ActivMEDIA INC

REAL WORLD

Saphira

Software Manual

Saphira Version 5.3

Saphira Manual Version 5.3, March, 1997.

Contents

List of Tables

List of Figures

1. Saphira Software & Resources

This Software Manual provides the general and technical details you will need to program and operate your Real World Interface, Inc. (RWI) Pioneer 1 (available through *Activ*Media, Inc.), B14, B21, or ATRV-1 Mobile Robot with Saphira software.

1.1. Saphira Client/Server

Saphira is a robotics application development environment written, maintained, and constantly updated at SRI International's (formerly Stanford Research Institute) Artificial Intelligence Center, notably under the direction of Dr. Kurt Konolige, who developed the Pioneer mobile robot platform.

Saphira operates in a client/server environment. The Saphira library is a set of routines for building clients. These routines perform most of the thankless work of communications and housekeeping for the robot server. And the Saphira library integrates a number of useful functions for sending commands to the server, gathering information from the robot's sensors, and packaging them for display in a graphical window-based user interface. In addition, Saphira supports higher-level functions for robot control and sensor interpretation, including fuzzy-control behavior and reactive planning systems, and a map-based navigation and registration system.

All the Saphira client expects is that the robot has the basic components for robotics sensing and navigation, including drive motors and wheels, position encoders, and sensors. Saphira also expects that the robot support some, albeit little, onboard intelligence to handle the low-level details of robot sensor and drive management, and to be able to send that information and respond to Saphira commands—act as a server—through a special communications packet protocol we describe in detail in Chapter 4. Some of the server details are robot-specific, so we encourage you to consult your robot's operation manual and Saphira supplementary materials for details, as well.

The Saphira client library is available for Microsoft Windows NT and 95 and for UNIX with Motif (SunOS, Solaris, SGI, FreeBSD, and Linux). Saphira sources and libraries are written in ANSI C. There is an Application Programmer's Interface (API) that calls functions in the Saphira system so you can write an application that defines new robot programs, and link it in to the Saphira library. Programming details are in the following chapters of this manual.

1.2. Robot Simulator

Saphira also comes with a software simulator of your physical robot and its environment. This feature allows you to debug your applications conveniently on your computer.

The simulator has realistic error models for the sonar sensors and wheel encoders. Even its communication interface is the same as for a physical robot, so you won't need to reprogram or make any special changes to the client to have it run with either the real robot or the simulator. But unlike the real thing, the simulator has a single-step mode which lets you examine each and every step of your program in detail.

The simulator also lets you construct 2-D models of real or imagined environments, called *worlds*. World models are abstractions of the real world, with linear segments representing the vertical surfaces of corridors, hallways, and the objects in them. Because the 2-D world models are only an abstraction of the real world, we encourage you to refine your client software using the real robot in a real world environment.

1.3. Required and Optional Components

The following is a list of components that you'll need, as well as some options you may desire, to operate your robot with Saphira. Consult your mobile robot's Operation Manual for component details.

✓ Mobile robot with Saphira-enabled servers

- ✓ Radio modems or Ethernet radio bridge (optional)
- ✓ Computer: Power Macintosh[1]; Pentium, Pentium Pro, or 486-class PC with Microsoft Windows 95 or NT, FreeBSD, or Linux operating system; or UNIX workstation
- ✓ Open communication port (TCP/IP or serial)
- ✓ Four to five megabytes of hard-disk storage
- ✓ PKUNZIP (PCs), GUNZIP (PCs and UNIX), StuffIt Lite, or compatible archive-decompression software
- ✓ Motif GUI and libraries for FreeBSD/Linux/UNIX
- ✓ C- program source file editor and compiler. Note: Current Windows95/NT version of Saphira supports only Microsoft's Visual C/C++ software, not Borland's Turbo-C/C++ products.

1.4. Saphira Client Installation

The Saphira distribution software, including the **saphira** demonstration program, simulator, and accompanying C libraries, come stored as a compressed archive of directories and files either on a 3.5-inch, 1.4 MB floppy diskette, or at the RWI Internet site. Each archive is configured and compiled for a particular operating system, such as Windows95/NT or Solaris. Choose the version that matches your client computer system. You may obtain additional Saphira archives for other platforms and updates from the RWI Internet site; see *Additional Resources* below in this chapter for details.

The Windows95/NT versions typically are PKZIP'd, and UNIX versions come GZIP'd and TAR'd. To decompress the software into useable files, you will need the appropriate decompression/archive software: pkunzip, gunzip, or compatible program. For proper operation, consult the respective program's user manual or help files.

For Linux and other UNIX users, we recommend that you create a **saphira** directory in **/usr/local**, and set the appropriate permissions for access and use by your robotics groups. Copy the Saphira archive to that directory, then uncompress and untar the Saphira archive. For example, with Linux:

```
tar –zxvf saphira–linux–5.3.gz
```

For Windows95/NT or Macintosh, similarly uncompress the ZIP or SIT archive, respectively, but the location of the files is up to you. The recommended directory is **c:\saphira**, which means the toplevel Saphira directory will be **c:\saphira\ver53**. This is the directory that the sample MSVC projects assume.

For all systems upon decompression, a hierarchy of folders and files will appear inside a newly established, version-related Saphira directory; **ver53** for Saphira version 5.3, for example. The distribution directory for the Windows95/NT Saphira version 5.3 looks like this:

```
ver53/
  bin/
    saphira.exe          Saphira controller example
    direct.exe           direct motion control example
    pioneer.exe          simulator
    btech.exe            Pioneer Fast-Track Vision system demo
    bgram                behavior grammar compiler
    sf.dll               DLL library for MS 95/NT
    msvcrt40.dll         required MS Windows DLL
  handler/
    src/
      samples/           tutorial examples
      apps/              application source examples
      basic/
        behavior.beh     behavior examples
```

---

[1] We do not recommend using Macintosh for Saphira development at this time because the native operating system (System 7.5) does not fully support multitasking which is essential for Saphira operation.

```
    include/              header files
    obj/                  library files
  maps/                   Saphira example maps
  worlds/                 simulator world files
  params/                 parameters for different robots
  readme                  explanation text file
  update                  comparison of version s
  license                 operation license
```

The uncompressed Saphira software typically requires 20 megabytes of hard-disk drive storage space.

IMPORTANT NOTICE!

All Saphira operations require that the environment variable SAPHIRA be set to the top-level directory, e.g., **/usr/local/saphira/ver53** on a Unix system (note there is no final slash), or **`c:\saphira\ver53`** on an MS Windows system. *If you do not set this variable correctly, Saphira clients and the simulator will fail to work, or fail to work properly!* Please set this as soon as you install the distribution.

If you have a previous installation of Saphira, your SAPHIRA environment variable will be set to the old toplevel directory. YOU MUST RESET IT to the toplevel directory of the new distribution. All new clients will complain and fail to execute until you do.

Unix systems should use one of the following methods, preferably in the user's **.cshrc** or other default shell script parameter file.

```
export SAPHIRA=/usr/local/saphira/ver53     (bash shell)
setenv SAPHIRA /usr/local/saphira/ver53     (csh shell)
```

In Windows 95 and NT 3.51, assuming the top-level Saphira directory is **c:\saphira\ver53**, add the following line to the **C:\AUTOEXEC.BAT** file:

```
SET SAPHIRA=C:\saphira\ver53
```

Finally, in Windows NT 4.0, go to Start/Settings/System, and click on the Environment tab. Add the SAPHIRA variable in either the user or system-wide settings.

1.5. Saphira Quick Start

To start the Saphira client demonstration program, navigate to inside the **`bin/`** directory and execute the program named *`saphira(.exe)`*. For instance, with the mouse, double-click the **`saphira.exe`** icon inside the **`saphira/ver53/bin/`** folder on your Windows95 desktop.

With UNIX, you must be running the X-Window system to execute the Saphira client software and make sure to **`export`** or **`setenv`** the SAPHIRA=*path* parameter.

Have a robot server or the simulator readied for a Saphira connection. For example, execute the **`saphira/ver53/bin/pioneer(.exe)`** robot simulator on the same computer, or simply turn on your Pioneer robot and connect its serial port (or radio modems) to your basestation computer running the Saphira demonstration program.

In the Saphira client **`main`** window, draw down the **`Connect`** menu and choose the connection port where the robot server is listening. For example, with the Pioneer robot choose the serial port **`cua0`** or **`cua1,`** or for a B14 Saphira server connected to your Ethernet network via TCP, select the "TCP" connection option in the Saphira demo and provide the robot's IP address or hostname when requested.

Once you initiate the connection, the Saphira client and robot server perform a synchronization routine and, if successful, will establish a connection. We provide a number of clues on both the client and server so you can follow the synchronization process. Success is distinct: The Saphira main window comes alive with sonar readings, and the robot's sonars begin a rhythmic, audible ticking.

We detail Saphira client operation in the following chapter. For now, we leave it to you to find the manual drive keys and take your robot for a joyride (hints: arrows move and the spacebar stops the motors).

1.6. *Additional Resources*

Every new Saphira customer gets three additional and valuable resources: a private account on RWI's Internet server for download of Saphira software, updates, and manuals; the opportunity to register on one or more of private robotics newsgroups; and email access to the Saphira support team.

1.6.1. FTP Software Archive

RWI has a server connected full-time to the Internet where customers may obtain Saphira software and support materials. Access is restricted to RWI/*Activ*Media customers, including Pioneer 1, Bxx, and ATRV-1 owners.

The RWI server name is:

`ftp.rwii.com`

To gain access, use the username and password that are written on the *Registration and Account Sheet* that accompanied this manual.

Saphira software, as well as the variety of support literature, including this manual, currently are stored in subdirectories under the pathname:

`pub/robots/Pioneer/Saphira`

Consult your computer system manual for connection software and operation for downloading files via the UNIX file transfer protocol **`(ftp`**) or equivalent service.

1.6.2. Saphira Newsgroup

RWI/*Activ*Media also maintain several special email-based newsgroups for robot owners to share ideas, software, and questions. To find out more about these special newsgroups, send an email message with your reply email address as follows.

To: **majordomo@rwii.com**
From: ***<your return email address goes here>***
Subject: **help** (Subject: always ignored)

(body of message—choose one or more commands:)
help (returns instructions)
lists (returns list of newsgroups)
subscribe <list name here> (waddayatink?)
unsubscribe <list name here> (ditto)
end

The groups currently are unmoderated, so please confine your comments and inquiries to those concerning robot operation and programming.

1.6.3. SRI Saphira Web Pages

Saphira is under continuing active development at SRI International. SRI maintains a set of web pages with more information about Saphira, including

- tutorials and other documentation on various parts of Saphira

- class projects from Stanford CS327B, *Real-World Autonomous Systems*
- information about SRI robots and projects that use Saphira, including the integration of Saphira with SRI's Open Agent Architecture
- links to other sites using Pioneer robots and Saphira

The entry to the SRI Saphira web pages is *http://www.ai.sri.com/~konolige/saphira*.

1.6.4. Support

Have a problem? Can't find the answer in this or any of the accompanying manuals? Or know a way that we might improve our robots and software? Share your thoughts and questions directly with us:

`support@rwii.com`

Your message goes to our team of developers who will help you directly or point you to where you may find help. Because this is a support option, not a general-interest newsgroup, we must reserve the option to reply only to questions about bugs or problems with RWI-manufactured robots or Saphira.

1.6.5. Acknowledgments

The Saphira system reflects the work of many people at SRI, starting with Stan Rosenschein, Leslie Kaelbling, and Stan Reifel, who built and programmed Flakey in the mid 1980's. Major contributions have been made by Alessandro Saffiotti, Karen Myers, Enrique Ruspini, Didier Guzzoni, and many others.

2. Saphira System Overview

Saphira is an architecture for mobile robot control. It was originally developed for the research robot Flakey[2] at SRI International, and after being in use for over 10 years has evolved into an architecture that supports a wide variety of research and application programming for mobile robotics. Saphira and Flakey appeared in the October, 1994 show *Scientific American Frontiers* with Alan Alda. Saphira and the Pioneer robots placed first in the AAAI robot competition "Call a meeting" in August 1996, and will also appear in an April, 1997 segment of the same program.[3]

The Saphira system can be thought of as two architectures, one built on top of the other. The *system architecture* is an integrated set of routines for communicating with and controlling a robot from a host computer. The system architecture is designed to make it easy to define robot applications by linking in client programs. Because of this, the system architecture is an *open architecture*. Users who wish to write their own robot control systems, but don't want to worry about the intricacies of hardware control and communication, can take advantage of the *micro-tasking* and *state reflection* properties of the system architecture to bootstrap their applications. For example, a user who is interested in developing a novel neural network control system might work at this level.

On top of the system routines is a *robot control architecture*, that is, a design for controlling mobile robots that addresses many of the problems involved in navigation, from low-level control of motors and sensors to high-level issues such as planning and object recognition. Saphira's control architecture contains a rich set of representations and routines for processing sensory input, building world models, and controlling the actions of the robot. As with the system architecture, the routines in the control architecture are tightly integrated to present a coherent framework for robot control. The control architecture is flexible enough so that users may pick among various methods for achieving an objective, for example, using a fuzzy control regime vs. more direct control of the motors. It is also an *open architecture*, as users may substitute their own methods for many of the pre-defined routines, or add new functionality and share it with other research groups.

In this section we'll give a brief overview of the two architectures, as well as the main concepts of Saphira. More in-depth information can be found in the documentation at the SRI Saphira web site (*http://www.ai.sri.com/~konolige/saphira*).

2.1. System Architecture

Think of Saphira's system architecture as the basic operating system for robot control. Figure 2-1 shows the structure for a typical Saphira application. Saphira routines are in blue, user routines in red. Saphira routines are all micro-tasks that are invoked every Saphira cycle (100 ms) by Saphira's built-in micro-tasking OS. These routines handle packet communication with the robot, build up an internal picture of the robot's state, and perform more complex tasks such as navigation and sensor interpretation.

---

[2] See *http://www.ai.sri.com/people/flakey* for a description of Flakey and further references.

[3] A write-up of this event is in AI Magazine, Spring 1997 (for a summary see *http://www.ai.sri.com/~konolige/saphira/aaai.html*).

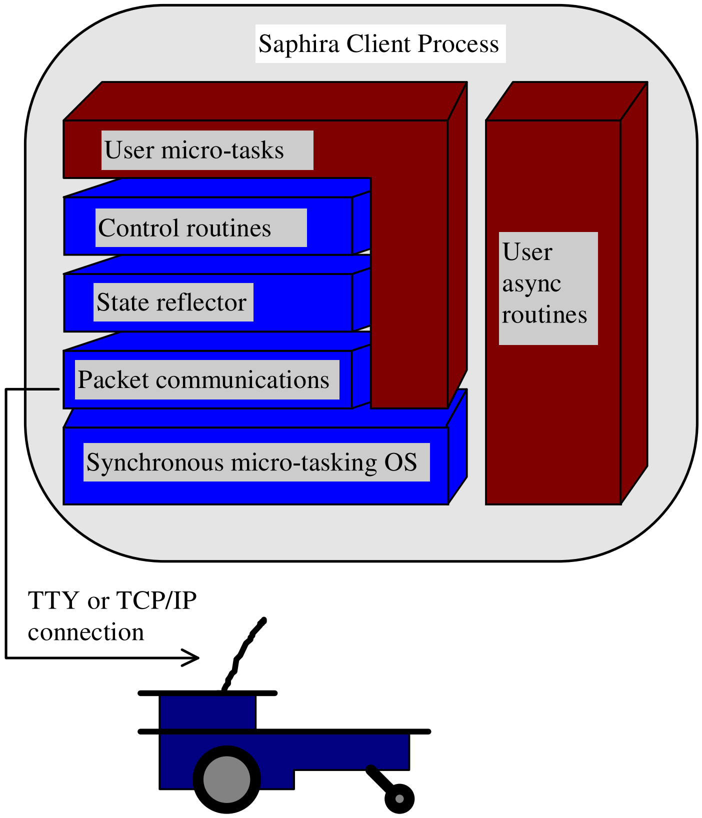

Figure 2-1 Saphira System Architecture.

Blue areas represent routines in the Saphira library, red routines are from the user. The left-hand side are all routines that get executed synchronously every 100 ms. Additional user routines may also execute asynchronously as separate threads, and share the same address space.

2.1.1. Micro-tasking OS

The Saphira architectures are built on top of a synchronous, interrupt-driven OS. Micro-tasks are finite-state machines (FSMs) that are registered with the OS. Each 100 ms, the OS cycles through all registered FSMs, and performs one step in each of them. Because these steps are performed at fixed time intervals, all the FSMs operate synchronously, that is, they can depend on the state of the whole system being updated and stable before they are called: it's not necessary to worry about state values changing while the FSM is executing. FSMs can also take advantage of the fixed cycle time to provide precise timing delays, which are often useful in robot control. Because of the 100 ms cycle, the architecture supports reactive control of the robot in response to rapidly changing environmental conditions.

The micro-tasking OS involves some limitations: each micro-task must accomplish its job within a small amount of time, and relinquish control to the micro-task OS. But with the computational capability of today's computers, where a 100 MHz Pentium processor is an average microprocessor, even complicated processing such as the probability calculations for sonar processing can be done in milliseconds.

The use of a micro-tasking OS also helps to distribute the problem of controlling the robot over many small, incremental routines. It is often easier to design and debug a complex robot control system by implementing small tasks, debugging them, and them combining them to achieve greater competence.

2.1.2. User Routines

User routines are of two kinds. The first kind is a micro-task, like the Saphira library routines, that runs synchronously every Saphira cycle. In effect, the user micro-task is an extension of the library routines, and can access the system architecture at any level. Typically the lowest level that user routines will work at is with the state reflector, which is an abstract view of the robot internal state.

Saphira and user micro-tasks are written in the C language, and all operate within the same executing thread, so they share variables and data structures. User micro-tasks have full access to all the information typically used by Saphira routines.

Although user micro-tasks can be coded directly as FSMs in the C language, it's often more convenient to use the *Activity Language* to describe FSMs. The activity language has a rich set of control concepts, and a user-friendly syntax, that makes writing control programs much easier. A translator converts activity language programs into FSMs.

Since they are invoked every 100 ms, micro-tasks must partition their work into small segments that can comfortably operate within this limit, e.g., checking some part of the robot state and issuing a motor command. For more complicated tasks such as planning, more time may be required, and this is where the second kind of user routine is important. Asynchronous routines are separate threads of execution that share a common address space with the Saphira library routines, but are independent of the 100 ms Saphira cycle. The user may start as many of these separate execution threads as desired, subject to limitations of the host operating system. The Saphira system has priority over any user threads; thus, time-consuming operations like planning can coexist with the Saphira system architecture, without affecting the real-time nature of robot control.

Finally, because all Saphira routines are in a library, user programs that link to these routines only need to include those routines they will actually use. So a Saphira client executable can be a compact program, even though the Saphira library itself contains facilities for many different kinds of robot programs.

2.1.3. Packet Communications

Saphira supports a packet-based communications protocol for sending commands to the robot server and receiving information back from the robot. Typical clients will send an average of 1 to 4 commands a second, and all clients receive 10 packets a second back from the robot. These information packets contain sensor readings and motor movement information (see Section 5.3). The amount of data sent is typically only 30 to 50 bytes per packet, so even a relatively modest 9600 baud channel can accommodate it. Saphira has the capability of connecting to a robot server over a tty line, the ethernet with TCP/IP, or a local IPC link.

