbuf::read described above but allows the user to receive out-of-band data if msgf is sockbuf::msg_oob or to preview the data waiting to be read if msgf is sockbuf::msg.peek. If msgf is 0, which is the default case, sockbuf::recv behaves exactly like sockbuf::read.

s.recvfrom(pi, msgf)

same as sockbuf::recv but works on unconnected sockets. pi specifies the from address for the message.
s.is_open()

returns a non-zero number if the socket descriptor is open else return 0.

s.is_eof()

returns a non-zero number if the socket has seen EOF while reading else return 0.

s.write(buf, bufsz)

returns an int which must be equal to bufsz if bufsz chars in the buf are written successfully. It returns 0 if there is nothing to write or if, in case of timeouts, the socket is not ready for write section Time Outs While Reading and Writing.

s.send(buf, bufsz, msgf)

same as sockbuf::write described above but allows the user to control the transmission of messages using the message flag msgf. If msgf is sockbuf::msg_oob and the socket type of s is sockbuf::sock_stream, s sends the message in out-of-band mode. If msgf is sockbuf::msg_dontroute, s sends the outgoing packets without routing. If msgf is 0, which is the default case, sockbuf::send behaves exactly like sockbuf::write.

s.sendto(sa, buf, bufsz, msgf)

same as sockbuf::send but works on unconnected sockets. sa specifies the to address for the message.

s.sendmsg(msgh, msgf)

same as sockbuf::send but sends a struct msghdr object instead.

s.sys_write(buf, bufsz)

calls sockbuf::write and returns the result. Unlike sockbuf::write sockbuf::sys_write is declared as a virtual function.

s.read(buf, bufsz)

returns an int which is the number of chars read into the buf. In case of EOF, return EOF. Here, bufsz indicates the size of the buf. In case of timeouts, return 0 section Time Outs While Reading and Writing.

s.recv(buf, bufsz, msgf)

same as sockbuf::read described above but allows the user to receive out-of-band data if msgf is sockbuf::msg_oob or to preview the data waiting to be read if msgf is sockbuf::msg_peek. If msgf is 0, which is the default case, sockbuf::recv behaves exactly like sockbuf::read.

s.recvfrom(sa, buf, bufsz, msgf)

same as sockbuf::recv but works on unconnected sockets. sa specifies the from address for the message.
s.recvmsg(msgh, msgf)

same as sockbuf::recv but reads a struct msghdr object instead.

s.sys_read(buf, bufsz)

calls sockbuf::read and returns the result. Unlike sockbuf::read sockbuf::sys_read is declared as a virtual function.

s.is_readready(wp_sec, wp_usec)

returns a non-zero int if s has data waiting to be read from the communication channel. If wp_sec >= 0, it waits for wp_sec 10^6 + wp_usec microseconds before returning 0 in case there are no data waiting to be read. If wp_sec < 0, then it waits until a datum arrives at the communication channel. wp_usec defaults to 0.

**Please Note:** The data waiting in sockbuf's own buffer is different from the data waiting in the communication channel.

s.is_writeready(wp_sec, wp_usec)

returns a non-zero int if data can be written onto the communication channel of s. If wp_sec >= 0, it waits for wp_sec 10^6 + wp_usec microseconds before returning 0 in case no data can be written. If wp_sec < 0, then it waits until the communication channel is ready to accept data. wp_usec defaults to 0.

**Please Note:** The buffer of the sockbuf class is different from the buffer of the communication channel buffer.

s.is_exceptionpending(wp_sec, wp_usec)

returns non-zero int if s has any exception events pending. If wp_sec >= 0, it waits for wp_sec 10^6 + wp_usec microseconds before returning 0 in case s does not have any exception events pending. If wp_sec < 0, then it waits until an exception event occurs. wp_usec defaults to 0.

**Please Note:** The exceptions that sockbuf::is_exceptionpending is looking for are different from the C++ exceptions.

s.flush_output()

flushes the output buffer and returns the number of chars flushed. In case of error, return EOF. sockbuf::flush_output is a protected member function and it is not available for general public.

s.doallocate()

allocates free store for read and write buffers of s and returns 1 if allocation is done and returns 0 if there is no need. sockbuf::doallocate is a protected virtual member function and it is not available for general public.

s.underflow()
returns the unread char in the buffer as an unsigned char if there is any. Else returns EOF if s cannot allocate space for the buffers, cannot read or peer is closed. sockbuf::underflow is a protected virtual member function and it is not available for general public.

    s.overflow(c)

if c==EOF, call and return the result of flush_output(), else if c=='\n' and s is linebuffered, call flush_output() and return c unless flush_output() returns EOF, in which case return EOF. In any other case, insert char c into the buffer and return c as an unsigned char.

sockbuf::overflow is a protected member virtual function and it is not available for general public.

    Note: linebuffered mode does not work with AT&T IOStream library. Use explicit flushing to flush sockbuf.

    s.sync()

calls flush_output() and returns the result. Useful if the user needs to flush the output without writing newline char into the write buffer.

    s.xsputn(buf, bufsz)

write bufsz chars into the buffer and returns the number of chars successfully written. Output is flushed if any char in buf[0..bufsz-1] is '\n'.

    s.recvtimeout(wp)

sets the recv timeout to wp seconds. If wp is -1, it is a block and if wp is 0, it is a poll. It affects all read functions. If the socket is not read ready within wp seconds, the read call will return 0. It also affects sockbuf::underflow. sockbuf::underflow will not set the _S_EOF_SEEN flag if it is returning EOF because of timeout. sockbuf::recvtimeout returns the old recv timeout value.

    s.sendtimeout(wp)

sets the send timeout to wp seconds. If wp is -1, it is a block and if wp is 0, it is a poll. It affects all write functions. If the socket is not write ready within wp seconds, the write call will return 0. sockbuf::sendtimeout returns the old send timeout value.

**Establishing connections**

A name must be bound to a sockbuf if processes want to refer to it and use it for communication. Names must be unique. An *inet* name is a 5-tuple, `<protocol, local addr, local port, peer addr, peer port>`. sockbuf::bind is used to specify the local half of the name--- `<local addr, local port>` for *inet*. sockbuf::connect and sockbuf::accept are used to specify the peer half of the name--- `<peer addr, peer port>` for *inet*.