Since the data channel may be unreliable (e.g., a radio modem), packets have a checksum to determine if the packet is corrupted. If so, the packet is discarded, which avoids the overhead of sending acknowledgment packets, and assures that the system will receive new packets in a timely manner. But the packet communication routines must be sensitive to lost information, and have several methods for assuring that commands and information are eventually received, even in noisy environments. If a significant percentage of packets are lost, then Saphira's performance will degrade.

2.1.4. State Reflector

It is tedious for robot control programs to deal with the issues of packet communication. So, Saphira incorporates an internal *state reflector* to mirror the robot's state on the host computer. Essentially, the state reflector is an abstract view of the actual robot's internal state. There is information about the robot's movement and sensors, all conveniently packaged into data structures available to any micro-task or asynchronous user routine. Similarly, to control the robot, a routine just sets the appropriate control variable in the state reflector, and the communication routines will send the appropriate command to the robot.

2.2. Saphira Control Architecture

The Saphira control architecture is built on top of the state reflector (Figure 2-2). It consists of a set of micro-tasks that implement all of the functions required for mobile robot navigation in an office environment. A typical client will use some subset of this functionality.

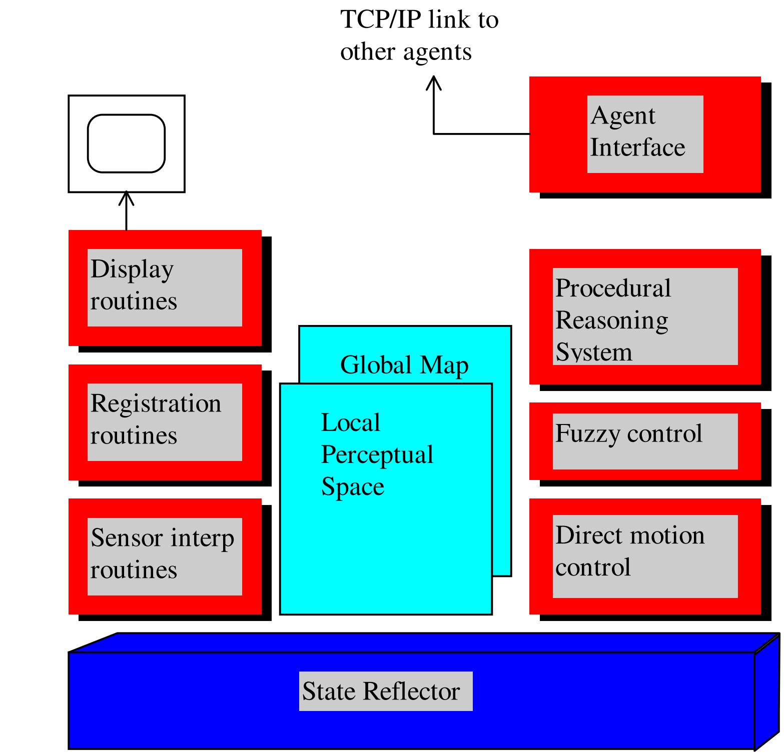

Figure 2-2 Saphira Control Architecture.

The control architecture is a set of routines that interpret sensor readings relative to a geometric world model, and a set of action routines that map robot states to control actions. Registration routines link the robot's local sensor readings to its map of the world, and the Procedural Reasoning System sequences actions to achieve specific goals. The agent interface links the robot to other agents in the Open Agent Architecture.

2.2.1. Representation of Space

Mobile robots operate in a geometric space, and the representation of that space is critical to their performance. There are two main geometrical representations in Saphira. The Local Perceptual Space (LPS) is an egocentric coordinate system a few meters in radius centered on the robot. For a larger perspective, Saphira uses a Global Map Space (GMS) to represent objects that are part of the robot's environment, in absolute (global) coordinates.

The LPS is useful for keeping track of the robot's motion over short space-time intervals, fusing sensor readings, and for registering obstacles to be avoided. The LPS gives the robot a sense of its local surroundings. The main Saphira interface window (Figure 2-2) displays the robot's LPS. In *local* mode (from the Display menu), the robot stays centered in the window, pointing up, and the world revolves around it. Keeping the robot fixed in position makes it easy to describe strategies for avoiding obstacles, going to goal positions, and so on.

Structures in the GMS are called *artifacts*, and represent objects in the environment or internal structures, such as paths. A collection of objects, such as corridors, doors, and rooms, can be grouped together into a *map* and saved for later use. The GMS is not displayed as a separate structure, but its artifacts appear in the LPS display window.

2.2.2. Direct Motion Control

The simplest method of controlling the robot is to modify the robot *motion setpoints* in the state reflector. A motion setpoint is a value for a control variable that the motion controller on the robot will try to achieve. For example, one of the motion setpoints is forward velocity. Setting this in the state reflector will cause the communications routines to reflect its value to the robot, whose onboard controllers will then try to keep the robot going at the required velocity.

There are two direct motion channels, for rotation and translation of the robot. Any combination of velocity or position setpoints may be used for these channels (see Section 6.4).

2.2.3. Behaviors and Fuzzy Control

For more complicated motion control, Saphira provides a facility for implementing *behaviors* as sets of fuzzy control rules. Behaviors have a priority, activity level, and other well-defined state variables that mediate their interaction with other behaviors and with their invoking routines. For example, a routine can check whether a behavior has achieved its goal or not by checking the appropriate behavior state variable.

2.2.4. Activities and PRSlite

To manage complex goal-seeking activities, Saphira provides a method of scheduling actions of the robot using the Procedural Reasoning System (PRSlite). With PRSlite, you can build libraries of action routines that sequence actions of the robot in response to environmental conditions. For example, a typical action routine might move the robot down a corridor while avoiding obstacles and checking for blockages.

Activity schemas are the basic building block of PRSlite. Each schema is a micro-task with enhancements for spawning child schemas and actions and keeping track of their states. Activity schemas are written using the *Activity Language*. The activity language has a rich set of control concepts, and a user-friendly syntax, that makes writing activities much easier.

Activity schemas can control the robot by invoking either direct motion commands or fuzzy control behaviors, or sequences of the two.

2.2.5. Sensor Interpretation Routines

Sensor interpretation routines are processes that extract data from sensors or the LPS, and return information to the LPS. Saphira activates interpretative processes in response to different tasks. Obstacle detection, surface reconstruction, and object recognition are some of the routines that currently exist, all working with reflected data from the sonars and from motion sensing.

2.2.6. Registration and Maps

In the global map space, Saphira maintains a set of internal data structures (*artifacts*) that represent the office environment. Artifacts include corridors, door, walls, and rooms. These maps can be created either by direct input from a map file, or by running the robot in the environment and letting Saphira extract the relevant information.

Registration is the process of keeping the robot's global location in an internal map consistent with sensor readings from the local environment. Routines exist for extracting relevant information from the LPS and matching it to map structures in the GMS, then updating the robot's position.

2.2.7. Graphics Display

Displaying internal information of the client is essential for debugging robot control programs. Saphira provides a set of graphics routines that can be called by micro-tasks. A set of pre-defined micro-tasks display information about the state reflector and other data structures, such as the artifacts of the GMS. User programs may also invoke the graphics routines directly to display relevant information.

2.2.8. Agent Interface

A Saphira client can communicate with other Internet-based agents through its agent interface to the Open Agent Architecture (OAA). The OAA is an agent-based architecture for distributed information gathering and control, and has extensive facilities for user interaction, such as speech and pen-based agents. Currently the OAA interface is under development at SRI, and issues concerning its use in Saphira outside SRI have to be resolved before it can be released.

2.3. Running the Sample Client

This section exercises some of the capabilities of Saphira through a sample client. It also illustrates the graphical user interface for interacting with clients.

To run the sample application, execute the file **`saphira(.exe)`** in the Saphira **`bin`** distribution directory. This executable should not need any local library files to run, although UNIX and Linux systems need Motif libraries for you to develop your own clients.[4]

The Saphira client will initialize an interface window showing the LPS (Figure 2-3). The robot is in the center of the display, pointing up. An information area appears at the bottom of the window, and the menu bar is at the top.

2.3.1. Connecting to a Robot

As we mentioned earlier, connecting Saphira with either the simulator or the actual robot is similar. First, if you are using the simulator, make sure that the correct robot parameters are loaded (see the *Simulator* Chapter 3 below). Otherwise, the Saphira client auto-detects the robot server type and loads its parameters when first connected (see Chapter 4 for details), so it won't be necessary to load a parameter file unless you're using a custom configuration.

If you are connecting to a robot server over the network, the Saphira client will open a dialog and ask for the hostname or IP address of the server. For Linux/UNIX systems, you may preset a default with an **`export`** or **`setenv`** shell command. For example:

```
export SERVER_NAME = servername
```

Now pull down the **`Connect`** menu and select the appropriate connection port: *simulator*, for the simulator (or a local server), a serial communications port to which you have run a tether cable or optional serial modem, or TCP for a network-based or radio-Ethernet connection. Saphira initiates the client/server synchronization sequence and displays a message when it has connected with the server.

If there is a problem connecting with either the simulator or robot server, the communication connection will fail, and a message describing the problem will appear in Saphira's main window information area. Some typical causes for failure with either the simulator or the actual robot and their solutions include:

- ✓ Make sure the physical robot's Saphira-compatible server software is properly installed and running and that no other Saphira client is connected to it.
- ✓ Make sure the simulator is running and there is no other Saphira client or simulator server running on the same machine.
- ✓ In rare cases, the communications pipe may be blocked, and can only be cleared by rebooting the machine. This can occur if either the server or client exits abnormally from a previous connection, without shutting it down properly.
- ✓ Make sure that the communications tether or radio modem is plugged into the correct serial port with the correct cable.
- ✓ Remove the serial tether cable from the robot's serial port if you use the radio modem.

---

[4] Some early versions of UNIX and Linux Pioneer and Saphira software must dynamically link with Motif GUI libraries to operate.

- ✓ Make sure the client radio modem is within range of robot, is on the correct channel, and has a strong link signal.
- ✓ Make sure the serial port is not in use by another application.

Once connected, the Saphira client will display information about the state of the robot, and allow you to command the robot from the menu and keyboard.

2.3.2. LPS Display

Once connected, the Saphira client's **`main`** display contains most of the items likely to be found in the robot's LPS (**Error! Reference source not found.**). It is a bird's-eye view of the environment around the robot. The LPS may be switched between a robot-centric display and global coordinates, using the **`local`** item in the Display menu.

The main Saphira window components include:

2.3.2.1. Robot icon

The robot icon in the center of the screen shows the robot in relation to its environment. If in local view, the LPS appears in robot-centric coordinates: the robot remains at the center of the screen and the environment moves around it. In GMS (global) mode (**`local`** mode off), the environment becomes fixed and the robot icon wanders around the screen. The size of the robot icon is controlled by the *RobotRadius* and *RobotDiagonal* values in the robot's parameter file.

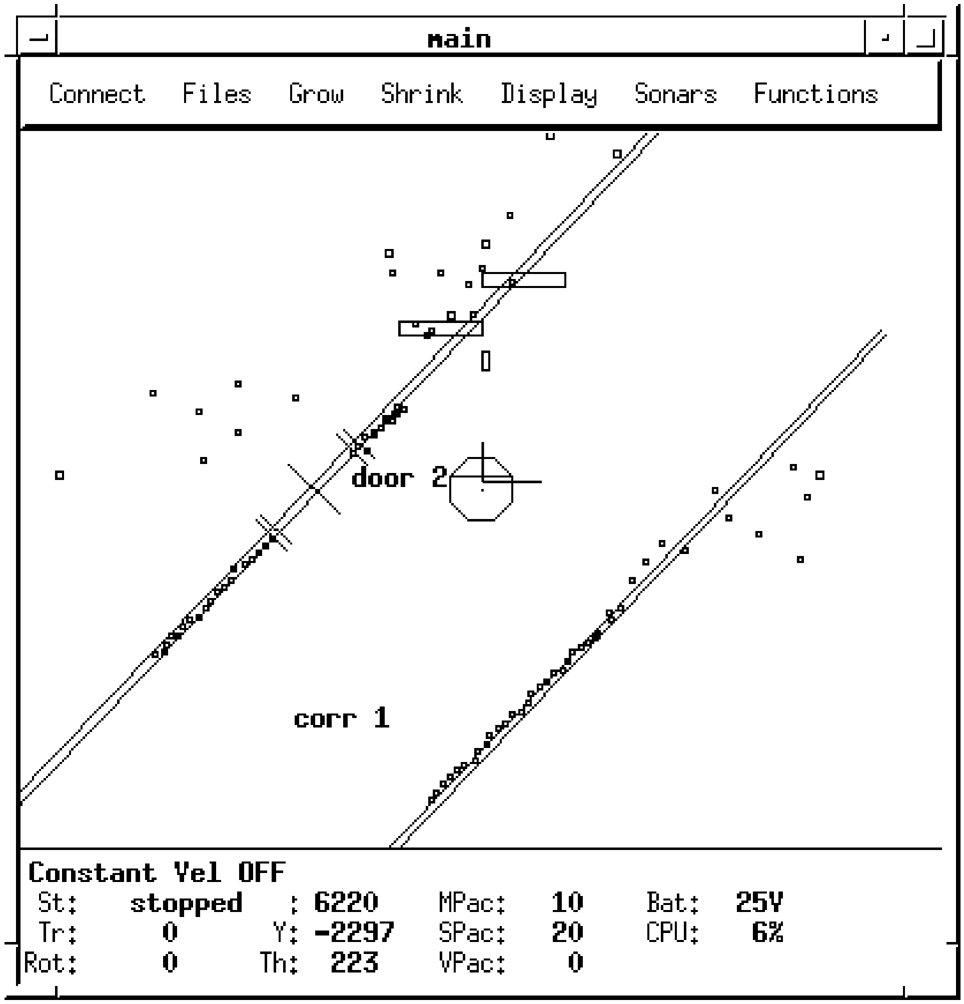

Figure 2-3 Saphira client LPS in local mode.

The corridor and door artifacts are the robot's internal map. Small rectangles are sonar readings. The larger rectangles are sensitivity areas used by the obstacle avoidance behaviors. The lines drawn at the center of the robot are angular and forward velocity. The small rectangle immediately in front of the robot is the angular setpoint.

2.3.2.2. Sonar readings

Accumulated sonar readings appear onscreen as small open rectangles. Current sonar readings are slightly larger open rectangles. Clear accumulated sonar readings from the screen by accessing the Sonars menu.

2.3.2.3. Control point

The elongated open rectangle directly in front of the robot icon is its heading control point, as returned by the server in robot-centric coordinates. Normally this control point is positioned directly ahead of the robot, veering to one side or the other in response to a turn directive from the client. The robot adjusts its heading accordingly, trying to keep heading towards the control point.

2.3.2.4. Velocity vectors

Two lines emanating from the center of the robot icon indicate the translational and rotational velocity of the robot, as returned from the robot server. The length of each vector is directly proportional to the velocity. Also, each vector points in the respective direction of motion. For example, when the robot is turning clockwise, as in Figure 6-3, the rotational vector points to the right.

2.3.2.5. Obstacle sensitivity areas

Several obstacle-avoidance behaviors temporarily draw large, open rectangles in the LPS indicating detected obstacles that they are actively avoiding. Obstacle-avoidance rectangles appear just ahead and to the sides of the robot in robot-centric coordinates. In the global view, these rectangles do not appear in the proper place near the robot icon.

2.3.2.6. Artifacts

Artifacts are internal representations of external objects or imaginary constructions, such as goal positions. Figure 6-3 shows a corridor artifact (long double lines) and a doorway labeled "**`door 2`**."

2.3.3. Information Area

The information area is at the bottom of the main window. It contains four columns of data returned from the robot server, and a message area.

2.3.3.1. Message area

At the top of the information area, Saphira posts messages about its functions. A Saphira API function, *sfMessage*, places text here.

2.3.3.2. Status (St)

Shows the robot server status as **`moving`**, **`stopped`**, or **`no servo`** when the motors are stuck.

2.3.3.3. Velocity (Tr, Rot)

The robots translational (Tr) velocity in millimeters per second and rotational (Rot) velocity in degrees per second.

2.3.3.4. Position (X, Y, Th)

Absolute robot position in millimeters and degrees. Note that this is *not* the server dead-reckoned position, which rolls over at three meters and has accumulated errors. Instead, it is the registered global position of the robot based on Saphira's map registration routines operating in conjunction with position integration returned from the server.

2.3.3.5. Communication (MPac, SPac, VPac)

The communication values in the information area are the number of packets of the given type received in the last second. They are useful for checking the communication link with the server. Normally, a client will receive ten motor packets (**`Mpac`**) and approximately 25 sonar packets (**`SPac`**) per second. Vision packets (**`Vpac`**) currently are not supported.

2.3.3.6. Miscellaneous (Bat, CPU)

The battery (**`Bat`**) voltage level on the server indicates when the robot needs to be recharged. The **`CPU`** utilization is the percentage of total processing time used by the client. On UNIX machines, this does *not* include CPU time used by the X server, which can be an appreciable fraction of total CPU time.

2.3.4. Menus

There are seven pulldown menus in the main client window.[5] These let you control the display of information in the LPS and related subwindows, manage communication to the server, and load and save parameter and map files:

---

[5] Not all menus are implemented for all versions.

2.3.4.1. Connect Menu

The **Connect** menu lets you make and break a connection to the robot server. There are five **Connect** items: the simulator, two serial ports, a local pipe, and a TCP connection. Choosing one of these items causes the client to try to connect to the physical robot or to the simulator.

The **Disconnect** item closes an open connection to the robot.

Exit causes the client program to terminate, closing any open connection first.

2.3.4.2. Files Menu

Load the robot's parameters and map files by selecting the appropriate item from the **Files** menu. A file-selection dialog appears for choosing the file. Loading a new map does not delete any old map artifacts; use the **Delete Map** item for this.

The current map can be saved to a file using the **Save Map** item, which invokes a file-save dialog.

All artifacts in the current map can be deleted with the **Delete Map** item.

2.3.4.3. Grow and Shrink

Clicking either the **Grow** or **Shrink** menu causes the LPS display to grow or shrink in scale, respectively.

2.3.4.4. Display Menu

The first item in the **Display** menu is another pulldown menu controlling the display update rate. On some systems, high update rates consume significant portions of available CPU time, and lowering the update rate will increase performance.

The **Local** item controls the LPS viewpoint. When on, the view is robot-centric; when off, the view is world-centric (global). Note that this only controls the display of information; all internal geometric structures remain the same.

Single Step mode is useful for debugging and can only be used with the simulator. When on, it causes the simulator to wait for a signal from the client at each 100 millisecond time step. Pressing the spacebar in the client window signals the next time step.

The **Wake** item, if on, deposits "breadcrumbs" in the display, showing the last ten seconds of robot travel.

The **Occ Grid** item, if on, displays the occupancy grid constructed using the MURIEL algorithm. This item is not implemented on Macintosh or machines with no color capability. On some machines, it may consume a large percentage of available CPU time for display.

2.3.4.5. Sonars Menu

The **Clear Buffer** item clears all of the accumulated sonar readings from the client internal buffers.

The **Sonars On** item toggles the sonar capability of the robot server. (Currently not implemented on the robot server; the sonars are always on.)

2.3.4.6. Functions Menu

The Functions menu toggles the display of the Behaviors, Processes, and Intentions windows.

2.3.5. Keyboard Actions

In addition to the Saphira pulldown menus, you may control some of the functions of the robot server directly from the client keyboard (Table 2-1).

Table 2-1. Keyboard-enabled Saphira drive and behaviors*

KEY	ACTION
i, ↑	Increment forward velocity
m, ↓	Decrement forward velocity
j, ←	Incremental left turn
l, →	Incremental right turn
k, space	All stop
g	Constant Velocity on/OFF
c	Obstacle Avoidance ON/off

* These keys work *only* when the main Saphira window is active.

The sample Saphira client we provide defines a set of keyboard actions for robot motion, and for turning some behaviors on and off. In a user application, the function *sfProcessKey* lets you intercept keystrokes and initiate your own "hot-key" actions.

2.3.6. Behaviors Window

Saphira's **`Behaviors`** window shows graphically the state of all current behaviors. It is invoked from the **`Functions`** menu in the main window. To understand the contents of this window, it may be useful to review the previous section in this chapter on Saphira behaviors.

Our sample Saphira client invokes four behaviors: two for obstacle avoidance, one for going forward at a constant velocity, and one for stopping. The obstacle avoidance behaviors are called *Avoid Collision* and *Keep Off*. Avoid Collision prevents the robot from banging into obstacles at close range by initiating a sharp turn and slowing down the robot. The Keep Off behavior deflects the robot from longer-range obstacles.

The Constant Velocity behavior attempts to keep the robot going forward at a fixed speed of about 300 millimeters per second.

The Stop behavior, not surprisingly, stops the robot. It is useful when you want the robot to stop if no other behavior is managing the robot's movements. For example, if the Constant Velocity behavior is invoked and then killed, the robot will still have a residual forward velocity. In the absence of any other behaviors, it will keep moving forward. Invoking the Stop behavior at a low priority assures that the robot will stop if it is not doing anything else.

Figure 2-4 shows a typical **`Behaviors`** window. The first two behaviors in our sample client are *active*, that is, they can contribute to the control of the robot (their *running* parameter is 1). The other two are inactive. The active state of a behavior may be toggled by clicking the radio button to the left of the behavior name with the mouse.

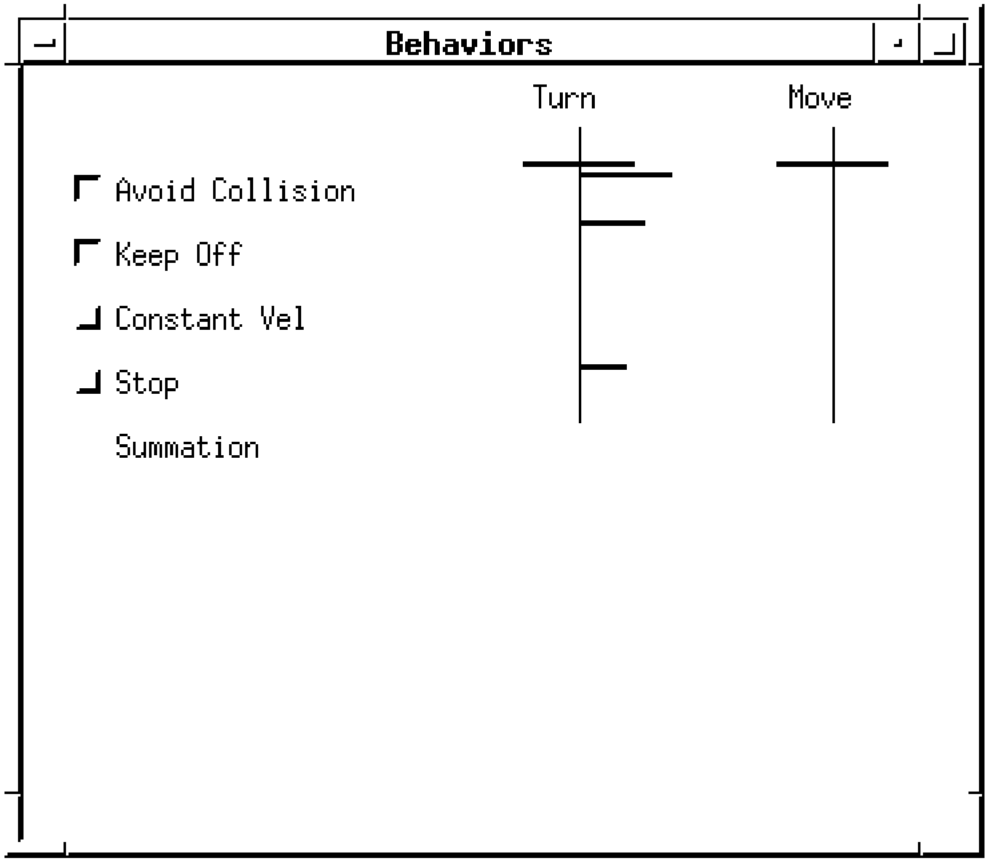

Figure 2-4 Saphira Behaviors window (Linux/Motif version)

The dark bar next to each behavior name indicates the state of the behavior. There are two vertical lines, representing the behavior's outputs for turning and forward/backward movement. For example, the **`Keep Off`** behavior in Figure 2-4 is fully active for both turning and moving, as indicated by the horizontal activity bars going through the vertical lines (details in Figure 2-5). This behavior instructs the robot to turn right and to move backwards (slow down) in this example, as indicated by the direction bars on either side of the vertical lines.

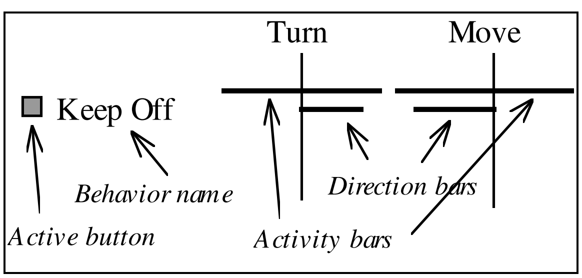

Figure 2-5 Keep Off behavior display expanded

The behaviors appear in order of their priority in influencing the robot's actions, with the highest priority behaviors at the top of the window. At the bottom, the *Summation* line gives the end result of combining the active behaviors according to their priority. It is the summation that ultimately controls the robot server's actions.

It's often useful to view an individual behavior's activity in more detail. Individual behavior windows can be opened by shift-clicking on the behavior name (UNIX systems) or left-clicking just to the right of the name (MS Windows). Figure 2-6 shows a typical behavior window while active. The invocation

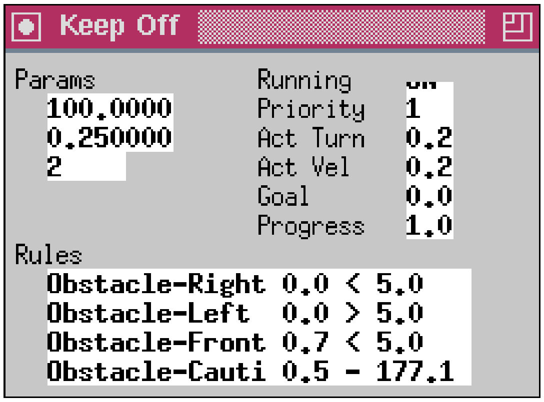

Figure 2-6 Behavior window for *Keep-Off*

parameters of the window are in the upper left; pointer parameters have their addresses printed. The right-hand side of the window shows the state variables of the behavior: whether it's active or not, activity levels, and so on. Finally, at the bottom of the window, the rules are printed, showing their antecedent values and control sets.

The format of the rules is: **Name Antecedent Direction Value.**

The antecedent value determines how strongly the rule applies. The direction is a single character: ">" or "<" for right or left turn, "+" or "-" for speed up or slow down. The value indicates the desired control signal; a left turn of 5.0 degrees, for example.

2.3.7. Processes Window

The **`Processes`** window displays the states of all micro-tasks in the Saphira client multitasking queue (Figure 2-7). Open it from the **`Functions`** menu in the main window. The **`Processes`** window contains a scrolled list, where each entry consists of the micro-task name and state. The display is updated in real time as the micro-task state changes.

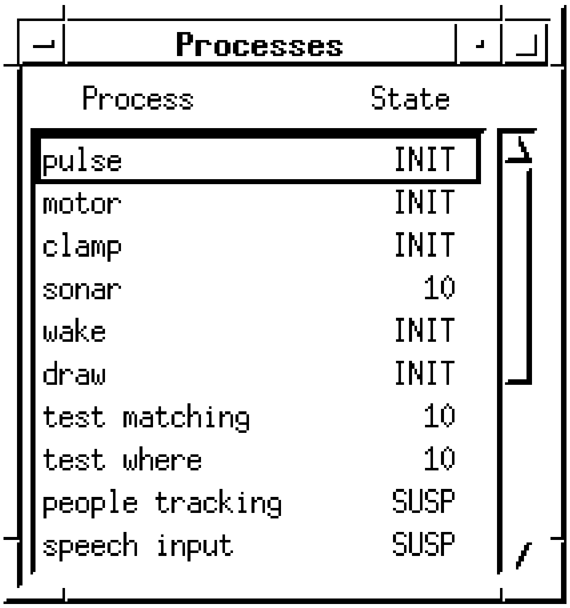

Figure 2-7 A sample Saphira Processes window

You may interrupt a running micro-task by selecting it in the window and pressing the Enter key, or by double-clicking with the mouse. This action forces the micro-task state to *INT* (interrupt). Resume an interrupted micro-task with the same action, which forces the micro-task state to *RES* (resume).

An interrupted micro-task does not automatically suspended processing; that depends on how the micro-task handles the interrupt state. Some micro-task ignore the interrupt and continue with their tasks. For example, the motor micro-task does not care what its state is—it always performs the same action of sending motor commands to the robot server. In general, you should only interrupt micro-tasks that you have added to the Saphira application, and for which there is a defined interrupt behavior.

2.3.8. Intentions Window

Saphira's **`Intentions`** window shows the state and relationship of all current PRS intentions (Figure 2-8). Open it from the **`Functions`** menu in the **`main`** window.

The **`Intentions`** window contains a scrolled list similar to the **`Processes`** window, and each line contains the intention name and its state. The state information is updated in real time as the intention state changes.

Relationships between intentions are indicated by line indentations. For instance, in the example Figure 2-8, the second intention "follow it" is indented to show that it is a child of the first intention. The two intentions combine to invoke a corridor-following sequence for the robot. The top-level intention waits until the robot has found a corridor, then invokes its child intention to select a path to follow down the center of the corridor. In addition, the top-level intention monitors the state of the robot, and when it is no longer in the corridor, or gets turned sideways to the corridor, it disables the follow it intention.

As with micro-tasks, you may manually interrupt an intention by selecting it and pressing the Enter key, or by double-clicking it with the mouse. If the intention is running, this will force it into the *INT* (interrupt) state. Normally, an intention will respond to this state by suspending. Use the same action to reactivate an interrupted/suspended intention. This will invoke the Saphira *RES* (resume) state. Normally, an intention will respond to this state by reinitializing and starting its characteristic behaviors.

There is one intention in the sample Saphira client that has the robot find and follow corridors. The intention monitors the robot environment until it detects a corridor, then starts a sub-behavior that projects a path for the robot down the middle of the corridor.

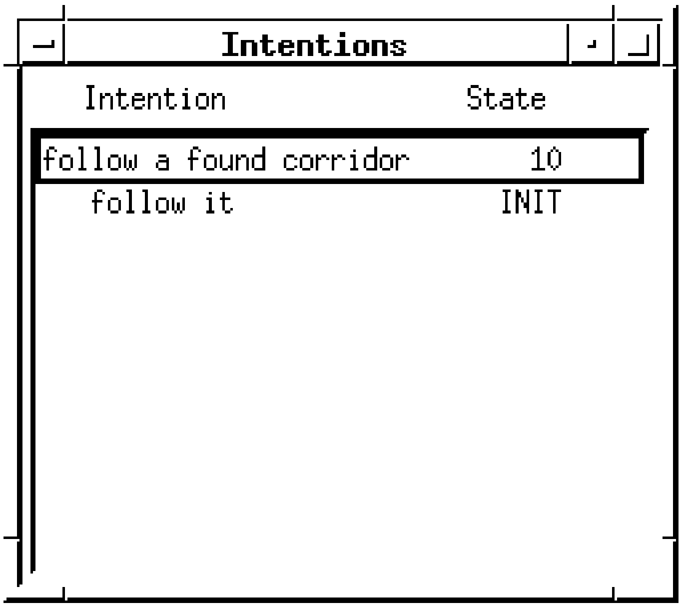

Figure 2-8 An example Saphira Intentions window

3. *The Simulator*

The simulator is a very useful alternative to a physical robot for developing robotics programs. Although there is nothing like real world conditions to humble the most ambitious robotics project, the simulator does have the distinct advantage of having a single-step mode in which you can reenact every detail of your programs, including a robotics fatality.

And, too, the simulator has realistic error models for the sonar sensors and wheel encoders so that, in general, if a client program works with the simulator, it will work on the physical robot. The simulator also lets you construct a simple world in which the simulated robot navigates. You can even change the robot's operating characteristics to simulate your own robot designs. And since the packet interface of the simulator is the same as the physical robot, no changes to the client program are required in switching between the two.

The disadvantage of the simulator is that the environment model is an abstraction of the real world, with simple 2-D linear segments in place of the complex geometrical objects the real robot will encounter in the real world. For example, the simulator assumes all objects are sensor-high, so it can't simulate a door stop—something the real robot will have to overcome to traverse rooms in a real building.

3.1. *Starting the Simulator*

Execute the program named **`pioneer(.exe)`** in the Saphira **`bin/`** directory. (By default, the simulator acts like the Pioneer 1 Mobile Robot—hence, its name. We tell you how to simulate other robots in a following section of this chapter.) Normally, connect and disconnect to the simulator from the client using an interprocess communications channel on the same machine. If for some reason the client terminates abnormally, the simulator can be disconnected using the Disconnect option from the **`Quit`** menu. Disconnecting or quitting the simulator while the client is connected will cause the client to quit.

Once connected with a client, the simulator displays a window of its activity. A sample window is shown in Figure 3-1. The simulated robot is the circular icon in the center of the screen; the straight lines are simulated world segments: walls, corridors, rooms, and so on. A collection of segments—a *world*—may be defined in a simple text file (see below) and loaded from the simulator's **`Load (Files)`** menu.

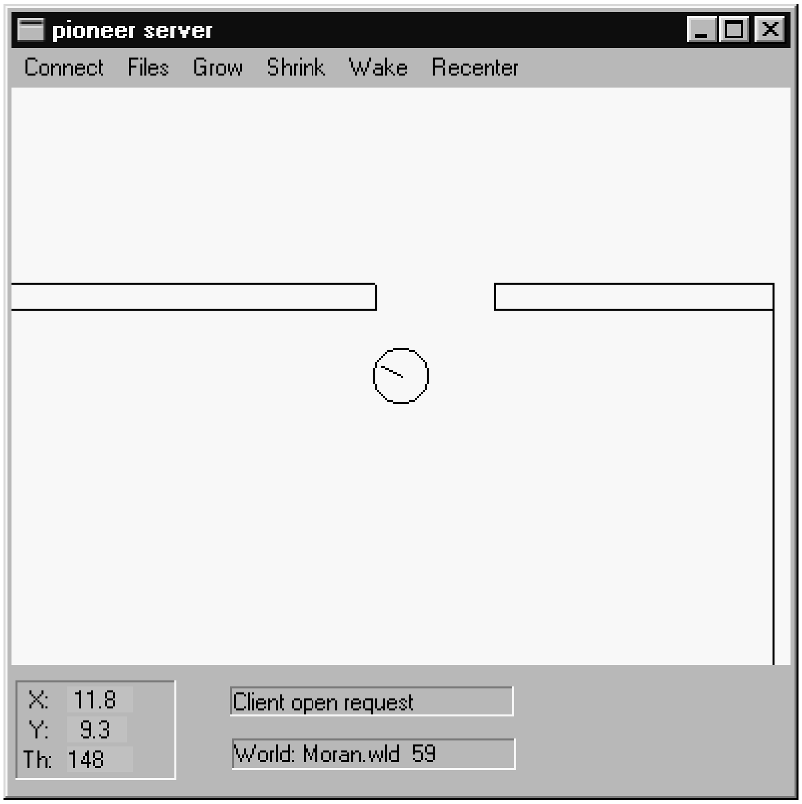

Figure 3-1 A sample window of the simulator

The simulator listens on an interprocess communication channel for connections from a server. In Unix systems, this is a local Unix socket; under Windows, it is a mailbox. Default names for these sockets are supplied by the simulator. Only one simulator may be connected at a time to that socket or mailbox. In some cases, it is convenient to start up multiple copies of the simulator; or, for some reason, the socket may be busy or unavailable. In these cases, the simulator can be started with an alternative socket name. Set the environment variable **`ROBOT_SOCKET`** to the name of the desired socket before starting the simulator, and it will be used instead of the default. The simulator window shows which socket it's listening on.

To connect to a particular socket form the client side, set the **`ROBOT_SOCKET`** environment variable to the name of the desired simulator socket before trying to connect.

3.2. Parameter File

The default operating parameters for the simulator are for the Pioneer 1. You may reset these working parameters to simulate nearly any mobile robot by constructing then loading a special robot parameter file into the simulator from the **`Load (Files)`** menu. Find a variety of prepared parameter files in the Saphira **`params/`** directory. The newly loaded model is active for as long as you run the simulator or until you load another parameter file.

You prescribe a variety of simulated robot characteristics in a parameter file, such as placement of sonars and drive error tolerances. Once constructed, normally store your parameter file in common text (ASCII) format in the **`params/`** directory, usually with a "**`.p`**" suffix to the filename. A sample, annotated parameter file listing is in the Appendix A, and the parameter file can be found in the Saphira collection as **`params/pioneer.p`**.

Three important parameters control the amount of error in the simulated robot's motion (Table 3-1). Consult the listing in Section 10 for more details.

Table 3-1. Example drive error tolerance values for a parameters file

Parameter	Pioneer Value	Description
EncodeJitter	0.01	Error in distance
AngleJitter	0.02	Error in angular position
AngleDrift	0.003	Angular drift with forward movement

3.3. World Description File

A world description file is a plain text (ASCII) document typically stored with the "**`.wld`**" filename suffix, which describes the size and contents of a simulated world. A sample world file can be found in the Section 11, along with instructions on how to create your own worlds. We've also included several sample world files with the Saphira distribution found in the **`worlds/`** directory.

3.4. Simulator Menus

Several simulator menus control the parameters and actions of the simulated robot. There are controls for loading world and parameter files, for adjusting the display, and for changing the connection type, for example. (Not all menus are implemented in every version.)

3.4.1. Load (Files) Menu

The **`Load Param File`** item brings up a file selection dialog to load a robot parameter file. The parameter file changes the characteristics of the simulated robot, such as the number and placement of the sonars. By default, the Pioneer robot parameters are loaded.

The **`Load World File`** item brings up a file selection dialog to load a world file.

3.4.2. Connect Menu

The **`Connect`** menu controls the port that the simulator listens on, and also disconnects the simulator from an aborted client.

By default, the simulator is listening on the interprocess communication port, waiting for a client on the same machine. The simulator can also listen on one of the serial ports, if the appropriate port name is selected from the menu. In this case, the simulator and client can run on different machines.

The **`Disconnect`** item causes an immediate disconnect of the simulator from its connected client. Normally, the simulator will disconnect automatically when the client sends it the *sfCLOSE* command.

In situations where the client has a system error and exits abnormally, the client may remain connected, even though the connection is no longer valid. In this case, the Disconnect item will force the connection to close, so the simulator can go back to a listening state.