In what follows,

- s and so are sockbuf objects
• sa is a sockaddr object

• nc is an integer denoting the number of connections to allow

```cpp
s.bind(sa)
```

binds sockaddr sa as the local half of the name for s. It returns 0 on success and returns the errno on failure.

```cpp
s.connect(sa)
```

sockbuf::connect uses sa to provide the peer half of the name for s and to establish the connection itself. sockbuf::connect also provides the local half of the name automatically and hence, the user should not use sockbuf::bind to bind any local half of the name. It returns 0 on success and returns the errno on failure.

```cpp
s.listen(nc)
```

makes s ready to accept connections. nc specifies the maximum number of outstanding connections that may be queued and must be at least 1 and less than or equal to sockbuf::somaxconn which is usually 5 on most systems.

```cpp
sockbuf so = s.accept(sa)
```

```cpp
sockbuf so = s.accept()
```

accepts connections and returns the peer address in sa. s must be a listening sockbuf. See sockbuf::listen above.

### Getting and Setting Socket Options

Socket options are used to control a socket communication. New options can be set and old value of the options can be retrieved at the protocol level or at the socket level by using setopt and getopt member functions. In addition, you can also use special member functions to get and set specific options.

In what follows,

• s is a sockbuf object

• opval is an integer and denotes the option value

• op is of type sockbuf::option and must be one of

  • sockbuf::ip_multicast_ttl is used to set the multicast TTL
  
  • sockbuf::ip_multicast_if is used to set the network interface that multicast will listen on
  
  • sockbuf::ip_multicast_loop will turn on or off the multicast loop-back support
- `sockbuf::ip_add_membership` will join a multicast group
- `sockbuf::ip_drop_membership` will drop a multicast group
- `sockbuf::so_error` used to retrieve and clear error status
- `sockbuf::so_type` used to retrieve type of the socket
- `sockbuf::so_debug` is used to specify recording of debugging information
- `sockbuf::so_reuseaddr` is used to specify the reuse of local address
- `sockbuf::so_keepalive` is used to specify whether to keep connections alive or not
- `sockbuf::so_dontroute` is used to specify whether to route messages or not
- `sockbuf::so_broadcast` is used to specify whether to broadcast `sockbuf::sock_dgram` messages or not
- `sockbuf::so_oobinline` is used to specify whether to inline *out-of-band* data or not.
- `sockbuf::so linger` is used to specify for how long to linger before shutting down
- `sockbuf::so_sndbuf` is used to retrieve and to set the size of the send buffer (communication channel buffer, not `sockbuf`'s internal buffer)
- `sockbuf::so_rcvbuf` is used to retrieve and to set the size of the recv buffer (communication channel buffer, not `sockbuf`'s internal buffer)

- `fop` is of type `sockbuf::FCnt1Cmd` and must be one of
  - `sockbuf::f_dupfd`

  Makes arg be a copy of fd, closing fd first if necessary.

  The same functionality can be more easily achieved by using dup2.

  The old and new descriptors may be used interchangeably. They share locks, file position pointers and flags; for example, if the file position is modified by using lseek on one of the descriptors, the position is also changed for the other.

  The two descriptors do not share the close-on-exec flag, however.

  On success, the new descriptor is returned.

- `sockbuf::f_getfd`
Read the close-on-exec flag. If the low-order bit is 0, the file will remain open across exec, otherwise it will be closed.

- sockbuf::f_setfd

  Set the close-on-exec flag to the value specified by arg (only the least significant bit is used).

- sockbuf::f_getfl

  Read the descriptor's flags (all flags (as set by open(2)) are returned).

- sockbuf::f_setfl

  Set the descriptor's flags to the value specified by arg. Only O_APPEND and O_NONBLOCK may be set.

  The flags are shared between copies (made with dup etc.) of the same file descriptor. The flags are shared between copies (made with dup etc.) of the same file descriptor.

  The flags and their semantics are described in open(2).

- sockbuf::f_getlk

- sockbuf::f_setlk

- sockbuf::f_setlkww

  Manage discretionary file locks.

- sockbuf::f_getown

  Get the process ID (or process group) of the owner of a socket.

  Process groups are returned as negative values.

- sockbuf::f_setown

  Set the process or process group that owns a socket.

  For these commands, ownership means receiving SIGIO or SIGURG signals.

  Process groups are specified using negative values.

- \texttt{farg} is of type \texttt{sockbuf::FCnt1Arg} and must be one of
  
  - \texttt{sockbuf::o_nonblock}

    makes the socket non-blocking
- `sockbuf::o_append`
  Set append mode

- `iocop` is of type `sockbuf::IOCtlRequest` and must be one of
  - `file`
    - `sockbuf::fionread` - get # bytes to read
    - `sockbuf::fionbio` - set/clear nonblocking i/o
    - `sockbuf::fionelex` - clear close-on-exec for fd
    - `sockbuf::fiolex` - set close-on-exec for fd
    - `sockbuf::fioasync` - set/clear asynchronous i/o
    - `sockbuf::fiosetown` - set owner
    - `sockbuf::fiogetown` - get owner
  - `socket`
    - `sockbuf::siocspgrp` - set process group
    - `sockbuf::siocpggrp` - get process group
    - `sockbuf::siocatmark` - at out-of-band mark?

  Solaris Only:
  - `sockbuf::siocshwat` - set high watermark
  - `sockbuf::siocghwat` - get high watermark
  - `sockbuf::siocslowat` - set low watermark
  - `sockbuf::siocglowat` - get low watermark

- **Routing table calls.**
  - `sockbuf::siocaddrt` - add routing table entry
  - `sockbuf::siocedlrt` - delete routing table entry
- Socket configuration controls.
  - `sockbuf::siocgifconf` - get iface list
  - `sockbuf::siocgifflags` - get flags
  - `sockbuf::siocsifflags` - set flags
  - `sockbuf::siocgifaddr` - get PA address
  - `sockbuf::siocsifaddr` - set PA address
  - `sockbuf::siocgifdstaddr` - get remote PA address
  - `sockbuf::siocsifdstaddr` - set remote PA address
  - `sockbuf::siocgifbrdaddr` - get broadcast PA address
  - `sockbuf::siocsifbrdaddr` - set broadcast PA address
  - `sockbuf::siocgifnetmask` - get network PA mask
  - `sockbuf::siocsifnetmask` - set network PA mask
  - `sockbuf::siocgifmetric` - get metric
  - `sockbuf::siocsifmetric` - set metric
  - `sockbuf::siocgifmtu` - get MTU size
  - `sockbuf::siocsifmtu` - set MTU size
  - `sockbuf::siocaddmulti` - Multicast address lists
  - `sockbuf::siocdelmulti` - 

  if not FreeBSD:
  - `sockbuf::siocgifmem` - get memory address (BSD)
  - `sockbuf::siocsifmem` - set memory address (BSD)