With the Windows95/NT version, an Exit option also is in the **`Connect`** menu

3.4.3. Display Menu (Grow, Shrink and Wake)

The **`Grow`** and **`Shrink`** menus or items in the simulator's **`Display`** menu change the size of the display.

The Wake item, if on, or **`Wake`** menu, when clicked, causes a the simulator to display a breadcrumb of the last few seconds of simulated robot travel.

3.4.4. Recenter Menu

Selecting the **`Recenter`** menu item centers the display around the current robot position. It does not change the robot's position.

Normally, the simulator will keep the robot icon near the center of the display by moving the display window when the robot approaches an edge.

3.4.5. Exit Menu

The **`Exit`** menu (or item in **`Connect`** menu) terminates the simulator. A connected simulator should be disconnected first from the client side, or it will cause the client to abort.

Exiting shuts down any current connection and exits the application. Quitting a connected simulator will usually cause the client to quit as well, so it's a good idea to disconnect from the client side first.

3.4.6. Information Area

The information area at the bottom of the simulator window shows messages about the connection status. It also shows the absolute x,y position of the robot in meters, and the angle of the robot in degrees.

3.5. Mouse Actions

The left mouse button puts the simulated robot at the position of the cursor. This moves the robot in its world, and the X, Y coordinates at the bottom of the screen will change. If the robot becomes stuck against a wall, using the left mouse button to move it a little can unstick it.

The middle button moves the simulated world position at the cursor to the center of the display.

4. Creating Saphira Clients

This chapter describes how to compile and link a Saphira client. The next chapter contains details of the Saphira API, which should be used as a reference guide to the Saphira libraries.

In addition to the Saphira API, the best reference material is the example clients that are defined in the Saphira distribution and in the tutorial documentation at the SRI Saphira website (*http://www.ai.sri.com/~konolige/saphira*). The sample clients are found in the **`handler/src/apps`** directory, and they are explained in more detail below. Other examples of behavior definition are in the **`handler/src/samples`** directory: see the documentation on fuzzy control behaviors at the website.

The functions in the Saphira library may be invoked by linking with other programs. Typically, developers write the client in C or C++ using the header files in **`handler/include`** that contain prototypes and definitions of structures and variables in the Saphira library. After compiling his or her files, the developer links them with the Saphira library to create an executable client, which can connect to the robot and control it.

It is also possible to use the Saphira system from other languages such as LISP or PROLOG, as long as they have a foreign-function interface facility. In this case, the developer writes some routines in C or C++ that are compiled into object files, and then these object files, together with the Saphira libraries, are loaded into the LISP or PROLOG system.

4.1. Host System Requirements

Saphira libraries are available for most Unix systems (including SunOS 4.1.3, Solaris 5.x, SGI Irix, Dec OSF, Linux, and FreeBSD), as well as MS Windows 95 and NT 3.51 and 4.0. For Unix systems, we recommend using the **`gcc`** compiler and linking tools from the Free Software Foundation. These tools provide a uniform base for making clients, and the sample programs are all made with them.

In addition, if you want to use any of the graphics or user interface routines, you will need the following libraries and headers:

1. X11R5 or later
2. Motif 2.0 or later

For MS Windows, the libraries have been compiled with MS Visual C 4.x tools. There is a **`DLL`** file and an associated **`LIB`** file. For the best compatibility, we recommend using MSVC 4.0 or later: all of the sample clients are given with **`.MAK`** files for MSVC 4.0. It may be possible to use Borland tools, but they have not been tested; there may be a problem with incompatibilities between MSVC and Borland **`LIB`** files.

4.2. Compiling and Linking Saphira Applications

To compile a Saphira client, you must have installed the Saphira distribution according to the directions in the **`readme`** file. In particular, the environment variable **`SAPHIRA`** must be set to the top level of the distribution: we recommend **`/usr/local/saphira/ver53`** in a UNIX system, for example.

Once the Saphira distribution is installed, there are three steps to creating a client:

1. Write a C or C++ program containing your code for the client, including calls to Saphira library functions.
2. Compile the program to produce an object file.
3. Link the object file together with the relevant Saphira library to create an executable.

In Unix systems, all the Saphira library routines are statically linked into the executable. In MS Windows, the executable does not contain the library code, but loads the Saphira **`DLL`** file when it executes.

4.2.1. Writing C or C++ Client Programs

To develop a Saphira application, you write one or more C or C++ programs that make calls to the Saphira library routines. It may help to review Section 2 for an explanation of micro-tasks and asynchronous user routines.

The main file will always have the following structure in Unix systems:

```
#include "saphira.h"     /* header file for Saphira library */

...definition of startup, connect, and disconnect callbacks...

void main(int argc, char **argv)
{
   /* register callbacks */
   sfOnConnectFn(myConnectFn);
   sfOnStartupFn(myStartupFn);

   /* start up Saphira micro-tasking OS */
   sfStartup(0);
}
```

The Saphira library headers, as well as other relevant system and graphics headers, are loaded by the **`saphira.h`** file. The callbacks are defined to start up Saphira or user micro-tasks when the client connects to or disconnects from the robot. The **`main`** function is the entry to the client; it registers the callbacks, and then starts up the Saphira micro-tasker with the call to **`sfStartup`**. The argument of 0 to this function means that control does not return to the main program: all processing is done using micro-tasks, and the client exits when the *Exit* item is chosen from the *File* menu.

Programming in MSVC is similar, except that the form of the **`main`** function changes to MS Windows programming standards.

```
#include "saphira.h"     /* header file for Saphira library */

...definition of startup, connect, and disconnect callbacks...

int PASCAL WinMain(Handle hInst, HANDLE hPrevInstance,
              LPSTR lpszCmdLine, int nCmdShow)
{
   /* register callbacks */
   sfOnConnectFn(myConnectFn);
   sfOnStartupFn(myStartupFn);

   /* start up Saphira micro-tasking OS */
   sfStartup(hInst, nCmdShow, 0);
   return 0;
}
```

In this case, control does return to the main program after the Saphira client exits, and the user should return 0 to indicate that the exit was normal.

For most robot programming, all operations can be handled in micro-tasks. If a more compute-intensive task must be done concurrently, then **sfStartup** should be called with an argument of 1, which means that the Saphira micro-tasking OS is started, and immediately returns control to the main program.

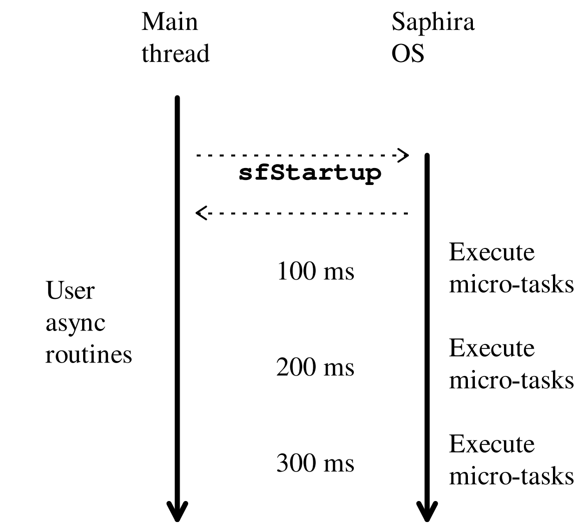

Figure 4-1. Concurrent execution of Saphira OS and user asynchronous tasks.

The user can now run any routines concurrently with the Saphira OS, which is executing its micro-tasks every 100 ms. The micro-tasks and the asynchronous user routines share the same address space, and can communicate via global variables.

Figure 4-1 is a graphical view of the execution process. The main client thread starts up, and invokes the Saphira OS with the **sfStartup** function. Once startup, the OS wakes up every 100 ms and runs every micro-task. If the argument to **sfStartup** is 0, then control never returns to the main thread. If it is 1, then control returns immediately, and both threads execute concurrently.

Explanations of some sample Saphira client programs are given later in this section.

4.2.2. Compiling and Linking Client Programs under Unix

Once the client programs are written, they must be compiled with a C or C++ compiler. We recommend the **gcc** compiler for Unix systems; all sample programs have been compiled using this compiler. Other C compilers provided with Unix systems should also work, however.

The compiler and linker are typically called using the *make* facility. Makefiles are given for all of the sample clients. Here is a portion of the makefile for **handler/src/apps/lowlev.c**:

```
#############################################################
# November 1996
#
# Makefile for Saphira applications
#
#############################################################

SRCD = ./
OBJD = ./
INCD = $(SAPHIRA)/handler/include/
LIBD = $(SAPHIRA)/handler/obj/
BIND = ./

# find out which OS we have
include $(SAPHIRA)/handler/include/os.h

CFLAGS =  -g -D$(CONFIG)
CC = gcc
INCLUDE = -I$(INCD) -I$(X11D)include


#############################################################
all: $(BIND)lowlev
        touch all

$(OBJD)lowlev.o: $(SRCD)lowlev.c $(INCD)saphira.h
        $(CC) $(CFLAGS) -c $(SRCD)lowlev.c $(INCLUDE) \
              -o $(OBJD)lowlev.o

$(BIND)lowlev: $(LIBD)libsf.a $(OBJD)lowlev.o
        $(CC) $(OBJD)lowlev.o -o $(BIND)lowlev -L$(MOTIFD)lib \       -
L$(OBJD) -L$(LIBD) -lsf $(LLIBS) -lc -lm
```

The first part of the makefile defines variables that are useful in compilation and linking. Note that the **SAPHIRA** environment variable must be defined as the top level of the Saphira distribution (with no final slash). The **handler/include** directory contains header files, and **handler/obj** has the libraries.

Next, the file **handler/include/os.h** is read in. This file determines the operating system type and sets some system library variables appropriately, for X windows and Motif. It also sets the **CONFIG** variable to the particular OS of the machine, which is important for handling some of the system routines correctly. For most OS's, the Motif (**MOTIFD**) , X11 (**X11D**) and system libraries (**LLIBS**) are set correctly, but there may be cases in which they are not. In this event, go into the **os.h** file and change the definitions under your OS.

One peculiarity of **os.h** is that it relies on the conditional preprocessing facilities of gnu **make** (**gmake**). Not all native **make**s support this facility. If you get errors during the preprocessing phase of the compilation from **os.h**, switch to **gmake**.

The compile command makes **lowlev.o** from the **lowlev.c** file. It is important that the variable **-D$(CONFIG)** is passed to the compiler, because this tells the header files what particular variant of Unix is being used. The include directories are the Saphira header directory and the X11 directory.

The link command takes the object file generated by the compile command and links it with the Saphira library and system libraries to form the executable. The Saphira library is indicated by **-lsf**. This is the library that opens a graphics window and has all the user interface functions. If you don't want a window, use the **-lsfx** library. The **LLIBS** variable indicates other system libraries that may be needed by this particular Unix system.

The executable is deposited in the same directory as the source file, and can be invoked by typing its name at the shell prompt.

4.2.3. Compiling and Linking Client Programs under MSVC

With Microsoft Windows, the sample Saphira clients are MS Visual C++ 4.x projects. There are **`.MAK`** files for all of the sample clients in the **`handler/src/apps`** directory; load these into MSVC and you should be able to compile and link the clients. One problem with the included projects is that they use absolute path names for the source files (including the library file **`sf.lib`** and resource file **`saphira.rc`**. At this time there seems to be no way to specify relative path names, so if you use a different distribution directory (i.e., *not* **`c:\saphira\ver53`**) then you will not be able to compile the sample applications until you add in the same files using the **`add files`** command.

To run the clients, make sure that the **`SF.DLL`** file is accessible in the **`C:\Windows\System`** directory, or in a directory on your **`PATH`** variable.

The easiest way to compile and link your own clients is to use the sample project files, and modify them to include your source files instead of the sample clients. Here are some things to remember when creating new MSVC projects.

1. The Saphira library file **`handler/obj/sf.lib`** must be included in the project files.
2. The Saphira resource file **`handler/saphira.rc`** must be included in the project files.
3. The project must compiled in 32 bit mode, not 16 bit mode.
4. You must add the directory for the include files, **`$(SAPHIRA)\handler\include`**, into the "Additional include directories" slot in the *Build/Settings* menu under the *C/C++* tab and *Preprocessor* category. Also, make sure the symbol **`_WINDOWS`** is defined in the *Preprocessor definitions* slot here.
5. You must add the directory for the include files, **`$(SAPHIRA)\handler\include`**, into the "Additional resource include directories" slot in the *Build/Settings* menu under the *Resource* tab.

4.3. Client Examples

In this section we illustrate some of the ways of writing Saphira clients with four examples. These files are all in **`handler/src/apps`**. For explanations of the functions and data structures, see the relevant sections of the Saphira API reference.

- **`direct.c`** This client uses the state reflector and the direct motion routines to move the robot back and forth between two points. The patrol routine is a Saphira micro-task.
- **`async.c`** This client also uses the state reflector and direct motion routines, but instead of invoking a micro-task it calls the motion routines asynchronously.
- **`nowin.c`** Like the previous client, this one calls the motion commands asynchronously, but ignores the user interface routines and connects to the robot directly.
- **`saphira.c`** This is the source for the demonstration client **`bin/saphira`**. It invokes behaviors, activities, and perception micro-tasks, as well as user-interface functions on the mouse buttons.
- **`packet.c`** This client bypasses the state reflector of Saphira, providing its own packet communication handler.

4.3.1. The `Direct` Client

Using direct motion commands, the **`direct`** client moves the robot back and forth along a two meter line. This activity is done by a micro-task, the **`patrol`** function. Only part of this function is shown here.

The example starts with a header file that reads in all prototype and structure information for the Saphira libraries. The headers can be read by C or C++ programs; all library names are C names. The file **`handler/include/saphira.h`** automatically configures the C compiler for the operating system you're running on: UNIX (SGI, Solaris, Linux, FreeBSD) or MS Windows 95/NT. If you need to customize these files, for example, if you have the Motif libraries in a different place than Saphira assumes, then look in **`handler/include/os.h`** and the various configuration files **`handler/include/conf-xxx.h`** for library and header file definitions.

Saphira provides a way to call user functions whenever it is started up or connects to the robot. It does this by registering user functions as *callbacks* with **`sfOnStartupFn`** and **`sfOnConnectFn`**. Whenever a startup or connect event takes place, Saphira calls the registered user function.

The startup callback is used to initialize various features of Saphira's display, such as the display rate, or local/global mode. You can't set these before calling **`sfStartup`** because the windows aren't created yet. If you don't want to do any special processing here, there's no need to define a startup callback.

In this application, **`myStartupFn`** is invoked when the Saphira OS is initialized, and it sets the display mode to global coordinates (see the *sfSetDisplayState* function in the API reference). **`myConnectFn`** is invoked when the client connects to the robot server (using the *Connect* menu); it starts up the communication micro-tasks that maintain the state reflector, and then starts the patrol micro-task.

In the **`main`** function, the callbacks are registered, and then the Saphira OS is started by **`sfStartup`**. Since the argument is 0, this function does not return, and all computation takes place in the micro-tasks.

```
#include "saphira.h"

void patrol(void)
{
     switch(process_state) {
     case INIT:
  case 20:
    sfSetPosition(2000);
    process_state = 21;
    break;
  case 21:
    if (sfDonePosition(100))
       process_state = 22;
    break;
   …
}}

void myStartupFn(void)
{
  sfSetDisplayState(sfGLOBAL, TRUE);  /* use the global view */
}

void myConnectFn(void)
{
  sfInitBasicProcs();           /* start up comm processes */
  sfSetMaxVelocity(200);        /* robot moves at this speed */
  sfInitProcess(patrol,"patrol");
}

void main(int argc, char **argv)
{
  sfOnConnectFn(myConnectFn);/* register a conn function */
  sfOnStartupFn(myStartupFn);/* register a startup function */
  sfStartup(0);           /* start up the Saphira window */

}
```

4.3.2. The `Async` Client

This client demonstrates asynchronous control of the robot, that is, outside the micro-task loop. As in the **`direct`** client, the startup and connect callbacks are defined and then registered in the **`main`** function. Then, **`sfStartup`** is called with an argument of 1, which starts up the Saphira OS, and then keeps executing the user's program.

The program waits in a **while** loop until the user connects to a robot, then starts to issue a series of direct motion commands. The motion commands are synchronized using the **sfDoneXXX** functions to wait for completion, and **sfPause** to wait for a time interval.

Finally, it closes the connection to the robot and exits. When the main program exits, the Saphira OS is also automatically exited. If you want to keep the micro-task OS operating, then just start a **while** loop whose body is **sfPause(1000)**.

Note that the packet communication and state reflection micro-tasks are initiated in the connect callback (**myConnectFn**). It's important to do this, since the direct motion commands rely on state reflection to control the robot.

```
#include "saphira.h"

void myStartupFn(void)
{
  sfSetDisplayState(sfGLOBAL, TRUE);  /* use the global view */
}

void myConnectFn(void)
{
  sfInitBasicProcs();          /* start up comm processes */
}

void main(int argc, char **argv)
{
  int i = 0;

  sfOnConnectFn(myConnectFn);  /* register a conn function */
  sfOnStartupFn(myStartupFn);  /* register a startup function */
  sfStartup(1);                /* start up the Saphira OS,
                                  and then keep going */

  while (!sfIsConnected) sfPause(0); /* wait until connected */

  sfSetRVelocity(100);         /* in mm/sec on each wheel... */
  sfPause(4000);
  sfSetRVelocity(0);
  sfPause(4000);

  for (i=0; i<280; i+=20)
    {
      printf("Turn %d degrees\n", i);
      sfSetDHeading(i);        /* turn i degrees cc */
      while (!sfDoneHeading(10))
          sfPause(0);    /* wait till we're within 10 degrees */
      sfSetDHeading(-i); /* turn i degrees c */
      while (!sfDoneHeading(10))
          sfPause(0);    /* wait till we're within 10 degrees */
    }

  sfSetVelocity(300);          /* move forward at 300 mm/sec */

  for (i=0; i<10; i++)
    {
      printf("X: %f  Y: %f\n", sfRobot.ax, sfRobot.ay);
      sfPause(1000);     /* DON'T USE SLEEP!!!! */
      sfSetDHeading(10);
    }

  sfSetVelocity(0);      /* move forward at 300 mm/sec */
  sfPause(4000);
  sfDisconnectFromRobot();     /* we're gone... */
}
```

4.3.3. The `Nowin` Client

Like the **`async`** client, this client makes use of the asynchronous execution of user routines. But instead of starting up the Saphira interface window, it just connects to the robot by a function call, and then starts executing direct motion commands. If this client is linked with the non-window library (**`sfx`**), then no interface window will appear (in MS Windows, you can use a console application instead of GUI application).

```
#include "saphira.h"

[ omitted callback definitions ]

void main(int argc, char **argv)
  int i = 0;

  sfOnConnectFn(myConnectFn);   /* register a conn function */
  sfOnStartupFn(myStartupFn);   /* register a startup function */
  sfStartup(1);                 /* start up the Saphira OS,
                                   and then keep going */
  if (sfConnectToRobot(sfLOCALPORT, sfCOMLOCAL))
                               /* this is for the simulator */
    {
      sfSetVelocity(300);       /* move forward at 300 mm/sec */

      for (i=0; i<10; i++)
              {
                printf("X: %f  Y: %f\n", sfRobot.ax, sfRobot.ay);
                sfPause(1000);  /* DON'T USE SLEEP!!!! */
                sfSetDHeading(10);
              }

      sfSetVelocity(0);         /* stop */
      sfPause(4000);
      sfDisconnectFromRobot();  /* we're gone... */
    }
  else
    printf("Can't connect!!\n");
}
```