- ARP cache control calls.
  - `sockbuf::siocdarp` - delete ARP table entry
  - `sockbuf::siocgarp` - get ARP table entry
• sockaddr::siocsarp - set ARP table entry

if Linux:

• sockaddr::siocgifname - get iface name
• sockaddr::siocsiflink - set iface channel
• sockaddr::siocsifhwaddr - set hardware address (NI)
• sockaddr::siocgifencap - get/set slip encapsulation
• sockaddr::siocsifencap -
• sockaddr::siocsifhwaddr - Get hardware address
• sockaddr::siocgifslave - Driver slaving support
• sockaddr::siocsifslave -
• sockaddr::siogifbr - Bridging support
• sockaddr::siocsifbr - Set bridging options

- RARP cache control calls.
  • sockaddr::siodrarp - delete RARP table entry
  • sockaddr::siograrp - get RARP table entry
  • sockaddr::siocsrarp - set RARP table entry

- Driver configuration calls
  • sockaddr::siogifmap - Get device parameters
  • sockaddr::siocsifmap - Set device parameters

```c
s.GetSocket()
    return the socket id.
```

```c
s.GetType() const
    return the protocol type
```

```c
s.ttl (u_char opt = (u_char)1) const
    set the multicast TTL to opt
```
s.interface (struct in_addr *addr = NULL) const
  set the out-going multicast interface to addr

s.loop (u_char opt = (u_char)1) const
  set the multicast loop-back on or off

s.join (struct ip_mreq *mreq = NULL) const
  join multicast group defined by mreq

s.drop (struct ip_mreq *mreq = NULL) const
  drop multicast group defined by mreq

s.join (struct in_addr *multiaddr = NULL, struct in_addr *interface = NULL) const
  join multicast group defined by the multicast address multiaddr, and by the network interface interface. If interface is INADDR_ANY, then the default interface is used.

s.drop (struct in_addr *multiaddr = NULL, struct in_addr *interface = NULL) const
  drop the multicast group defined by the multicast address multiaddr, and by the network interface interface. If interface is INADDR_ANY, then the default interface is assumed.

s.FCntl(fop, farg)
  execute the file-control operation fop with argument farg using the system call fcntl().

s.IOCtl(iocop, char *arg)
  execute the I/O control operation iocop with argument arg using the system call ioctl().

s.GetNumBytesToRead()
  return the number of outstanding bytes in the socket receive buffer

s.SetNonBlocking()
  set this socket to be non-blocking

s.ClearNonBlocking()
  set this socket to be blocking

s.SetAsyncIO()
  turn on asynchronous I/O on this socket

s.ClearAsyncIO()
  turn off asynchronous I/O on this socket

s.SetCloseOnExec()
  set this socket to be closed on an exec() call

s.ClearCloseOnExec()
  set this socket not to be closed on an exec() call
s getopt(op, &opval, sizeof(opval), oplevel)

gets the option value of the sockbuf::option op at the option level oplevel in opval. It returns the actual size of the buffer opval used. The default value of the oplevel is sockbuf::sol_socket.

s setopt(op, &opval, sizeof(opval), oplevel)

sets the option value of the sockbuf::option op at the option level oplevel to opval. The default value of the oplevel is sockbuf::sol_socket.

s gettype()

gets the socket type of s. The return type is sockbuf::type.

s clearerror()

gets and clears the error status of the socket.

s debug(opval)

if opval is not -1, set the sockbuf::so_debug option value to opval. In any case, return the old option value of sockbuf::so_debug option. The default value of opval is -1.

s getreuseaddr()

return the option value of sockbuf::so_reuseaddr option.

s setreuseaddr(opval)

if opval is not -1, set the sockbuf::so_reuseaddr option value to opval. The default value of opval is -1.

s dontroute(opval)

if opval is not -1, set the sockbuf::so_dontroute option value to opval. In any case, return the old option value of sockbuf::so_dontroute option. The default value of opval is -1.

s oobinline(opval)

if opval is not -1, set the sockbuf::so_oobinline option value to opval. In any case, return the old option value of sockbuf::so_oobinline option. The default value of opval is -1.

s getbroadcast()

return the option value of sockbuf::so_broadcast option.

s setbroadcast(opval)

if opval is not -1, set the sockbuf::so_broadcast option value to opval. The default value of opval is -1.

s keepalive(opval)
if opval is not -1, set the sockbuf::so_keepalive option value to opval. In any case, return the old option value of sockbuf::so_keepalive option. The default value of opval is -1.

s.getsendbufsz()

return the old buffer size of the send buffer.

s.setsendbufsz(opval)

if opval is not -1, set the new send buffer size to opval. The default value of opval is -1.

s.getrecvbufsz()

return the old buffer size of the recv buffer.

s.setrecvbufsz(opval)

if opval is not -1, set the new recv buffer size to opval. The default value of opval is -1.

s.linger(tim)

if tim is positive, set the linger time to tim seconds. If tim is 0, set the linger off. In any case, return the old linger time if it was set earlier. Otherwise return -1. The default value of tim is -1.

**Time Outs While Reading and Writing**

Time outs are very useful in handling data of unknown sizes and formats while reading and writing. For example, how does one communicate with a socket that sends chunks of data of unknown size and format? If only sockbuf::read is used without time out, it will block indefinitely. In such cases, time out facility is the only answer.

The following idiom is recommended.

```c
int old_tmo = s.recvtimeout (2) // set time out (2 seconds here)
for (;;) { // read or write
    char buf[256];
    int rval = s.read (buf, 256);
    if (rval == 0 || rval == EOF) break;
    // process buf here
} s.recvtimeout (old_tmo); // reset time out
```

In what follows,

- s is a sockbuf object
- wp is waiting period in seconds

s.recvtimeout(wp)

sets the recv timeout to wp seconds. If wp is -1, it is a block and if wp is 0, it is a poll. It affects all
read functions. If the socket is not read ready within \( wp \) seconds, the read call will return 0. It also affects `sockbuf::underflow`. `sockbuf::underflow` will not set the `_S_EOF_SEEN` flag if it is returning EOF because of timeout. `sockbuf::recvtimeout` returns the old recv timeout value.

\[ \text{s.sendtimeout}(wp) \]

sets the send timeout to \( wp \) seconds. If \( wp \) is -1, it is a block and if \( wp \) is 0, it is a poll. It affects all write functions. If the socket is not write ready within \( wp \) seconds, the write call will return 0. `sockbuf::sendtimeout` returns the old send timeout value.