4.3.4. The `Saphira` Client

The example Saphira client makes use of many pre-defined behaviors and micro-tasks to implement a simple handler for the robot. There are behaviors for obstacle avoidance and forward motion at constant velocity, as well as processes for interpreting sonars, recognizing corridors, and registering the robot against previously found objects. In the connect callback, all these routines are started up using Saphira library calls.

```
/*
 * The Saphira application: all the basic processes loaded
 *  plus registration and map-making processes
 *  Behaviors: avoid-obstacle, keep-off, constant-velocity, stop
 *  Intentions: follow-it
 */
#include "saphira.h"

void myConnectFn(void);
void myStartupFn(void);
int myKeyFn(int ch);     /* any user key processing here */
int myButtonFn(int x, int y, int b);

void main(int argc, char **argv)
{
  /* set up user button and key processing */
  sfButtonProcFn(myButtonFn);
  sfKeyProcFn(myKeyFn);
  sfOnConnectFn(myConnectFn);
  sfOnStartupFn(myStartupFn);

  /* start up, give it control */
  sfStartup(0);
}

void myStartupFn(void)
{
  sfSetDisplayState(sfDISPLAY, 2);    /* set it to 5 Hz */
}

void myConnectFn(void)         /* start those processes */
{
  sfInitBasicProcs();
  sfInitRegistrationProcs();
  sfInitSpecialProcs();
  sfInitInterpretationProcs();
  sfInitControlProcs();
  sfInitAwareProcs();
  sfInitProcess(test_control_proc,"User Process");
}
```

The user micro-task (*test_control_proc*) is very simple: it starts up several behaviors and one PRS intention, then puts itself into a suspended state. You can change the state of the invoked behaviors, intentions, and processes from Saphira's **Function** menu (see previous chapter). All of the behaviors used in this function are available as part of the Saphira library.

```
void test_control_proc(void)
{
  switch(process_state)
    {
    case INIT:
      sfPreferredTurnDir = sfLEFTTURN;
      sfInitBehavior(sfConstantVelocity, 3, False,
                     sfFLOAT, 300.0,
                     sfEND);
      sfInitBehavior(sfStop, 4, False,
                          sfEND);

      sfInitBehavior(sfAvoidCollision, 0, False,
                     sfFLOAT, 3.0, /* front sensitivity */
                     sfFLOAT, 3.0, /* side sensitivity */
                     sfFLOAT, sfSHARPLY, /* turn gain */
                     sfFLOAT, 100.0, /* standoff */
                     sfEND);

      sfInitBehavior(sfKeepOff, 1, False,
                     sfFLOAT, 100.0, /* caution speed */
                     sfFLOAT, 0.25, /* sensitivity */
                     sfEND);
      sfInitIntention(sfFindAndFollow, "follow corr", 0, sfEND);
      sfSetProcessState(sfFindProcess("follow corr"),
                            sfINTERRUPT);
      process_state = SUSPEND;
      break;
    case RESUME:
      sfMessage("Resumed");
      break;
    }
}
```

The Saphira main window system passes keystrokes to your process via the callback registered with **sfKeyProcFn**. This callback should return 0 if the you want the default key action: moving the robot when the user presses one of the movement keys, for example. Otherwise, the function should return 1 to signal that it has handled the keypress. If you don't want to perform any special keyboard actions, you don't have to register a callback.

Similarly, mouse clicks are sent to the callback registered with **sfButtonProcFn**. Again, returning 0 from the callback means the default action is invoked; returning 1 means the callback handled the mouse click.

```
int myButtonFn(int x, int y, int b)
{
  return 0;              /*  don't do default handling */
}


int myKeyFn(int ch)      /* any user processing here */
{
  BEHCLOSURE b;
  switch(ch)
    {
    case SPACEKEY:
      if (NULL != (b = sfFindBehavior("sfConstantVelocity")))
              {
                if (b->running) sfMessage("Constant Vel OFF");
                sfSetBehaviorState(b, sfOFF);
              }
      sfSetVelocity(0);  /* stop the robot */
      break;

    …
      return 1;
    }
  return 0;              /* return 0 if we don't process it */
}
```

The example keypress callback above uses API calls for finding active behaviors, changing their state, and printing messages to the user. These calls are all documented in the following chapter. The mouse callback simply returns 0, invoking the default mouse click action. Note that the mouse callback could have been omitted; it's simply here to illustrate how a mouse callback is invoked.

4.3.5. The `Packet` Client

This client handles low-level communication with the robot server, bypassing Saphira's OS, which is never started. It takes advantage of the low-level Saphira communication routines, which parse packets and put the information into the state reflector structures.

```
#include "saphira.h"

void main(int argc, char **argv)
{
  int i = 0;

  /* open up the connection, to the simulator or robot */

  if (!sfConnectToRobot(sfTTYPORT, sfCOM1)) /* tty port */
    {
      printf("Couldn't open robot!\n");
      exit(0);
    }

  sfRobotComInt(sfCOMOPEN,1);   /* open the motor controller */
  sfResetRobotVars();           /* reset all app variables */
  sfRobotCom(sfCOMPULSE);       /* ask for data */
  sfRobotComInt(sfCOMVEL, 300); /* move forward at 300 mm/sec */

  /* read 100 packets */
```

```
  while (i<100)
    {
      if (sfWaitClientPacket(1000)) /* wait 1 sec for packet */
              {
                i++;
                sfProcessClientPacket(sfReadClientByte());
              }
      if (i % 10 == 0)
              {
                sfRobotCom(PULSE_COM); /* keep asking */
                sfRobotComInt(VEL_COM, 300); /* keep it going */
                printf("%d packets received\n", i);
                printf("X: %f  Y: %f\n", sfRobot.x, sfRobot.y);
              }
    }

  sfRobotComInt(sfCOMVEL, 0);   /* stop the robot */
  sfDisconnectFromRobot();
}
```

In the main function, the client starts by opening a connection to the robot, and then tells it to move forward at 300 millimeters per second. The robot sends an information packet to the client every 100 milliseconds; the main client loop collects 100 server packets, parsing each into the state reflector structures. After the packets, the client stops the robot, disconnects from the server, and ends execution.

5. Saphira Servers

In the Saphira client/server model, the robot server works to manage all the low-level details of the robot's systems, including operating the drives, firing the sonars and collecting echoes, and so on, on command from and reporting to a separate client application, such as Saphira. With Pioneer, this is the Pioneer Server Operating System (PSOS. The capabilities of the Pioneer robot server, and its connection to the client, are shown in Figure 5-1.

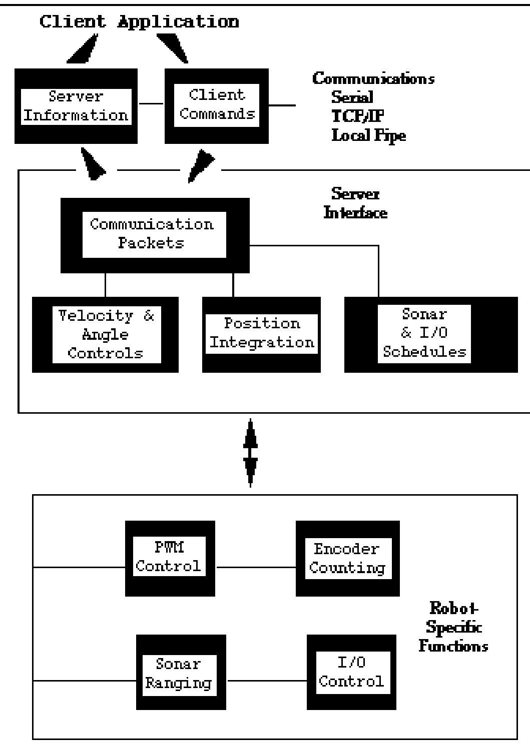

Figure 5-1 Saphira client-robot server architecture

High-level robotics applications developers do not need to know many details about a particular robot server, because the Saphira client insulates them from this lowest level of control. Some of you, however, may want to write your own robotics control and reactive planning programs, or just would like to have a closer programming relationship with your robot. This chapter explains how to communicate with your robot via the Saphira client/server interface. The functions and commands, of course, are supported in the Saphira C libraries that came with your robot, but not every robot supports all commands. Please consult your robot's operation manual or Saphira supplement for those details.

5.1. Communication Packet Protocol

The Saphira-mediated robot or its simulator communicates with a client application using a special packet protocol. It is a bit stream consisting of four main elements (Table 5-1): a two-byte header, a one-byte count of the number of data and checksum bytes in the packet, a client command including arguments or a server information data block, and ending with a two-byte checksum.

Table 5-1 Main elements of PSOS communication packet protocol

Component	Bytes	Value	Description
Header	2	0xFA, 0xFB	Packet header; same for client and server
Byte Count	1	N + 2	Number of subsequent data bytes plus checksum; must be < 200 total bytes long
Data	N	command or SIB	Client command or server information block (discussed in subsequent sections)
Checksum	2	computed	Packet integrity checksum

5.1.1. Packet Data Types

Packetized client commands and server information blocks use several data types, as defined in Table 5-2. There is no convention for sign; each packet type is interpreted idiosyncratically by the receiver. Negative integers are sign-extended.

Table 5-2 Communication packet data types

Data Type	Byte Count	Byte Order
Integer	2	b_0 low byte; b_1 high byte
Word	4	b_0 low byte; b_3 high byte
String	up to ~200, length-prefixed	b_0 length of string; b_1 first byte of string
String	unlimited null-terminated	b_0 first byte of string; 0 (null) last byte

5.1.2. Packet Checksum

A communication packet checksum is derived by successively adding data byte pairs (high byte first) to the running checksum (initially zero), disregarding sign and overflow. If there is an odd number of data bytes, the last byte is XOR-ed in to the low-order byte of the checksum.

NOTE: The checksum word is placed at the end of the packet with its bytes in the reverse order of that used for arguments and data; that is, b_0 is the high byte, and b_1 is the low byte.

Use the following C-code fragment in your client applications to compute a checksum:

```
int
calc_chksum(unsigned char *ptr) /* ptr is array of bytes, first is data count
*/
{
  int n;
  int c = 0;
  n = *(ptr++);
  n -= 2;                      /* don't use chksum word */
  while (n > 1) {
    c += (*(ptr)<<8) | *(ptr+1);
    c = c & 0xffff;
    n -= 2;
    ptr += 2;
  }
  if (n > 0) c = c ^ (int)*(ptr++);
  return(c);
}
```

5.1.3. Packet Errors

Currently, the Saphira server interface ignores a client command packet whose byte count exceeds 200 or has an erroneous checksum. The client should similarly ignore erroneous server information packets (Saphira does).

The Saphira client/server interface does not acknowledge receipt of a command packet, nor has any facility to handle client acknowledgment of a server information packet. Hence, Saphira client/server communication is as reliable as the physical communication link. UNIX pipes with the simulator or a cable tether between the robot and client computer are very reliable links. Radio modem-mediated communication is much less reliable. Accordingly, when designing client applications that may use radio modems, do not expect to receive every information packet intact, nor have every command accepted by the server.

The design decision to provide an unacknowledged packet interface is a consequence of the realtime nature of the client/server interaction. Simply retransmitting server information blocks or command packets would result in antiquated data not at all useful for a reactive client or server.

For some operations, however, the data do not decay as rapidly: some commands are not overly time-sensitive, such as those that perform housekeeping functions like changing the sonar polling sequence. It would be useful to have a reliable packet protocol for these operations, and we are considering this for a future release of Saphira server interface.

In the meantime, the Saphira client/server interface provides a simple means for dealing with ignored command packets: Most of the client commands alter state variables in the server. By examining those values in the server information packet, client software may detect ignored commands and reissue them until achieving the correct state.

5.2. Client Commands

Saphira client/server interface implements a structured command format for receiving and responding to directions from the client for control and operation of the robot or its simulator. You may send client commands to the robot at a maximum rate of once every 100 milliseconds. The client must send a command at least once every two seconds; otherwise, the server will stop the robot’s onboard drives.

The client command is comprised of a one-byte command number optionally followed by, if required by the command, a one-byte description of the argument type and the argument. To work, of course, the client command and its optional argument must be included as the data component of a client communication packet (Table 5-3; also see earlier sections of this chapter).

Table 5-4 contains the list and brief descriptions of the currently implemented Saphira client commands, which we discuss in detail in following sections. These and additional server operating commands used by most, but not all, Saphira-enabled robots, also appear in the Saphira header file

handler/include/saphira.h. Check your robot's operation manual, Saphira supplement, and Saphira distribution *UPDATE* text file for the latest details.

Table 5-3 Client command communication packet

Component	Bytes	Value	Description
Header	2	0xFA, 0xFB	Packet header; same for client and server
Byte Count	1	N + 2	Number of command bytes plus checksum; must be < 200 total bytes long
Command Number	1	0 - 255	Client command number; see Table 4-4
Arg Type (optional)	1	0x3B or 0x1B or 0x2B	Data type of command argument, if included: (*sfARGINT*) positive integer (*sfARGNINT*) negative int or absolute value (*sfARGSTR*) string, null-terminated
Argument (optional)	N	data	Command argument; integer or null-terminated string
Checksum	2	computed	Packet integrity checksum

5.2.1. Client Command Argument Types

There are three different types of client command arguments: positive integers two bytes long, negative integers two bytes long, and strings of up to 195 characters long (200-byte limit on packets) terminated with a 0 (NULL). Byte order is least-significant byte first. Negative integers are transmitted as their absolute value (unlike information packets, which use sign extension for negative integers; see below). The argument is either an integer, a string, or nothing, depending on the command.

5.2.2. Saphira Client Command Support

Saphira fully supports client commands with useful library functions. Prototypes can be found in **handler/include/saphira.h** and **saphira.pro**. See Chapters 5 and 6 for details.

Table 5-4 PSOS 4.2 supported client commands

Command Name	Number	Argument Value(s)	Description
sfSYNC0	0	none	Start connection; server echoes these synchronization commands back to client.
sfSYNC1	1	none	
sfSYNC2	2	none	
sfCOMPULSE	0	none	Communication pulse
sfCOMOPEN	1	none	Open the motor controller
sfCOMCLOSE	2	none	Close server and client connection
sfCOMPOLLING	3	string	Set sonar polling sequence
sfCOMSETO	7	none	Set server origin
sfCOMVEL	11	signed int mm/sec	Forward (+) or reverse (-) velocity
sfCOMHEAD	12	unsigned int degrees	Turn to absolute heading 0-360 degrees
sfCOMDHEAD	13	signed int degrees	Turn heading +-255 degrees
sfCOMRVEL	21	signed int degrees/sec	Set rotational velocity +- 255 degrees/sec
sfCOMVEL2	32	2 bytes 4*mm/sec	Set wheel velocities independently +-4mm/sec
sfCOMDIGOUT	30	integer bits 0-7	Set digital output bits
sfCOMTIMER	31	integer pin 0-7	Initiate user input timer, triggering an event with specified pin
sfCOMGRIPPER	33	integer 0, 1, 4, 5	Sets gripper state
sfCOMPTUPOS	41	integer 1-2000 ms	Set pulse-width for position servo control.
sfCOMSTEP	64	none	Single-step mode (simulator only)

5.3. Server Information Packets

The Saphira-aware server automatically sends a packet of information over the communication port back to the client every 100 milliseconds. The server information packet informs the client about a number of the robot's operating parameters and readings, using the order and data types shown in Table 5-5. Your client application may use the Saphira library function **sfProcessClientPacket** to parse the server information and deposit the results in various buffers of the state reflector. See the section on the state reflector in the API reference for information about these structures.

Table 5-5 Saphira server information data packet (minimum contents)

Name	Data Type	Description
Header	int	Exactly 0xFA, 0xFB
Byte Count	byte	Number of data bytes + 2; must be less than 201 (0xC9)
Status	byte = 0x3S; where S =	Motors status
	sfSTATUSNOPOWER	Motors power off
	sfSTATUSSTOPPED	Motors stopped
	sfSTATUSMOVING	Robot moving
Xpos	unsigned int (15 ls-bits)	Wheel-encoder integrated coordinates; platform-dependent units—multiply by
Ypos	unsigned int (15 ls-bits)	*DistConvFactor* in the parameter file to convert to mm; roll-over ~ 3 m
Th pos	signed int	Orientation in platform-dependent units—multiply by *AngleConvFactor* for radians
L vel	signed int	Wheel velocities (respective Left and Right) in platform-dependent units—
R vel	signed int	multiply by *VelConvFactor* to convert to mm/sec.
Battery	byte	Battery charge in tenths of volts
Bumpers	2 bytes - L and R	Motor stall indicators
Bumpers	unsigned int	
Control	signed int	Setpoint of the server's angular position servo—multiply by *AngleConvFactor* for radians
PTU	unsigned int	Pulse width of position servo
Say	byte	verbal/sound clues
Sonar readings	byte	Number of new sonar readings included in information packet; readings follow:
Sonar num	byte	Sonar number
Sonar range	unsigned int	Sonar reading—multiply by *RangeConvFactor* for mm
... *rest of the sonar readings ...*		
Input timer	unsigned int	User input timer reading
User Analog	byte	User analog input reading
User Input	byte	User digital input pins
User Output	byte	User digital output pins
Checksum	int	Checksum (see previous section)

In future versions, server information packets may contain additional, appended data fields. To remain compatible, have your client application accept the entire data packet, even though it may use only a few selected fields.

5.4. *Start Up and Shut Down*

Before exerting any control, a client application must first establish a connection to the robot server via an RS-232 serial link (9,600 baud), a interprocess connection (UNIX pipe, for example, or MS Windows mailslot), or TCP/IP network (Table 5-6. Port types and names for clinet/server connections). Over that established communication link, the client then sends commands to and receives back operating information from the server.

Table 5-6. Port types and names for clinet/server connections

Port types	*sfLOCALPORT*	connect to simulator on the host machine
	sfTTYPORT	connect to robot on a tty port
	sfTCPPORT	connect to robot on over TCP/IP network
Port names	*sfCOMLOCAL*	local pipe or mailslot name
	sfCOM1	tty port 1 (/dev/ttya or /dev/cua0 for UNIX; COM1 for MSW; modem for Mac)
	sfCOM2	tty port 2 (/dev/ttyb or /dev/cua1 for UNIX, COM2 for MSW, printer for Mac)
	SERVER_NAME	hostname/IP address of server (not for Pioneer)

5.4.1. Synchronization—sfCOMSYNC

When first started, the Saphira-aware server, including the simulator, is in a “wait” state listening for communication packets over its designated port. (See your robot operating manual for details about your robot’s servers.) To establish a connection, the client application sends a series of three synchronization packets through the host communication port—*sfSYNC0*, *sfSYNC1*, and *sfSYNC2*, in succession. The server responds to each, forming a succession of identical synchronization packets. The client should listen for the returned packets and only issue the next synchronization packet after it has received the echo.

A string may be used for unusual port names—if there is a serial communications card with extra tty ports, for instance. With Macintosh, it’s best to use the **`modem`** port, if it’s available, rather than the **`printer`** port.

5.4.2. Autoconfiguration

The Saphira-aware servers (PSOS v4.1 or later, for example) send configuration information back to the client in the last sync packet (*sfSYNC2*). After the sync byte, there are 3 null-terminated strings that represent the robot name, robot class, and robot subclass (Table 5-7). You can read these strings with the Saphira function *sfReadClientString*. The following table shows what these strings are for different robots.

Table 5-7. Robot configuration information

Name string	"Robot" for Pioneer-class robots Computer name for Bxx--class robots "Simulator" for the simulator
Class string	"Pioneer", "B14", or "B21"
Subclass string	PSOS version for Pioneer-class robots and their simulator Null string for other robots and simulators

The parameter file that is appropriate for a robot can be found in the Saphira **`params`** directory. The name of the parameter file will be the same as the lowercase version of the subclass string (if it exists) or the class string.