**sockAddr Class**

Class `sockAddr` is an abstract base class for all socket address classes. That is, domain specific socket address classes are all derived from `sockAddr` class.

**Note:** `sockAddr` is not spelled `sockaddr` in order to prevent name clash with `struct sockaddr` declared in `\'<sys/socket.h>\'`.

Non-abstract derived classes must have definitions for the following functions.

\[ \text{sockAddr::operator void* ()} \]

should simply return this.

\[ \text{sockAddr::size()} \]

should return size of (*this). The return type is int.

\[ \text{sockAddr::family()} \]

should return address family (domain name) of the socket address. The return type is int

**sockinetbuf Class**

`sockinetbuf` class is derived from `sockbuf` class and inherits most of the public functions of `sockbuf`. See section `sockbuf Class`, for more information on `sockbuf`. In addition, it provides methods for getting `sockinetaddr` of local and peer connections. See section `sockinetaddr Class`, for more information on `sockinetaddr`.

**Methods**

In what follows,

- \( \text{ty} \) denotes the type of the socket connection and is of type `sockbuf::type`
• proto denotes the protocol and is of type int
• si, ins are sockbuf objects and are in inet domain
• adr denotes an inet address in host byte order and is of type unsigned long
• serv denotes a service like "nntp" and is of type char*
• proto denotes a protocol like "tcp" and is of type char*
• thostname is of type char* and denotes the name of a host like "kelvin.acc.virginia.edu" or "128.143.24.31".
• portno denotes a port in host byte order and is of type int

sockinetbuf ins(ty, proto)

Constructs a sockinetbuf object ins whose socket communication type is ty and protocol is proto.proto defaults to 0.

sockinetbuf ins(si)

Constructs a sockinetbuf object ins which uses the same socket as si uses.

ins = si

performs the same function as sockbuf::operator=. See section sockbuf Class, for more details.

ins.open(ty, proto)

create a new sockinetbuf whose type and protocol are ty and proto respectively and assign it to ins.

sockinetaddr sina = ins.localaddr ()

returns the local inet address of the sockinetbuf object ins. The call will make sense only after a call to either sockbuf::bind or sockbuf::connect.

sockinetaddr sina = ins-peeraddr()

returns the peer inet address of the sockinetbuf object ins. The call will make sense only after a call to sockbuf::connect.

const char* hn = ins.localhost ()

returns the local inet thostname of the sockinetbuf object ins. The call will make sense only after a call to either sockbuf::bind or sockbuf::connect.

const char* hn = ins.peerhost ()
returns the peer *inet* hostname of the *sockinetbuf* object *ins*. The call will make sense only after a call to *sockbuf::connect*.

```cpp
int pn = ins.localport()
```

returns the local *inet* port number of the *sockinetbuf* object *ins* in host byte order. The call will make sense only after a call to either *sockbuf::bind* or *sockbuf::connect*.

```cpp
int pn = ins.peerport()
```

returns the peer *inet* port number of the *sockinetbuf* object *ins* in local host byte order. The call will make sense only after a call to *sockbuf::connect*.

```cpp
ins.passive ( char *service, int qlen )
```

binds *ins* to the current host’s address and the port *service*. For stream sockets, make the queue length to be *qlen*. It returns 0 on success and returns the *errno* on failure.

```cpp
ins.bind ()
```

binds *ins* to the default address *INADDR_ANY* and the default port. It returns 0 on success and returns the *errno* on failure.

```cpp
ins.bind (adr, portno)
```

binds *ins* to the address *adr* and the port *portno*. It returns 0 on success and returns the *errno* on failure.

```cpp
ins.bind (adr, serv, proto)
```

binds *ins* to the address, *adr* and the port corresponding to the service *serv* and the protocol *proto*. It returns 0 on success and returns the *errno* on failure.

```cpp
ins.bind (thostname, portno)
```

binds *ins* to the address corresponding to the hostname *thostname* and the port *portno*. It returns 0 on success and returns the *errno* on failure.

```cpp
ins.bind (thostname, serv, proto)
```

binds *ins* to the address corresponding to the hostname *thostname* and the port corresponding to the service *serv* and the protocol *proto*. It returns 0 on success and returns the *errno* on failure.

```cpp
ins.connect (adr, portno)
```

connects *ins* to the address *adr* and the port *portno*. It returns 0 on success and returns the *errno* on failure.

```cpp
ins.connect (adr, serv, proto)
```

connects *ins* to the address, *adr* and the port corresponding to the service *serv* and the protocol *proto*. It returns 0 on success and returns the *errno* on failure.
ins.connect (hostname, portno)

connects ins to the address corresponding to the hostname hostname and the port portno. It returns 0 on success and returns the errno on failure.

ins.connect (hostname, serv, proto)

connects ins to the address corresponding to the hostname hostname and the port corresponding to the service serv and the protocol proto. It returns 0 on success and returns the errno on failure.

**inet Datagram Sockets**

The following two programs illustrates how to use socketbuf class for datagram connection in *inet* domain. tdinread.cc also shows how to use isocketnet class and tdinwrite.cc shows how to use osocketnet class.

**tdinread.cc**

```cpp
#include <sockinet.h>

int main(int ac, char** av)
{
    isocketnet is (sockbuf::sock_dgram);
    is->bind();

    cout << "localhost = " << so.localhost() << endl
     << "localport = " << so.localport() << endl;

    char buf[256];
    int n;

    is >> n;
    cout << av[0] << ": ";
    while(n--) {
        is >> buf;
        cout << buf << ", ";
    }
    cout << endl;

    return 0;
}
```

**tdinwrite.cc**

```cpp
#include <socketnetbuf.h>
#include <stdlib.h>

int main(int ac, char** av)
```
{ 
    if (ac < 3) {
        cerr << "USAGE: " << av[0] << " thostname port-number "
             << "data ... " << endl;
        return 1;
    }

    osockinet os (sockbuf::sock_dgram);
    os->connect (av[1], atoi(av[2]));

    cout << "local: " << so.localport() << ' ' 
         << so.localhost() << endl
         << "peer: " << so.peerport() << ' ' 
         << so.peerhost() << endl;

    os << ac-3; av += 3;
    while(*av) os << *av++ << ' ';
    os << endl;
    return 0;
}

inet Stream Sockets

The following two programs illustrates the use of sockinetbuf class for stream connection in inet domain. It also shows how to use iosockinet class.