5.4.3. Opening the Servers—`sfCOMOPEN`

Once the communication link is established, the client should then send the **`sfCOMOPEN`** command, which causes the robot or the simulator to perform some housekeeping functions, start the sonar and motor controllers (among other things), start listening for client commands, and to begin transmitting server information.

5.4.4. Keeping the Beat—`sfCOMPULSE`

As mentioned earlier, a server "safety watchdog" expects that the robot receives at least one communication packet from the client every two seconds. Otherwise, it assumes the client/server connection is broken and shuts down the robot's motors. If your client application will be otherwise distracted for some time, periodically issue the **`sfCOMPULSE`** client command to let the server know you are indeed alive and well. If the robot shuts down due to lack of communications traffic, it will revive upon receipt of a client command and automatically accelerate to the last-specified speed at the current heading.

5.4.5. Closing the Connection—`sfCOMCLOSE`

To close a connection and reset the server to the wait state, simply issue the client **`sfCOMCLOSE`** command.

5.4.6. Movement Commands

As of PSOS 4.2, the robot server accepts several different types of motion commands. You can set the turn angle or velocity, and the forward/back velocity; or, you can control the two wheel velocities independently. Table 5-8 summarizes the command modes available.

Table 5-8. Server motion command types

Rotation		Translation	
`sfCOMHEAD`	absolute heading		
`sfCOMDHEAD`	differential heading		
`sfCOMRVEL`	rotational velocity	**`sfCOMVEL`**	forward/back velocity
`sfCOMVEL2`	left and right wheel velocities		

The robot server automatically switches to the required motion control mode when it receives one of these commands. For example, if it is in two-wheel velocity mode, and it is sent an **`sfCOMHEAD`** command, it abandons two-wheel velocity mode and starts controlling the heading and velocity of the robot.

The arguments for these commands are given in Table 5-9 below. The heading commands are with respect to the robot's internal coordinate system (see the section below).

Table 5-9. Motion command arguments

Command	Argument(s)	Typical invocation
`sfCOMHEAD`	degrees (int) [0, 360]	`sfRobotComInt(sfCOMHEAD, 320)`
`sfCOMDHEAD`	degrees (int) [-180, 180]	`sfRobotComInt(sfCOMDHEAD, -10)`
`sfCOMRVEL`	degrees/sec (int) [-200, 200]	`sfRobotComInt(sfCOMRVEL, -80)`
`sfCOMVEL`	mm/sec (int) [-400, 400]	`sfRobotComInt(sfCOMVEL, 150)`
`sfCOMVEL2`	4 mm/sec (int) [-100, 100]	`sfRobotCom2Bytes(sfCOMVEL2,40,50)`

The Saphira-aware robot server will try to make the robot achieve the desired velocity and heading as soon as the commands are received, using its internal (de)acceleration managers. Check your robot's operation manual to find its absolute maximum achievable motion and rotational velocities.

5.5. Robot in Motion

When the Saphira-aware robot server receives a velocity command, it accelerates at a constant rate set internally to the speed you provided as the argument for **`sfCOMVEL`**. Rotational headings are achieved by a trapezoidal velocity function (Figure 5-2). This function is re-computed each time a new heading command is received, making on-the-fly orientation changes possible.

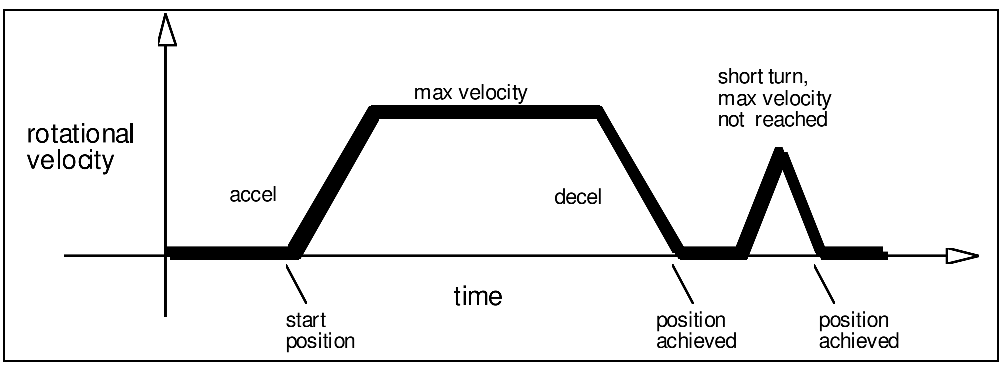

Figure 5-2 Trapezoidal turning velocity profile

5.5.1. Position Integration

Depending on your robot, it keeps track of its position and orientation based on dead-reckoning from wheel motion, which is an *internal coordinate position*. A server command, **`sfCOMSETO`**, resets the robot server's internal x,y position coordinates to (0,0,0).

Registration between external and internal coordinates deteriorates rapidly with movement, due to gearbox play, wheel imbalance and slippage, and many other real-world factors. You can rely on the dead-reckoning ability of the robot for just a short range—on the order of several meter and one revolution, depending on the surface (carpets tend to be worse than hard floors).

Also, moving either too fast or too slow tends to exacerbate the absolute position errors. Accordingly, consider the robot's dead-reckoning capability as a means of tying together sensor readings taken over a short period of time, not as a method of keeping the robot on course with respect to a global map.

The orientation commands **sfCOMHEAD** and **sfCOMDHEAD** turn the robot with respect to its internal dead-reckoned angle (Figure 5-3). On startup, the robot is at the origin (0,0), pointing towards the positive x-axis at 0 degrees. Absolute angles vary between 0 and 360 degrees. As the robot moves, it will update this internal position based on dead-reckoning. The x,y position is always positive, and rolls over at about 3,000 millimeters. So, if the is at position (400,2900) and moves +400 millimeters along the y-axis and -600 millimeters along the x-axis, its new position will be (2800, 300).

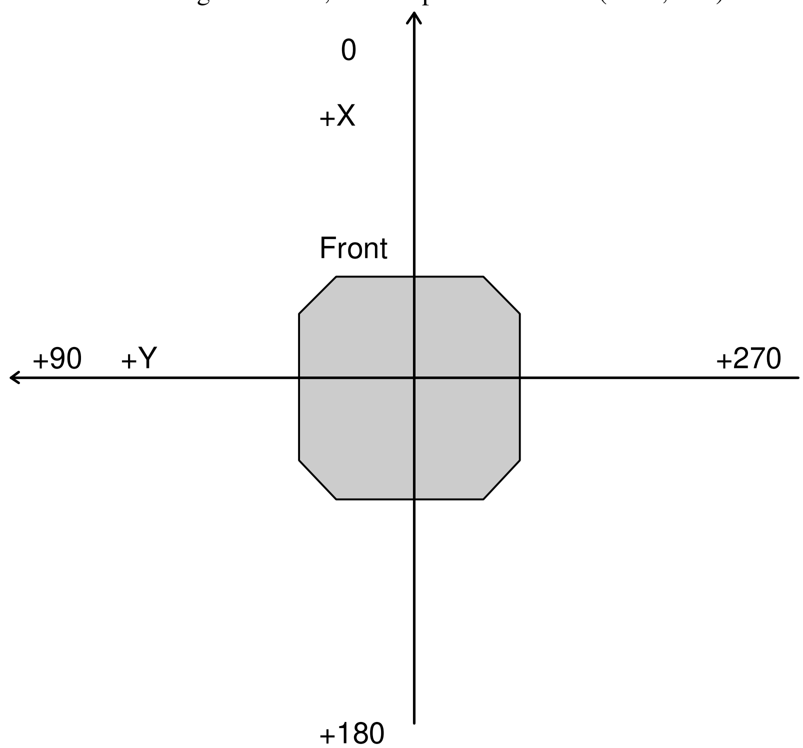

Figure 5-3 Saphira-aware server internal coordinate system

5.6. Sonars

When opened by the appropriate client command (see **sfCOMOPEN** above), the Saphira-aware robot server automatically coordinates and begins firing the robot sonars in a pre-defined default sequence, and sends the results to the client via the server information packet. Details about the configuration and firing sequence of the sonars are found in the robot's operation manual.

Use the **sfCOMPOLLING** command to change the polling sequence of the sonars:

```
sfRobotComStr(sfCOMPOLLING, str)
```

where str is a null-terminated string of bytes at most 12 bytes long. Each byte is 1 + sonar number. For example, the string

```
"\001\002\001\006"
```

starts the sonar polling sequence 0, 1, 0, 5. Note that sonar numbers can be repeated. If the string is empty, all sonars are turned off.

6. Guide to the Saphira API

This chapter details the current library of functions for development of a Saphira client. Additional information about prototypes, structures, and variables can be found in the various header files in the **`handler/include/`** directory of your Saphira distribution. Also study the **`saphira.c`** source file in that distribution as an example of a working Saphira application.

6.1. Saphira OS Functions

Use the following functions to initialize, configure, and operate the Saphira OS (see Section 2 for a summary of OS properties).

```
void sfStartup (int async)
void sfStartup (HANDLE hInst, int cmdShow, int async)
void sfPause(int ms)
```

The first form is for UNIX systems, the second for MS Windows. When invoked, **`sfStartUp`** initializes the Saphira OS. If the client has been linked with the window libraries, a user interface window is opened and Saphira information is displayed graphically.

If *async* is 0, Saphira has principal control of the client and thereafter calls other functions only from the Saphira multi-tasking OS (see below). If *async* is 1, control returns immediately to the calling program, and the Saphira interface runs as a separate thread.

The **`sfStartUp`** function may be called at any time by your program, but should be called only once. Also include with the Windows version of this function the application instance handle (*hInst*) and the window visibility parameter (*cmdShow*).

If the client program is running asynchronously, in parallel with the Saphira OS, then it may be useful to insert timing breaks in the client code. The appropriate method is with **`sfPause`**, which waits a specified number of milliseconds before continuing. **`sfPause`** allows the Saphira OS to keep running during the break.

```
void sfOnStartupFn (void (*fn)())
void sfOnConnectFn (void (*fn)())
void sfOnDisconnectFn (void (*fn)())
int  sfIsConnected
```

These functions register callbacks for Saphira events: when the Saphira OS first starts up, when it connects to a robot, and when it disconnects. None of these callbacks are obligatory; usually the connect callback, at least, is registered. The startup callback should include any relevant initialization code, such as menu or directory settings, in this function. The connect callback should start processes, behaviors, and other Saphira control routines. The disconnect callback can be used to clean up after the Saphira client disconnects from a robot.

```
void sfSetDisplayState (int menu, int state)
```

Use the **`sfSetDisplayState`** function to change the state of a display mode in the Saphira window interface. If you call this function before connecting to the robot (in the startup callback), it will set the default state for the display function. Thereafter, the preset display values are *sticky*—Saphira automatically resets them to the preset values, perhaps different from the defaults given in Table 6-1), whenever a new connection is made with the robot.

Table 6-1 Optional states for various Saphira display functions

Menu	State (int)*	Description
`sfDISPLAY`	0-10; **2**	Controls display update rate. State is the number of 100 ms cycles between updates. Value 10 is once per second, for example. Value of 0 turns the display off.
`sfGLOBAL`	*TRUE*, ***FALSE***	Controls local/global viewpoint of display window.
`sfWAKE`	*TRUE*, ***FALSE***	Controls drawing of breadcrumb wake behind robot.
`sfSTEP`	*TRUE*, ***FALSE***	Controls single-step mode when connected to the Pioneer simulator.
`sfOCCGRID`	*TRUE*, ***FALSE***	Controls display of occupancy grid results. If enabled, enables global viewpoint.

* Default state values are in **bold** typeface.

```
void sfMessage (char *str)
void sfSMessage (char *str, ...)
void sfErrMessage (char *str)
void sfErrSMessage (char *str, ...)
```

`sfMessage` writes the null-terminated string **`str`** into the message section of the information area in the Saphira main window. Use **`sfSMessage`** to format the string similar to the standard **`printf`** C function, which accepts optional arguments that are to be inserted into the string

The second set of functions write into the error message section of the information area.

```
void sfKeyProcFn (int (*fn)())
int  myKeyFn(int ch)
```

(The **`sfKeyProcFn`** registers an optional user key process callback, with the prototype of **`myKeyFn.`** It is called by Saphira whenever the user presses a key when the main Saphira window is active. The **`ch`** argument is the character representing the key that was pressed and is operating-system dependent. Return 0 if you don't handle the keypress; return 1 if you do, particularly to over-ride any of Saphira's built-in key processing routines (see Table 6-1).

```
void  sfButtonProcFn (int (*fn)())
int   myButtonFn (int x, int y, int b)
float sfScreenToWorldX (int x, int y)
float sfScreenToWorldY(int x, int y)
```

The **`sfButtonProcFn`** registers an optional user button process callback, with the prototype of **`myButtonFn`**. It is called by Saphira whenever the user clicks the mouse when the main Saphira window is active. The *x* and *y* arguments are the screen position of the cursor; *b* is the mouse button, with the values **`sfButtonLeft`**, **`sfButtonRight`**, and **`sfButtonMiddle`**. Return 0 if you don't handle the mouse click; return 1 if you do, particularly to over-ride any of Saphira's built-in mouse processing routines.

To convert from screen to global robot coordinates, use the **`sfScreenToWorld`** functions, which return their answers in mm.

6.2. Predefined Saphira Processes

We've provided a variety of predefined Saphira processes for control of the robot. You may initiate these process sets using the API functions described here, or invoke processes individually using the **`sfInitProcess`** API call

void sfInitBasicProcs(void)

.Starts up a set of basic communication, display, motor, and sensor control processes. Among other activities, these processes implement the client state reflector. The processes invoked are:

pulse_proc	*communication pulse every 1 second*
motor_proc	*coordinates keyboard and behavior motor commands*
clamp_proc	*rotates the world around the robot*
sonar_proc	*adds new sonar readings to the sonar buffer*
wake_proc	*draws a wake of the robot's motion*
draw_proc	*updates Saphira display window*
process_waiting_packets	*parses information packets from robot server*

Drawing, wake, and clamping processes are affected by variables that users can set from Saphira's **main** window's **Display** menu.

void sfInitControlProcs(void)

Starts up a process for evaluating all active behaviors. If you want to run *without* using the fuzzy behavior controller, by using the direct motion functions, then don't initiate this process.

execute_current_behaviors *evaluates behaviors and outputs a motor control*

void sfInitInterpretationProcs (void)

Starts up processes for interpretation of sonar results:

occgrid_proc	*computes an occupancy grid*
side_segment_proc	*forms linear artifacts robot motion*
test_wall_proc	*wall recognition*
test_wall_break_proc	*door and junction recognition*

These processes must be started to have results deposited in **sfLeftWallHyp** and **sfRightWallHyp**.

void sfInitRegistrationProcs (void)

Starts up some position registration processes useful for navigation in an office environment:

test_match_proc	*matching of linear and point artifacts*
test_environment_proc	*identification of current situation*

6.3. State Reflection

State reflection is a way of isolating client programs from the work involved in send control commands and gathering sensory information from the robot. The *state reflector* is a set of data structures in the client that reflects the sensor and motor state of the robot. The client can examine sensor information by looking at the reflector data, and can control the robot by setting reflector control values. It is the responsibility of the Saphira OS to maintain the state reflector by communicating with the robot server, receiving information packets and parsing them into the state reflector, and sending command packets to implement the state reflector control values. The micro-tasks started by **sfInitBasicProcs** are the relevant ones: you must invoke this function for the state reflector to function.

There are three important data structures in the state reflector.

1. The **sfRobot** structure holds motion and position integration information, as well as some sensor readings (motor stall sensors, digital I/O ports).
2. The sonar buffers hold information about current and past sonar returns.
3. The control structures command robot motions.

This section describes the robot and sonar information structures; the next one, the direct motion commands that affect the control structures.

`struct robot sfRobot`

The sfRobot structure holds basic information reflected from the robot server. Table 6-2 below shows the values of the various fields in this structure; the definition is in **`handler/include/struct.h`**.

All of the values in the **`sfRobot`** structure are reflected from the robot server back to the client, providing information about the robot's state. In this way, it is possible to tell if a command has been executed. For example, the **`digoutput`** field reflects the actual value of the digital output bits set on the robot.

The interpretation of some of the values in the structure is robot-dependent, e.g., the **`bumpers`** field reflects motor stall information for the Pioneer robots. The Saphira library provides some convenience functions for interpreting these fields; see the following subsections.

Table 6-2 Definition of the `sfRobot` structure

`sfRobot` field	Units	Description
`x, y, th`	mm, mm, radians	robot's location in robot coords; always (0, 0, 0)
`ax, ay, ath`	mm, mm, radians	robot's global location
`tv, mtv`	mm/sec	current and max velocity
`rv, mrv`	deg/sec	current and max rotational velocity
`leftv, rightv`	mm/sec	left and right wheel velocities
`status`	int STATUS_STOPPED STATUS_MOVING STATUS_NOT_CONNECTED STATUS_NO_HIGH_POWER	robot status: *robot stopped* *robot moving* *client not connected* *robot motors stalled*
`battery`	1/10 volt	battery power
`bumpers`	int	bumper state
`ptu`	usecs	pan/tilt unit (servo) heading
`diginput`	int	digital input state
`digoutput`	int	digital output state
`analog`	0-255 [0V-5V]	analog input voltage
`motor_packet_count` **`sonar_packet_count`** **`vision_packet_count`**	counts per second	packet communication information

6.3.1. Motor Stall Function

On Pioneer-class robots, the motors stall if the robot encounters an obstacle. Each motor can stall independently, and this can yield information about where the obstacle is, e.g., if the right motor stalls, then the right wheel or right side of the robot is affected. However, you can't rely absolutely on this behavior, as sometimes both motors will stall even when the obstacle is on one side or the other. Motor stall information is returned in the **`bumpers`** field.

```
int sfStalledMotor (int which)
```

Return 1 if the motor *which* is stalled, and 0 if it isn't. The argument *which* is *sfLEFT* or *sfRIGHT*.

6.3.2. Sonar buckets

The current range reading of a sonar sensors is held in an **`sdata`** structure, defined below. The structures for all the sonars are in an array called **`sbucket`**, e.g., **`sbucket[2]`** is the **`sdata`** structure for sonar number 2.

Fields in the sdata structure indicate the robot's position when the sonar was fired, the range of the sonar reading, and the position in robot coordinates of the point on the sonar axis at the range of the reading. The field snew is set to 0xFFFF when a new reading is received; the client program can poll this field to ascertain if the reading is new, and set it to 0 to indicate that it has been read.

A value of 5000 for the sonar range indicates that no echo was received after the sonar fired and waited for a return.

Some convenience functions for accessing current sonar readings are described below.

Sonar readings are accumulated over short periods of time into a set of buffers in the LPS; see below in the section on the LPS.

```
typedef struct           /* sonar data collection buffer */
{
  float fx, fy, fth;     /* robot position when sonar read */
  float afx, afy, afth;  /* absolute position when sonar read */
  float x, y;            /* sonar reading in flakey RW coords */
  int range;             /* sonar range reading in mm */
  int snew;              /* whether it's a new reading */
} sdata;

IMPORT extern sdata sbucket[];  /* holds one sdata per sonar, indexed by sonar
number */
```

```
int   sfSonarRange(int num)
float sfSonarXCoord(int num)
float sfSonarYCoord(int num)
int   sfSonarNew(int num)
```

The first three functions return the range and x,y coordinates of the sonar reading. The last function returns 1 if it's a new reading, 0 if not; it also resets the **`new`** flag to 0 so that the same reading won't be returned twice.