tsinread.cc

// receives strings from tsinwrite.cc and sends the strlen  
// of each string back to tsinwrite.cc
#include        <sockinet.h>

int main()
{
    sockinetbuf si(sockbuf::sock_stream);
    si.bind();

    cout << si.localhost() << ' ' << si.localport() << endl;
    si.listen();

    iosockinet s = si.accept();
    char buf[1024];

    while (s >> buf) {
        cout << buf << ' ';
        s << ::strlen(buf) << endl;
    }
    cout << endl;
    return 0;
}
tsinwrite.cc

// sends strings to tsinread.cc and gets back their length
// usage: tsinwrite hostname portno
// see the output of tsinread for what hostname and portno to use

#include <sockinet.h>
#include <stdlib.h>

int main(int ac, char** av)
{
    iosockinet sio (sockbuf::sock_stream);
    sio->connect (av[1], atoi (av[2]));

    sio << "Hello! This is a test\n" << flush;

    // terminate the while loop in tsinread.cc
    si.shutdown(sockbuf::shut_write);

    int len;
    while (s >> len) cout << len << ' ';
    cout << endl;

    return 0;
}

sockinetaddr Class

Class sockinetaddr is derived from sockAddr declared in <sockstream.h> and from sockaddr_in declared in <netinet/in.h>. Always use a sockinetaddr object for an address with inet domain of sockets. See section Establishing connections.

In what follows,

- adr denotes an inet address in host byte order and is of type unsigned long
- serv denotes a service like "nntp" and is of type char*
- proto denotes a protocol like "tcp" and is of type char*
- thostname is of type char* and denotes the name of a host like "kelvin.acc.virginia.edu" or "128.143.24.31".
- portno denotes a port in host byte order and is of type int

sockinetaddr sina
Constructs a `sockinetaddr` object `sina` with default address `INADDR_ANY` and default port number 0.

`sockinetaddr sina(adr, portno)`

Constructs a `sockinetaddr` object `sina` setting inet address to `adr` and the port number to `portno`. `portno` defaults to 0.

`sockinetaddr sina(adr, serv, proto)`

Constructs a `sockinetaddr` object `sina` setting inet address to `adr` and the port number corresponding to the service `serv` and the protocol `proto`. The protocol defaults to "tcp".

`sockinetaddr sina(thostname, portno)`

Constructs a `sockinetaddr` object `sina` setting inet address to the address of `thostname` and the port number to `portno`. `portno` defaults to 0.

`sockinetaddr sina(thostname, serv, proto)`

Constructs a `sockinetaddr` object `sina` setting inet address to the address of `thostname` and the port number corresponding to the service `serv` and the protocol `proto`. The protocol defaults to "tcp".

`void* a = sina` returns the address of the `sockaddr_in` part of `sockinetaddr` object `sina` as void*.

`int sz = sina.size()` returns the size of `sockaddr_in` part of `sockinetaddr` object `sina`.

`int af = sina.family()` returns `sockinetbuf::af_inet` if all is well.

`int pn = sina.getport()` returns the port number of the `sockinetaddr` object `sina` in host byte order.

`const char* hn = gethostname()` returns the host name of the `sockinetaddr` object `sina`.

**sockstream Classes**

Sockstream classes are designed in such a way that they provide the same interface as their stream counterparts do. We have `isockstream` derived from `istream` and `osockstream` derived from `ostream`. We also have `iosockstream` which is derived from `iostream`. 
Each domain also has its own set of stream classes. For example, inet domain has isockinet, osockinet, and iosockinet.

**isockstream Class**

Since isockstream is publicly derived from istream, most of the public functions of istream are also available in isockstream.

isockstream redefines rdbuf() defined in its virtual base class ios. Since, ios::rdbuf() is not virtual, care must be taken to call the correct rdbuf() through a reference or a pointer to an object of class isockstream.

In what follows,

- sb is a sockbuf object
- sbp is a pointer to a sockbuf object

isockstream is(sb)

Constructs an isockstream object is with sb as its sockbuf.

isockstream is(sbp)

Constructs an isockstream object is with *sbp as its sockbuf.

sbp = is.rdbuf()

returns a pointer to the sockbuf of the isockstream object is.

isockstream::operator -> ()

returns a pointer to the isockstream's sockbuf so that the user can use isockstream object as a sockbuf object.

is->connect (sa); // same as is.rdbuf()->connect (sa);

**osockstream Class**

Since osockstream is publicly derived from ostream, most of the public functions of ostream are also available in osockstream.

osockstream redefines rdbuf() defined in its virtual base class ios. Since, ios::rdbuf() is not virtual, care must be taken to call the correct rdbuf() through a reference or a pointer to an object of class osockstream.

In what follows,

- sb is a sockbuf object
- sbp is a pointer to a sockbuf object

iosockstream os(sb)

Constructs an iosockstream object os with sb as its sockbuf.

iosockstream os(sbp)

Constructs an iosockstream object os with *sbp as its sockbuf.

sbp = os.rdbuf()

returns a pointer to the sockbuf of the iosockstream object os.

iosockstream::operator -> ()

returns a pointer to the iosockstream's sockbuf so that the user can use iosockstream object as a sockbuf object.

    os->connect (sa); // same as os.rdbuf()->connect (sa);

**iosockstream Class**

Since iosockstream is publicly derived from iostream, most of the public functions of iostream are also available in iosockstream.

iosockstream redefines rdbuf() defined in its virtual base class ios. Since, ios::rdbuf() is not virtual, care must be taken to call the correct rdbuf() through a reference or a pointer to an object of class iosockstream.

In what follows,

- sb is a sockbuf object

- sbp is a pointer to a sockbuf object

iosockstream io(sb)

Constructs an iosockstream object io with sb as its sockbuf.

iosockstream io(sbp)

Constructs an iosockstream object io with *sbp as its sockbuf.

sbp = io.rdbuf()

returns a pointer to the sockbuf of the iosockstream object io.

iosockstream::operator -> ()

returns a pointer to the iosockstream's sockbuf so that the user can use iosockstream object as
a sockbuf object.

    io->connect (sa); // same as io.rdbuf()->connect (sa);

**iosockinet Stream Classes**

We discuss only isockinet class here. osockinet and iosockinet are similar and are left out. However, they are covered in the examples that follow.