6.4. Direct Motion Control

Direct motion control uses the state reflector capability of the Saphira OS to implement a useful client-side motion control system. Instead of sending motor commands to the server, a client sets motion setpoints in the state reflector. The OS takes care of transmitting appropriate motor commands to the robot.

Direct motion control offers three advantages over sending motor control packets directly.

1. It checks that the setpoints are actually sent to the robot server, given the unreliability of the communication channel.
2. It implements a set of checking functions for determining when the motion commands are finished.

3. It has a position control mode which moves the robot a specified distance forward or backward.

Direct control of the two control channels (translation and rotation) is independent, and commands to control them can be issued and will execute concurrently.

The direct motion functions require the state reflector to be operational, that is, the function **`sfInitBasicProcs`** must be called.

```
void sfSetVelocity(int vel)
void sfSetRVelocity(int rvel)
```

Set the translational and rotational setpoints in the state reflector. If the state reflector is active, these setpoints are transferred to the robot. Values for translational velocity are in mm/sec; for rotational velocity, degrees/sec.

```
void sfSetHeading(int head)
void sfSetDHeading(int dhead)
```

The first function sets the absolute heading setpoint in the state reflector. The argument is in degrees, from 0 to 359.

The second function increments or decrements the heading setpoint. The argument is in degrees, from -180 to +180.

If the state reflector is active, the heading setpoint is transferred to the robot.

```
Void sfSetPosition(int dist)
void sfSetMaxVelocity(int vel)
```

The first function sets the distance setpoint in the state reflector. Argument is in mm, either positive (forwards) or negative (backwards). If the state reflector is active, it sends motion commands to the robot to move the required distance. The maximum velocity attained during motion is given by **`sfSetMaxVelocity`**, in mm/sec.

```
int sfDonePosition(int dist)
int sfDoneHeading(int ang)
```

Checks whether a previously-issued direct motion command has completed. The argument indicates how close the robot has to get to the commanded position or heading before it is considered completed. Arguments are in mm for position, and degrees for heading. On a Pioneer robot, you should use at least 100 mm for the distance completion, and 10 degrees for angle. The robot may not move enough to trigger the completion function otherwise. Note that, even though the robot may not achieve a given heading very precisely if it is just turning in a circle, as it moves forward or backward it will track the heading better.

```
float sfTargetVel(void)
float sfTargetHead(void)
```

These functions return the current reflected values for the velocity and heading setpoints, respectively. Values are in mm/sec and degrees.

6.5. Saphira Multi-tasking

One problem facing any high-level robotics controller is developing an adequate real-time base for the many concurrent processes that must be run. Rather than depend on the machine OS for this capability, we have implemented a simple "round-robin" cooperative scheme that places responsibility on each individual process to complete its task in a timely and reasonable manner. Each process is called a *micro-task*, because it accomplishes just a limited amount of work.

Compute-intensive processes that take a long time to complete, but can execute asynchronously with the Saphira system, can be implemented as concurrently executing threads. Accordingly, use the Saphira **`sfStartup`** function with an **`async`** argument of 1 and prepare your processes so they execute as a concurrent thread, as we describe below.

6.5.1. Process Definition

Running processes are functions with no arguments together with state information. Processes access their state through a global integer variable, **`process_state`**. Processes are initiated by an API call, **`sfInitProcess`**, which places the function onto the process stack. Once initialized, Saphira will call the process with an initial state of **`sfINIT`**. The process can change its state by setting the value of **`process_state`**. User-defined state values are integers greater than 10; values less than 10 are reserved for special states (Table 6-3).

Table 6-3. Saphira multiprocessing reserved process state values

State	Explanation
`sfINIT`	Initial state
`sfSUSPEND`	Suspended state
`sfRESUME`	Resumed state
`sfINTERRUPT`	Interrupted state
`-n`	Suspend this process for *n* cycles

Process cycle time is 100 milliseconds. On every cycle, Saphira calls each process with **`process_state`** set to the current value for that process. The process may change its state by resetting **`process_state`**. A process may suspend itself by setting the state to **`sfSUSPEND`**. Some other process or your program must resume a suspended process (see below for relevant functions). A process may also suspend itself for *n* cycles by setting **`process_state`** to *-n*, in which case it will resume after the allotted time with state **`sfRESUME`**.

The **`sfINTERRUPT`** state indicates an interrupt request from another process or the user. Processes should be written to respond to interrupts by saving needed information, then suspending until receipt of a resume request. Many of Saphira's predefined processes are written in this way.

The fixed cycle time of process invocation means that processes can have guaranteed response time for critical tasks; a controller can issue a command every 100 millisecond, for example. Of course, response time depends on the conformity of all processes: the combined execution time of all processes must never exceed 100 milliseconds. If it does, the cycle time will exceed 100 milliseconds for all processes. Hence, allow around 2-5 milliseconds compute time per process, and either divide large processes into smaller pieces, each able to execute within the 2-5 millisecond time frame, or run them as concurrent threads.

Here is an example of a typical interpretation process function. It starts by setting up some housekeeping variables, then proceeds to alternate door recognition with display of its results every second or so.

```
#define FD_FIND 20
#define FD_DISPLAY 21
void find_doors(void)
{
      int found_one;
      switch(process_state)
      {
        case sfINIT:       /* Come here on startup */
           found_one = 0;
           { ... }
           process_state = FD_FIND;
           break;
        case sfRESUME:     /* Come here after suspend */
           process_state = FD_FIND;
           break;
        case sfINTERRUPT:  /* Interrupt request */
           found_one = 0;
           process_state = sfSUSPEND;
           break;
        case FD_FIND:      /* Looking for doors */
           { call recognition function }
           process_state = FD_DISPLAY;
           break;
        case FD_DISPLAY:   /* Now we display it */
           if (found_one)
           { call display function }
           process_state = -8;  /* suspend for 8 ticks */
           break;
        }
}
```

6.5.2. Process Manipulation

When instantiating a process, give it a unique string name and later refer to it by name or pointer. The following Saphira functions initiate, suspend, and resume processes:

```
process *sfInitProcess (void *fn(void), char *name)
```

The **sfInitProcess** function starts up a process with the name **name** and function **fn**, and returns the process instance pointer, which can be used in process-manipulation functions. There is no corresponding function for deleting processes—suspend it if it is no longer needed.

```
process *sfFindProcess (char *name)
```

The **sfFindProcess** function searches for and returns the first process instance it finds with the name **name**. A process instance pointer is returned if successful; else **NULL**.

```
void sfSetProcessState (process *p, int state)
void sfSuspendProcess (process *p, int n)
void sfSuspendSelf (int n)
void sfInterruptProcess (process *p)
void sfInterruptSelf (void)
void sfResumeProcess (process *p)
```

The *sfSetProcessState* function sets the state of process instance *p* to *state*. The argument *p* must be a valid process instance pointer, returned from *sfFindProcess* or *sfInitProcess*. The other functions are particular calls to *sfSetProcessState*.

6.6. Local Perceptual Space

Local Perceptual Space (LPS) is a geometric representation of the robot and its immediate environment. Unlike the internal coordinate system we described earlier in Chapter 4, which represents the dead-reckoned position of the robot server, the LPS is an egocentric coordinate space that remains clamped to the robot center (Figure 6-1).

6.6.1. Sonar buffers

The current range readings of all the sonars can be found in the sonar bucket structures: see the section on the state reflector above. As the robot moves, these readings are accumulated in the LPS in three internal buffers. These buffers are available to user programs, and are also used by the obstacle-finding functions in the next subsection.

The reading values are placed on the centerline of the sonar at the range that the sonar indicates. Saphira's display routines draw sonar readings as small open rectangles, and if the robot moves about enough, they give a good picture of the world.

There three buffers are the front and two side buffers (left and right). Each buffer is a **`cbuf`** structure, defined below. Client programs, unless they are interested in the temporal sequence of sonar readings, can just treat these buffers as linear structures with size **`limit`**. The buffer size can be changed using the functions defined below.

The reason for having different buffers is that they satisfy different needs of the robot control software. The front sonars, pointed in the direction of the robot's travel, warn when obstacles are approaching. But the spatial definition of these sonars isn't very good, and its almost impossible to distinguish the shape of the obstacle for recognition purposes. A wall in front of the robot, for example, will only look a little bit like a straight line (see the excellent book by Leonard and Durant-Whyte).

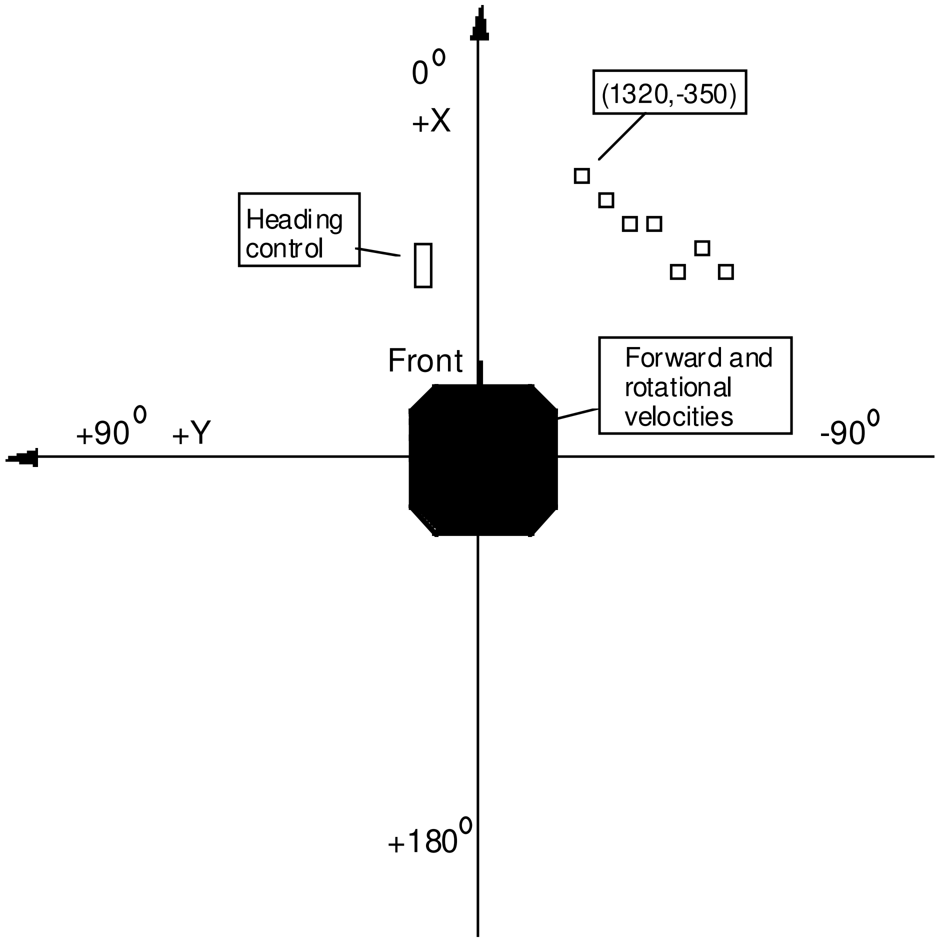

Figure 6-1. Saphira's LPS coordinate system

The side-pointing sonars are somewhat useful for obstacle avoidance, because they signal when it isn't useful to turn to one side or the other. But their main purpose is to delineate features for the recognition algorithms. They are good for this purpose because the robot is often moving parallel to wall surfaces, and by accumulating the side sonar readings, it's possible to pick out a nice straight feature.

The buffers differ slightly in how they accumulate sonar readings, based on the difference in their utility. They are all circular buffers, that is, a new reading replaces the oldest one. The front buffer, *sraw_buf*, accumulates one reading each time a sonar is fired, regardless of whether it sees anything. If nothing is found, the *valid* flag at that buffer position is set to 0; otherwise, it is set to 1, and the *xbuf* and *ybuf* slots are set to the position of the sonar reading, in the robot's local coordinate system. This strategy guarantees that the front buffer can be cleared out after a short amount of time when nothing is in the robot's way. For example, if the robot is getting 20 front sonar readings a second, and the front buffer is 30 elements long, it will be completely clear in 1.5 seconds if there is nothing in front of the robot.

The two side buffers, *sr_buf* and *sl_buf*, accumulate sonar readings only when a side sonar actually sees a surface; hence, their *valid* flag is always set. Thus, readings stay in the side buffers for longer periods of time, and Saphira has a chance to figure out what the features are.

As the robot moves, all the entries in the circular buffers are updated to reflect the robot's motion, i.e., the sonar readings stay *registered* with respect to the robot's movements.

```
#define CBUF_LEN 200
typedef struct          /* Circular buffers. */
{
  int start;            /* internal buffer pointer */
  int end;              /* internal buffer pointer */
  int limit;            /* current buffer size */
  float xbuf[CBUF_LEN];
  float ybuf[CBUF_LEN];
  int valid[CBUF_LEN];  /* set to 1 for valid entry */
} cbuf;

cbuf *sraw_buf, *sr_buf, *sl_buf;
```

```
void sfSetFrontBuffer (int n)
void sfSetSideBuffer (int n)
float sfFrontMaxRange
```

The first two functions, when given an argument greater than zero, set the front and side buffer limits to that argument, respectively. If given an argument of zero, they clear their buffers, that is, set the *valid* flags to 0. These buffer limits can also be set from the parameter file, and are initialized for a particular robot on connection.

sfFrontMaxRange is the maximum range at which a front sonar reading is considered valid. It is initially set to 2500 (2.5 meters). Setting this range higher will make the obstacle avoidance routines more sensitive and subject to false readings; setting it lower will make them less sensitive.

6.6.2. Occupancy functions

The following functions look at the raw sonar readings to determine if there is an obstacle near the robot. Other Saphira interpretation processes use the sonar readings to extract line segments representing walls and corridor.

Saphira has several functions for testing whether sonar readings exist in areas around the robot. The different functions are useful in different types of obstacle detection routines; for example, when avoiding obstacles in front of the robot, it's often useful to disregard readings taken from the side sonars.

The detection functions come in two basic flavors: "box" functions and "plane" functions. Box functions look at a rectangular region in the vicinity of the robot, while plane functions look at some portion of a half-plane.

```
int sfOccBox (int xy,  int cx, int cy, int h, int w)
int sfOccBoxRet (int xy, int cx, int cy, int h, int w, int *x, int *y)
```

When using these functions, it helps to keep in mind the coordinate system of the LPS. They look at a rectangle centered on *cy, cy* with height *h* and width *w*. **sfOccBox** returns the distance in millimeters to the nearest point to the center of the robot in the X direction (**xy = sfFRONT**) or Y direction (**xy = sfSIDES**). The returned value will always be a positive number, even when looking on the right side of the robot (negative Y values). If there is no sonar reading in the rectangle, it returns 5,000 (5 meters).

For example, in the case of an LPS shown in Figure 6-2, **sfOccBox(sfSIDES,1000,600,900,800,1)** returns 300; **sfOccBox(sfFRONT, 1000,-600,900,600,0)** returns 600.

sfOccBoxRet returns the same result as **sfOccBox,** but also sets the arguments ***x** and ***y** to the closest reading in the rectangle, if one exists.

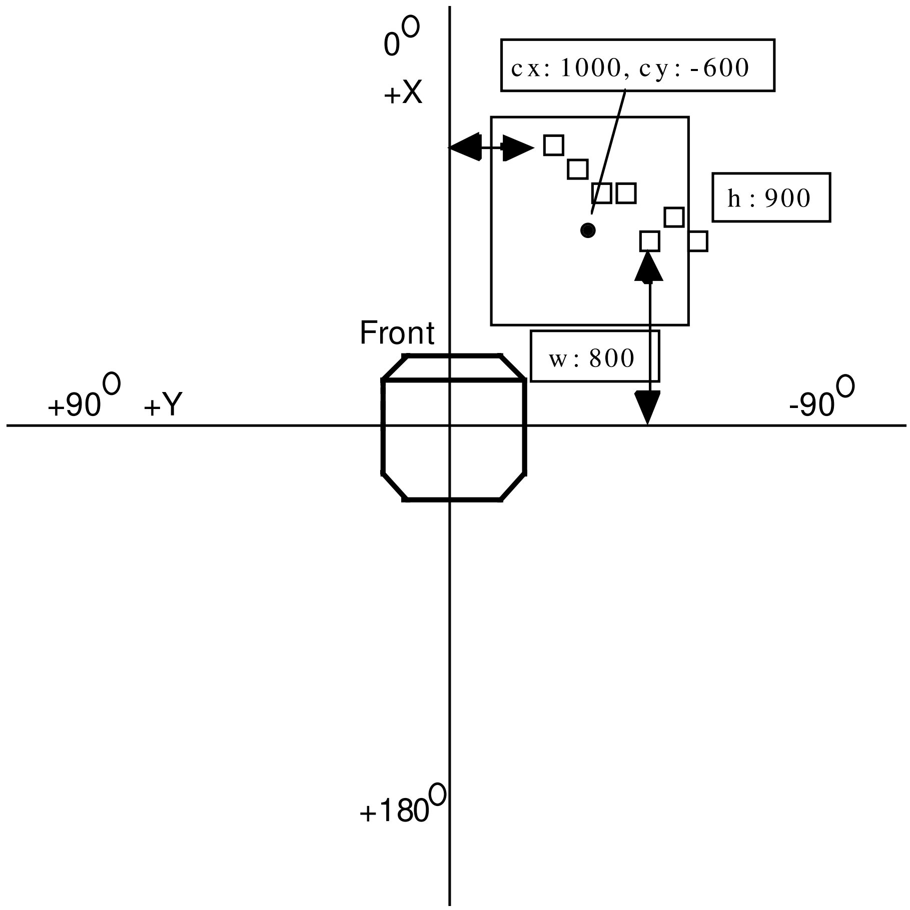

Figure 6-2. Sensitivity rectangle for the *sfOccBox* functions

```
int sfOccPlane (int xy, int source, int d, int s1, int s2)
int sfOccPlaneRet (int xy, int source, int d, int s1, int s2, int *x, int
*y)
```

The plane functions are slightly different. Instead of looking at a centered rectangle, they consider an infinite rectangle defined by three sides: a line perpendicular to the direction in question, and two side boundaries.

Figure 6-3 shows the relevant areas for *sfOccPlane(sfFRONT,sfFRONT,600,400,1200)*. The first parameter indicates positive X direction for the placement of the rectangle. The second parameter indicates the source of the sonar information: the front sonar buffer (*sfFRONT*), the side sonar buffer (*sfSIDES*), or both (*sfALL*).

The rectangle is formed in the positive X direction, with the line X = 600 forming the bottom of the rectangle. The left side is at Y = 400, the right at Y = -1200. The nearest sonar reading within these bounds is at an X distance of 650, and that is returned.