**isockinet**

isockinet is used to handle interprocess communication in *inet* domain. It is derived from isockstream class and it uses a sockinetbuf as its stream buffer. See section **iosockstreams**, for more details on isockstream. See section **sockinetbuf Class**, for information on sockinetbuf.

In what follows,

- **ty** is a sockbuf::type and must be one of sockbuf::sock_stream, sockbuf::sock_dgram, sockbuf::sock_raw, sockbuf::sock_rdm, and sockbuf::sock_seqpacket
- **proto** denotes the protocol number and is of type int
- **sb** is a sockbuf object and must be in *inet* domain
- **sinp** is a pointer to an object of sockinetbuf

isockinet is (ty, proto)

constructs an isockinet object is whose sockinetbuf buffer is of the type ty and has the protocol number proto. The default protocol number is 0.

isockinet is (sb)

constructs a isockinet object is whose sockinetbuf is sb. sb must be in *inet* domain.

isockinet is (sinp)

constructs a isockinet object is whose sockinetbuf is sinp.

sinp = is.rdbuf ()

returns a pointer to the sockinetbuf of isockinet object is.

isockinet::operator ->

returns sockinetbuf of sockinet so that the sockinet object acts as a smart pointer to sockinetbuf.

    is->localhost (); // same as is.rdbuf ()->localhost ();
**iosockinet examples**

The first pair of examples demonstrates datagram socket connections in the **inet** domain. First, **tdinread** prints its local host and local port on stdout and waits for input in the connection. **tdinwrite** is started with the local host and local port of **tdinread** as arguments. It sends the string "How do ye do!" to **tdinread** which in turn reads the string and prints on its stdout.

```cpp
// tdinread.cc
#include <sockinet.h>

int main ()
{
    char buf[256];
    isockinet is (sockbuf::sock_dgram);
    is->bind ();
    cout << is->localhost() << ' ' << is->localport() << endl;
    is.getline (buf);
    cout << buf << endl;
    return 0;
}
```

```cpp
// tdinwrite.cc--tdinwrite hostname portno
#include <sockinet.h>
#include <stdlib.h>

int main (int ac, char** av)
{
    osockinet os (sockbuf::sock_dgram);
    os->connect (av[1], atoi(av[2]));
    os << "How do ye do!" << endl;
    return 0;
}
```

The next example communicates with an **nntp** server through a **sockbuf::sock_stream** socket connection in **inet** domain. After establishing a connection to the **nntp** server, it sends a "HELP" command and gets back the HELP message before sending the "QUIT" command.

```cpp
// tnntp.cc
#include <sockinet.h>

int main ()
{
    char buf[1024];
    iosockinet io (sockbuf::sock_stream);
    io->connect ("murdoch.acc.virginia.edu", "nntp", "tcp");
    io.getline (buf, 1024); cout << buf << endl;
    io << "HELP\r\n" << flush;
    io.getline (buf, 1024); cout << buf << endl;
    while (io.getline (buf, 1024))
```
if (buf[0] == '.' && buf[1] == '\r') break;
else if (buf[0] == '.' && buf[1] == '.') cout << buf+1 << endl;
else cout << buf << endl;
io << "QUIT\r\n" << flush;
io.getline (buf, 1024); cout << buf << endl;
return 0;
Index

- .map .................................. 24
  .xbm .................................. 24

  _S_DELETE_DONT_CLOSE.............. 59

  A
  accept .................................. 64
  Act_As_Gateway ........................ 17
  Advertise interval ..................... 20
  af inet .................................... 79
  Alarm interval .......................... 20
  angle .................................. 44, 46, 47, 48, 49, 50, 51
  append .................................. 52
  apply .................................... 54
  area .................................... 44, 47
  ARP table entry ......................... 68, 69
  assign .................................... 53
  asynchronous i/o ........................ 67
  Auto Update ................................ 11
  Automatically Update User Position ... 11

  B
  back .................................... 52
  Begin ..................................... 40
  bind ....................................... 64, 75
  Bind ....................................... 32
  Bridging support ........................ 69
  bucket_sort ................................ 54
  BufferSize ................................ 40
  Buttons along the top ................... 11

  C
  Cache_Prune_Period ...................... 16
  center .................................. 11, 12, 18, 24, 36, 44, 45, 46
  Center .................................... 12
  Center_Latitude ........................ 24
  Center_Longitude ........................ 24
  CharFill .................................. 39
  Check_Interval .......................... 17
  ChunkFill .................................. 39
  circle  2, 11, 16, 18, 19, 20, 23, 36, 37, 45, 46, 47
  Circle .................................... 11, 18, 30, 36, 37
  clear ..................................... 54
  ClearAsyncIO ............................. 70
  ClearCloseOnExec ........................ 70
  clearerror ................................ 71
  ClearNonBlocking ........................ 70
  close ..................................... 59
  close-on-exec ............................ 66, 67
  cmp_segments_at_xcoord .................. 50
  cmp_slopes ................................ 50
  cocircular ................................ 45
  collinear .................................. 44, 45, 46
  COM ....................................... 21
  conc ...................................... 53
  configuration 4, 6, 7, 9, 15, 16, 20, 22, 24, 68, 69
  connect ................................... 64, 75
  Connect .................................... 32
  Contains .................................. 26
  contains 20, 27, 36, 43, 46, 49, 51, 52
  contents .................................. 52
  Controller ................................ 18
  Coordinates ................................ 11
  Creation i, 2, 27, 31, 36, 37, 43, 45, 47, 48, 50, 51
  cyclic_pred .............................. 52
  cyclic_succ ................................ 52

  D
  datagram ................................ 32, 35, 76, 83
  debug ..................................... 71
  Default .................................... 18
  Definitioni 2, 24, 27, 31, 36, 37, 43, 45, 47, 48, 50, 51
  Degree_Height ............................ 24
  Degree_Width ............................ 24
getport ........................................... 79  
GetPort ........................................... 79  
GeoMesg ........................................... 30  
GeoSocket ........................................... 31  
GetPos ............................................. 34  
GetPriority ...................................... 29  
getrecvbufsz...................................... 72  
getreuseaddr ..................................... 71  
getsendbufsz ..................................... 72  
GetSenderAddr  
GeoMesg ........................................... 30  
GeoSocket ........................................... 32  
GetShape .......................................... 28  
GeoMesg ........................................... 30  
GeoSocket ........................................... 32  
GetShapeType .................................... 37  
GetSocket ......................................... 69  
GetSpecMsgInfo .................................. 34  
gethostname ..................................... 79  
GetTopRight ..................................... 28, 37  
gettype ............................................ 71  
GetType ........................................... 29, 69  
GetVersion ....................................... 29  
GPS .................................................. i, 3, 9, 21, 22, 34  
ip_multicast_if ................................... 64  
ip_multicast_loop .................................. 64  
ip_multicast_ttl ................................... 64  
is_degenerate .................................... 46  
is_eof ............................................. 61  
is_exceptionpending  
is_horizontal .................................... 49  
is_open ........................................... 59, 61  
is_readyready ................................... 62  
is_trivial ......................................... 46, 49  
is_vertical ....................................... 49  
is_writeready ................................... 62  
isockinet ........................................ 2, 57, 76, 80, 82, 83  
isockstream ...................................... 2, 79, 80, 82  
IterativeServer .................................. 16, 23  
join ............................................... 70  
keepalive ........................................... 71, 72  
last ............................................... 52  
Last of multicast .................................. 20  
latitude ........................................... 9, 20, 21, 23, 24, 34  
Latitude .......................................... 23  
LD_LIBRARY_PATH ................................ 4  
LEDA ............................................... 1, 43  
left_tangent ..................................... 46  
left_turn .......................................... 45  
length .............................................. 48, 49, 50, 51, 52, 54, 55, 75, 78  
Life: ............................................... 12  
lifetime ........................................... 4, 8, 9, 10  
linger .............................................. 65, 72  
Linux .............................................. 69  
listen ............................................... 64  
Local_Addr ........................................ 19  
localaddr .......................................... 74  
localhost ........................................... 74, 76, 77, 82, 83  
localport .......................................... 75, 76, 77, 83  
LongFill .......................................... 39  
longitude .......................................... 9, 20, 21, 23, 24, 34  
Longitude ......................................... 23  
Look_For_Interfaces  
loop ............................................... 70  
Map area .......................................... 12  
Map Display ....................................... 12  
Map Movement ..................................... 12  
max ............................................... 54  
Max_Cache_Age ................................... 16  
Max_WaitTime .................................... 17
<table>
<thead>
<tr>
<th>Operation</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>MaxPacketSize</td>
<td>17</td>
</tr>
<tr>
<td>Mcast:</td>
<td>12</td>
</tr>
<tr>
<td>memory address</td>
<td>68</td>
</tr>
<tr>
<td>merge</td>
<td>54</td>
</tr>
<tr>
<td>Message Line along the bottom</td>
<td>13</td>
</tr>
<tr>
<td>Message Text Area</td>
<td>12</td>
</tr>
<tr>
<td>MessageCreate</td>
<td>28</td>
</tr>
<tr>
<td>MessageParse</td>
<td>28</td>
</tr>
<tr>
<td>metric</td>
<td>18, 68</td>
</tr>
<tr>
<td>Metric</td>
<td>18</td>
</tr>
<tr>
<td>mhd</td>
<td>5, 6, 7, 9, 21, 31, 35</td>
</tr>
<tr>
<td>midpoint</td>
<td>44</td>
</tr>
<tr>
<td>min</td>
<td>54</td>
</tr>
<tr>
<td>Min_WaitTime</td>
<td>17</td>
</tr>
<tr>
<td>Mouse</td>
<td>12</td>
</tr>
<tr>
<td>Mouse Actions</td>
<td>11</td>
</tr>
<tr>
<td>move_to_front</td>
<td>53</td>
</tr>
<tr>
<td>move_to_rear</td>
<td>53</td>
</tr>
<tr>
<td>msg_dontroute</td>
<td>60, 61</td>
</tr>
<tr>
<td>msg_oob</td>
<td>60, 61</td>
</tr>
<tr>
<td>msg_peek</td>
<td>60, 61</td>
</tr>
<tr>
<td>MTU</td>
<td>68</td>
</tr>
<tr>
<td>multicast</td>
<td>17, 19</td>
</tr>
<tr>
<td>multicast address for control messages from the router</td>
<td>20</td>
</tr>
<tr>
<td>Multicast_Addr</td>
<td>19</td>
</tr>
<tr>
<td>MulticastDrop</td>
<td>32</td>
</tr>
<tr>
<td>MulticastJoin</td>
<td>32</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>16</td>
</tr>
<tr>
<td>Net</td>
<td>18</td>
</tr>
<tr>
<td>NolIntersectContain</td>
<td>36</td>
</tr>
<tr>
<td>nonblocking</td>
<td>67</td>
</tr>
<tr>
<td>norm</td>
<td>50</td>
</tr>
<tr>
<td>NoType</td>
<td>29, 30, 36, 37</td>
</tr>
<tr>
<td>Number of addresses to follow</td>
<td>20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>O</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>o_append</td>
<td>67</td>
</tr>
<tr>
<td>O_APPEND</td>
<td>66</td>
</tr>
<tr>
<td>o_nonblock</td>
<td>66</td>
</tr>
<tr>
<td>O_NONBLOCK</td>
<td>66</td>
</tr>
<tr>
<td>oobline</td>
<td>71</td>
</tr>
<tr>
<td>open</td>
<td>58, 74</td>
</tr>
<tr>
<td>Open</td>
<td>11</td>
</tr>
<tr>
<td>Operations, 2, 28, 31, 36, 38, 44, 46, 47, 48, 50, 51, 52</td>
<td>11</td>
</tr>
<tr>
<td>operator -&gt;</td>
<td>80, 81, 82</td>
</tr>
<tr>
<td>operator bool</td>
<td>38</td>
</tr>
<tr>
<td>operator char*</td>
<td>38</td>
</tr>
<tr>
<td>operator const char*</td>
<td>38</td>
</tr>
<tr>
<td>operator const void*</td>
<td>38</td>
</tr>
<tr>
<td>operator void*</td>
<td>38, 73</td>
</tr>
<tr>
<td>orientation</td>
<td>44, 45, 46, 50</td>
</tr>
<tr>
<td>osockinet</td>
<td>57, 76, 77, 80, 82, 83</td>
</tr>
<tr>
<td>osostream</td>
<td>2, 79, 80, 81</td>
</tr>
<tr>
<td>outcircle</td>
<td>45</td>
</tr>
<tr>
<td>outside</td>
<td>45, 46, 47</td>
</tr>
<tr>
<td>overflow</td>
<td>63</td>
</tr>
<tr>
<td>owner</td>
<td>66, 67</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PA address</td>
<td>68</td>
</tr>
<tr>
<td>PA mask</td>
<td>68</td>
</tr>
<tr>
<td>packetinet</td>
<td>6, 7, 8, 9, 15, 19, 20, 22, 23, 34</td>
</tr>
<tr>
<td>parameter</td>
<td>75</td>
</tr>
<tr>
<td>PCMCIA</td>
<td>1, 21</td>
</tr>
<tr>
<td>PEER AREA</td>
<td>5</td>
</tr>
<tr>
<td>peeraddr</td>
<td>74</td>
</tr>
<tr>
<td>peerhost</td>
<td>74, 77</td>
</tr>
<tr>
<td>peerport</td>
<td>75, 77</td>
</tr>
<tr>
<td>permute</td>
<td>54</td>
</tr>
<tr>
<td>perpendicular</td>
<td>49</td>
</tr>
<tr>
<td>Pixel_Height</td>
<td>24</td>
</tr>
<tr>
<td>Pixel_Width</td>
<td>24</td>
</tr>
<tr>
<td>point</td>
<td>9, 11, 18, 19, 20, 27, 28, 30, 32, 34, 35, 36, 37, 43, 44, 45, 46, 47, 48, 49, 50</td>
</tr>
<tr>
<td>Point</td>
<td>11, 18, 28, 30, 36, 37, 43, 44</td>
</tr>
<tr>
<td>point1</td>
<td>46</td>
</tr>
<tr>
<td>point2</td>
<td>46</td>
</tr>
<tr>
<td>point3</td>
<td>46</td>
</tr>
<tr>
<td>polygon</td>
<td>4, 10, 11, 18, 23, 27, 28, 29, 32, 33, 34, 36, 37, 47, 48</td>
</tr>
<tr>
<td>Polygon</td>
<td>11</td>
</tr>
<tr>
<td>Polygon Type</td>
<td>52</td>
</tr>
<tr>
<td>pop</td>
<td>52</td>
</tr>
<tr>
<td>Pop</td>
<td>53</td>
</tr>
<tr>
<td>pop_back</td>
<td>53</td>
</tr>
<tr>
<td>pop_front</td>
<td>53</td>
</tr>
<tr>
<td>Port</td>
<td>5, 12, 16, 17, 19, 23</td>
</tr>
<tr>
<td>Port:</td>
<td>12</td>
</tr>
<tr>
<td>position</td>
<td>21</td>
</tr>
<tr>
<td>pred</td>
<td>52</td>
</tr>
<tr>
<td>print</td>
<td>15, 19, 21, 22, 23, 51, 55</td>
</tr>
<tr>
<td>process group</td>
<td>66, 67</td>
</tr>
<tr>
<td>Program</td>
<td>16, 22</td>
</tr>
<tr>
<td>push</td>
<td>52</td>
</tr>
<tr>
<td>push_back</td>
<td>52</td>
</tr>
<tr>
<td>push_front</td>
<td>52</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>radius</td>
<td>18, 20, 23, 36, 45, 46</td>
</tr>
<tr>
<td>Radius</td>
<td>23</td>
</tr>
<tr>
<td>rank</td>
<td>52, 55</td>
</tr>
<tr>
<td>Rank</td>
<td>19</td>
</tr>
<tr>
<td>RARP table entry</td>
<td>69</td>
</tr>
<tr>
<td>rdbuf</td>
<td>80, 81, 82</td>
</tr>
</tbody>
</table>
## T

<table>
<thead>
<tr>
<th>Table_Refresh_Period</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table_Update_Period</td>
<td>16</td>
</tr>
<tr>
<td>tail</td>
<td>52</td>
</tr>
<tr>
<td>target</td>
<td>30, 32, 48, 49, 50</td>
</tr>
<tr>
<td>Time Outs</td>
<td>2, 60, 61, 72</td>
</tr>
<tr>
<td>Timeout_Max</td>
<td>16</td>
</tr>
<tr>
<td>Timer_Rate</td>
<td>17</td>
</tr>
<tr>
<td>Title</td>
<td>23</td>
</tr>
<tr>
<td>To</td>
<td>12</td>
</tr>
<tr>
<td>translate</td>
<td>44, 46, 47, 49</td>
</tr>
<tr>
<td>translate_by_angle</td>
<td>44</td>
</tr>
<tr>
<td>Trimble_Navigation's</td>
<td>21</td>
</tr>
<tr>
<td>ttl69</td>
<td>7, 16, 17, 19, 64, 69</td>
</tr>
<tr>
<td>Tunnel</td>
<td>19</td>
</tr>
<tr>
<td>Tunnel_Init_Timeout</td>
<td>16</td>
</tr>
<tr>
<td>Tunnels_to_GeoNodes</td>
<td>18</td>
</tr>
</tbody>
</table>

## U

| underflow           | 62, 63, 73 |
| unique              | 53, 54, 63 |
| unite               | 47 |
| Urgent              | 12 |
| URGENT              | 29 |
| Url                 | 23 |
| User                | 11 |

## V

| vector              | 2, 43, 44, 46, 47, 48, 49, 50, 51 |
| vertices            | 4, 37, 47, 48 |

## W

| watermark           | 67 |
| write               | 60, 61 |
| Write               | 33 |

## X

<p>| x_proj              | 49 |
| xcoord              | 44, 49 |
| xcoord1             | 48, 49 |
| xcoord2             | 49 |
| xdist               | 44 |
| xsputn              | 63 |</p>
<table>
<thead>
<tr>
<th></th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>X-Windows bitmap</td>
<td>24</td>
</tr>
<tr>
<td>y_abs</td>
<td>49</td>
</tr>
<tr>
<td>y_proj</td>
<td>49</td>
</tr>
<tr>
<td>ycoord</td>
<td>44, 49</td>
</tr>
<tr>
<td>ycoord1</td>
<td>49</td>
</tr>
<tr>
<td>ycoord2</td>
<td>49</td>
</tr>
<tr>
<td>ydist</td>
<td>44</td>
</tr>
<tr>
<td>ydist</td>
<td>44</td>
</tr>
<tr>
<td>Zoom-in</td>
<td>11</td>
</tr>
<tr>
<td>Zoom-out</td>
<td>11</td>
</tr>
</tbody>
</table>