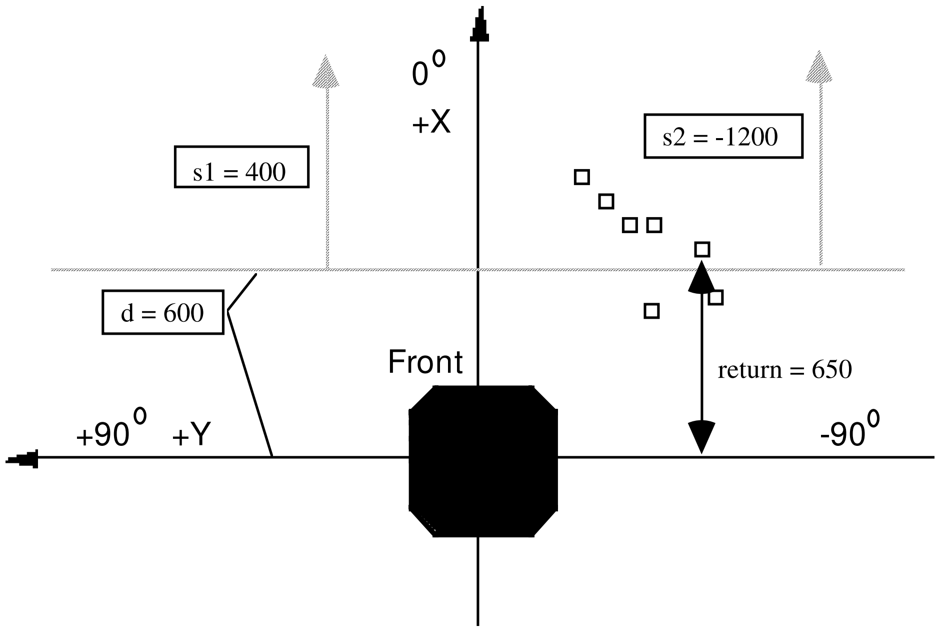

Figure 6-3 Sensitivity rectangle for *sfOccPlane* functions

Note that the baseline of **`sfOccPlane`** is always a positive number. To look to the rear, use an **`xy`** argument of **`sfBACK`**; left side is **`xy = sfLEFT`**, right side is **`xy = sfRIGHT`**.

As with **`sfOccBox`**, a value of 5000 is returned if there is no sonar reading. And, to return the coordinates of the nearest point in the rectangle, use the **`sfOccPlaneRet`** funtion.

6.6.3. Artifacts

Through Saphira, you can place a variety of artificial constructs within the geometry of the LPS and have them registered automatically with respect to the robot's movement. Generally, these *artifacts* are the result of sensor interpretation routines and represent points and surfaces in the real world. But they can also be purely imaginary objects: for example, a goal point to achieve or the middle of a corridor.

Artifacts, like the robot, exist in both the LPS and the global map space. Their robot-relative coordinates in the LPS (*x, y, th*) can be used to guide the robot locally, e.g., to face towards a goal point. Their global coordinates (*ax, ay, ath*) represent position and orientation in the global space. As the robot moves, Saphira continuously updates the LPS coordinates of all artifacts, to keep them in their relative positions with respect to the robot. The global positions of artifacts doesn't change, of course. But the dead-reckoning used to update the robot's global position as it moves contains errors, and the robot's global position gradually decays in accuracy. To bring it back into alignment with stationary artifacts, *registration routines* use sensor information to align the robot with recognized objects. These functions are described in a subsequent section.

You may add and delete artifacts in the LPS. There are two types of artifacts that users add. *Map artifacts* are permanent artifacts representing walls, doorways, and so on in the office environment. *Goal artifacts* are temporary artifacts placed in the LPS when a behavior is invoked. The artifact functions as an input to the behavior: for example, there is a behavior to reach a goal position, and the goal is represented as a point artifact in the LPS. These artifacts are usually deleted when the behavior is completed.

The system also maintains artifacts of different types. There is an artifact representing the origin of the global coordinate system. There are various *hypothesis artifacts* representing hypothesized objects extracted by the perceptual routines, and used by the registration routines.

6.6.3.1. Points and Lines

All artifacts are defined as C structures. Each has a *type* and a *category*. The type defines what the artifact represents; the simplest artifacts are points and lines, while corridors are a more complex type. You may define your own artifact types.

The category of an artifact relates to its use by the LPS. Currently, Saphira supports three categories: *system* for artifacts with an internal function, *percept* for artifacts representing hypothesized objects extracted from sensor input, and *artifact* for user-created artifacts such as map information and goal artifacts..

```
typedef enum
{
 SYSTEM, PERCEPT, ARTIFACT
} cat_type;

typedef enum
{
 INVALID, POS, WALL, CORRIDOR, LANE, DOOR, JUNCTION, OFFICE, BREAK, OBJECT
} pt_type;
```

The *point* type consists of a directed point (position and direction), with an identifier, a type, a category, and other parameters used by the system. X, Y coordinates are in millimeters, and direction is in radians from π to $-\pi$. The type *POS* is used for goal positions in behaviors. Other types may add additional fields to the basic *point* type, e.g., length and width for corridors.

```
typedef struct
{
  float x, y, th;              /* x,y,th position of point relative to robot
*/
  pt_type type;                /* type of point */
  cat_type cat;                /* category */
  boolean snew;                /* whether we just found it */
  boolean viewable;      /* whether it's valid */
  int id;
  float ax, ay, ath;     /* global coords */
  unsigned int matched;  /* last time we matched */
  unsigned int announced;      /* last time we announced */
} point;
```

The orientation of a point is useful when defining various behaviors. For example, a doorway is represented by a point at its center, a width, and a direction indicating which way is into the corridor.

```
point *sfCreateLocalPoint (float x, float y, float th)
point *sfCreateGlobalPoint (float x, float y, float th)
void   sfSetLocalCoords (point *p)
void   sfSetGlobalCoords (point *p)
```

The first two functions create new *ARTIFACT* points of type *POS,* based on the supplied coordinates. For example, *sfCreateLocalPoint(1000.0, 0.0, 0.0)*creates a point 1 meter in front of the robot. Very useful for behavior goal positions.

The second two functions reset the local or global coordinates from the other set, based on the robots current position. These functions are useful after making a change in one set of coordinates.

In order to keep a point's local coordinates updated within the LPS, it must be added to the *pointlist* after it is created. The pointlist is a list of artifacts that Saphira updates when the robot moves. The following functions add and delete members of the pointlist:

```
void sfAddPoint (point *p)
void sfAddPointCheck (point *p)
void sfRemPoint (point *p)
point *sfFindArtifact (int id)
```

Normally, to add a point to the pointlist, use *sfAddPointCheck,* which first checks to make sure point *p* is not in the list already before adding it. It is not a good idea to have two copies of a pointer to a point in the pointlist, because its position will get updated twice. The *sfRemPoint* function removes a point from the list, of course. Finally, *sfFindArtifact* returns the artifact on *pointlist* with identifier *id*, if it exists; else it returns *NULL.*

```
point *sfGlobalOrigin
point *sfRobotOrigin
```

These are *SYSTEM* points representing the global origin (0,0,0) and the robot's current position.

```
float sfNormAngle(float ang)
float sfNorm2Angle(float ang)
float sfNorm3Angle(float ang)
float sfAddAngle(float a1, float a2)
float sfSubAngle(float a1, float a2)
float sfAdd2Angle(float a1, float a2)
float sfSub2Angle(float a1, float a2)
```

These functions compute angles in the LPS. Normally, angles in the LPS are represented in radians using floating-point numbers. Artifact angles are always normalized to the interval $[0,2\pi]$.

sfNormAngle will put its argument into this range. The corresponding functions **sfAddAngle** and **sfSubAngle** also normalize their results this way.

It is often convenient to give headings in terms of positive (counterclockwise) and negative (clockwise) angles. The second normalization function, sfNorm2Angle, converts its argument to the range $[-\pi,+\pi]$, so that the discontinuity in angle is directly behind the robot. The corresponding functions **sfAdd2Angle** and **sfSub2Angle** also normalize their results this way.

Finally, it is sometimes useful to reflect all angles into the upper half-plane $[-\pi/2,\pi/2]$. The function **sfNorm3Angle** will do this to its argument, by reflecting any angles in the lower half-plane around the X-axis, e.g., +100 degrees is reflected to +80 degrees.

```
float sfPointPhi (point *p)
float sfPointDist (point *p)
float sfPointNormalDist (point *p)
float sfPointDistPoint(point *p1, point *p2)
float sfPointNormalDistPoint (point *p, point *q)
void  sfPointBaricenter (point *p1, point *p2, point *p3)
```

The first three functions compute properties of points relative to the robot. The *sfPointPhi* function returns the angle of the vector between the robot and the point *p*, in radians from π to -π. *sfPointDist* returns the distance from the point to the robot. *sfPointNormalDist* returns the distance from the robot to the line represented by the artifact point.

The second three functions compute properties of points. *sfPointDistPoint* returns the distance between its arguments. *sfPointNormalDistPoint* returns the distance from point *q* to the line represented by artifact point *p*. *sfPointBaricenter* sets point *p3* to be the point midway between point *p1* and *p2*.

```
void  sfChangeVP (point *p1, point *p2, point *p3)
void  sfUnchangeVP (point *p1, point *p2, point *p3);
float sfPointXo (point *p)
float sfPointYo (point *p)
```

```
float sfPointXoPoint (point *p, point *q)
float sfPointYoPoint (point *p, point *q)
void  sfPointMove (point *p1, float dx, float dy, point *p2)
void  sfMoveRobot (float dx, float dy, float dth)
```

These functions transform between coordinate systems. Since each point artifact represents a coordinate system, it is often convenient to know the coordinates of one point in another's system. All functions that transform points operate on the *local* coordinates; if you want to update the global coordinates as well, use **sfSetGlobalCoords**.

sfChangeVP takes a point **p2** defined in the LPS, and sets the local coordinates of **p3** to be **p2**'s position in the coordinate system of **p1**. **sfUnchangeVP** does the inverse, that is, takes a point **p2** defined in the coordinate system of **p1**, and sets the local coordinates of **p3** to be **p2**'s position in the LPS.

In some behaviors it's useful to know the robot's position in the coordinate system of a point. **sfPointXo** and **sfPointYo** give the robot's *x* and *y* coordinates relative to their argument's coordinate system. **sfPointXoPoint** and **sfPointYoPoint** do the same for an arbitrary point **q**. **sfPointMove** sets *p2* to the coordinates of **p1** moved a distance **dx** and **dy** in its own coordinate system.

sfMoveRobot moves the robot in the global coordinate system by the given amount. This is a trickier operation than one might suspect, because the *local* coordinates of all artifacts must be updated to keep them in proper correspondence with the robot. Note that the values **dx** and **dy** are in the robot's coordinate system, e.g., **sfMoveRobot(1000, 0, 0)** moves the robot forward 1 meter in the global coordinate system.

Line artifacts are called *walls*. A wall consists of a straight line segment defined by its directed centerpoint, plus length. Any linear surface feature may be modeled using the wall structure. The only type currently defined is **WALL**.

Like points, walls may be added or removed from the pointlist so that Saphira registers them in the LPS with the robot's movements. Cast each to type *point* before manipulating them with the pointlist functions described above.

Drawing artifacts on the LPS display screen is useful for debugging behaviors and interpretation routines. Saphira currently draws most types of artifacts if their *viewable* slot is greater than 0.

6.7. Sensor Interpretation

Besides the occupancy functions, the Saphira library includes functions for analyzing a sequence of sonar readings and constructing artifacts that correspond to objects in the robot's environment. We are gradually making these internal functions available to users, as we work on tutorial materials illustrating their utility. Currently, the only interpretation routines are for wall hypotheses.

```
wall sfLeftWallHyp
wall sfRightWallHyp
```

These wall structures contain the current wall hypothesis on the left and right sides of the robot, using the side sonar buffers. If a wall structure is found, then the **viewable** flag is set non-zero in the structure, and the wall dimensions are updated to reflect the sensor readings. In order for wall hypotheses to be found, the wall-finding routines must be invoked with **sfInitInterpretationProcs**.

6.8. Drawing and Color Functions

Use the following commands function to display custom lines and rectangles on the screen and to control the screen colors. All arguments are in millimeters in the LPS coordinate system.

```
void set_vector_buffer (int w)
void sfSetLineWidth (int w)
void sfSetLineType (int w)
void sfSetLineColor (int color)
```

```
void sfSetPatchColor (int color)
void sfDrawCenteredRect (float x, float y, float w, float h)
void sfDrawRect (float x,float y,float dx,float dy)
```

The function **set_vector_buffer** switches between drawing thick (argument 1) and thin (argument 0) lines. To draw a rectangle, use the function **sfDrawCenteredRect** or **sfDrawRect**. The centered version takes a center point of the rectangle, and a width and height. The other version takes the lower-left corner position, a width, and a height.

By default, sonar points display in a light blue and artifacts are in a dull red. Saphira's graphics routines now use a state machine model, in which color, line thickness, and other graphics properties are set by a function, and remain for all subsequent graphics calls until they are set to new values. Note that you cannot depend on the state of the graphics context when you make a graphics call, and should set it appropriately.

For lines, set the width w to the desired pixel width. You may select one of two line types: Set the w function parameter to 0 for a solid line, and 1 for a dashed line. The patch and line colors accept a color value as shown in Table 6-4.

Table 6-4. Saphira colors

Color Reference	Value
sfColorYellow	0
sfColorLightYellow	3
sfColorRed	5
sfColorLightRed	8
sfColorDarkTurquoise	10
sfColorDarkOliveGreen	11
sfColorOrangeRed	12
sfColorMagenta	13
sfColorSteelBlue	14
sfColorBrickRed	15
sfColorBlack	100
sfColorWhite	101

6.9. Maps and Registration

Saphira has a set of routines for creating and using global maps of an indoor environment. This facility is still under construction; this section gives an overview of current capabilities and some of the functions a client program can access.

A *map* is a collection of artifacts with global position information. Typically, a map will consist of corridors, doors, and walls—all artifacts of the offices where the robot is situated. Maps may be loaded and deleted using the interface Files menu, or by function calls.

A map can either be created by the robot as it wanders around the environment, or you may create one as a file.

6.9.1. Map File Format

A map file contains optional comments, designated with a semi-colon (;) prefix, and lines specifying artifacts in the map. For example:

```
;;
;; Map of a small portion of the SRI AIC
;;

CORRIDOR (1)   0, 0, 0, 10000, 2000
CORRIDOR (2)   0, 0, 90, 10000, 2000
DOOR (3)       3000, 1000, 90, 1000
WALL           1000, 1000, 0, 3000
```

The *CORRIDOR* lines define a series of corridor artifacts. The number in parentheses is the artifact id, and it must be a positive integer. The first three coordinates are the X, Y, and θ position of the center of the corridor in millimeters and degrees. The fourth coordinate is the length of the corridor, and the fifth is the width.

DOOR entries are defined in much the same way, except that the third coordinate is the direction of the normal of the door, which is useful for going in an out. The fourth coordinate is the width of the door.

The *WALL* entry does not have an id. The first two coordinates are the X, Y position of the center of the wall; the third is the direction of the wall, and the fourth is its length. Wall segments are used where a corridor is not appropriate: the walls of rooms or for large open areas, for example.

```
int sfLoadMapFile (char *name) (UNIX, MS Windows)
int sfLoadMapFile(char *name, int vref)(Macs)
```

The *sfLoadMapFile* function loads a map file *name* into Saphira and destroys any other map artifacts. Returns 0 if successful; -1 if not.

6.9.2. Map Registration and Creation

As the robot moves, its dead-reckoned position will accumulate errors. To eliminate these errors, a registration routine attempts to match linear segments and door openings against its map artifacts. This lets you align the robot's global position with the global map. The process that performs registration is called *test matching*. In the sample Saphira client, this process is invoked by the function *sfInitRegistrationProcs.* To disable registration, either do not start the *test matching* process, or set its state to *sfSUSPEND* or *sfINTERRUPT,* using *sfFindProcess* and *sfSetProcessState*.

A by-product of the registration process is that sometimes a corridor or doorway is found that does not match any map artifact. In this case, Saphira will, by default, create a new artifact and add it to the map. To turn off this feature, set the variable *add_new_features* to FALSE.

In finding corridors, Saphira by default attempts to align them on 90 degree angles, which is typical for office environments. To turn off this feature, set the variable *snap_to_right_angle_grid* to FALSE.

7. Saphira Behaviors

Controlling the movement of the robot server is a difficult job. At the lowest level, proportional-integral-differential (PID) control can make the wheels turn to move the robot at a fixed speed in a desired direction. But there is a lot more to robot motion than this: The trajectory of the robot must satisfy conflicting demands from the task and various maintenance policies. For example, in navigating from one room to another in an office environment, the trajectory in large part is defined by goal positions at corridor intersections. The robot should achieve these positions as quickly as possible, subject to safety and power considerations. On a more local scale, the robot should avoid obstacles and respond to contingencies such as closed doors or blocked corridors.

7.1. Behaviors and Fuzzy Control

Saphira implements behaviors as sets of fuzzy control rules which map states of the LPS into control actions for the robot. A tutorial on Saphira's fuzzy control system can be found in the Saphira documentation. Please refer to it for explanations of the concepts referred to here.

7.2. Behavior Grammar

The *behavior grammar* defines a convenient syntax for defining behaviors. The BNF for the grammar is given below. For reference, here is an example of a typical behavior using this syntax. This behavior sends the robot towards a goal position.

```
BeginBehavior myGoto     /* behavior name */
             Params
                         sfPTR goal_pt       /* pointer to goal point */
                         sfFLOAT radius      /* how close we come, in mm */
             Rules
                         If too_left Then Turn Right
                         If too_right Then Turn Left
                         If Not (near_goal Or too_left Or too_right) Then
Speed 200.0
                         If near_goal Or too_left Or too_right Then Speed 0.0
             Update
                         float phi = sfPointPhi(goal_pt);
                         float dist = sfPointDist(goal_pt);
                         too_left = up_straight(phi, 0.1, 0.6);
                         too_right = straight_down(phi, -0.6, -0.1);
                         near_goal = straight_down(dist, radius, radius*2);
             Activity
                         Turn Not near_goal
                         Speed Not near_goal
                         Goal near_goal
EndBehavior
```

7.3. Behavior Grammar in BNF

Here are the complete rules for the behavior grammar, in the form accepted by the YACC or BISON parsers.

```
/*  Behavior definition: name params rules init update activity */

BEHAVIOR:=
     "BeginBehavior"     symbol
     "Params"            [PARAM_STMTS]
     "Rules"             [RULE_STMTS]
    ["Init"              C_STMTS]
     "Update"            [C_STMTS]
     "Activity"          [ACT_STMTS]
     "EndBehavior"

/* behavior parameters */
PARAM_STMTS:=
       {"sfINT" | "sfFLOAT" | "sfPTR"} symbol [PARAM_STMTS]

/* Rule definition: name fuzzy-var action mod */
RULE_STMTS:=
      [SYMBOL] "If" FUZZY_EXP "Then" CONTROL [RULE_STMTS]

/* fuzzy expression */
FUZZY_EXP:=
       symbol | float
      | "Not" FUZZY_EXP
      | FUZZY_EXP "And" FUZZY_EXP
      | FUZZY_EXP "Or"  FUZZY_EXP
      | "(" FUZZY_EXP ")"

/* rule actions and modifiers */
CONTROL:=
       "Turn Left"  [MOD] | "Turn Right" [MOD]
      | "Turn" symbol [MOD]| "Speed" MVAL

MOD:=
       "Very Slowly" | "Slowly" | "Moderately" | "Sharply"
      | "Very Sharply"| symbol

MVAL:=
       symbol | int | float

/* activity statements */
ACT_STMTS:=
      {"Turn" | "Speed" | "Goal" | "Progress"} FUZZY_EXP
         [ACT_STMTS]
```

7.4. Behavior Executive

Before any behaviors can be invoked and run, the behavior executive must be started. Normally this is done using the **sfInitControlProcs** call.

Behaviors and direct motion control will conflict if a client attempts to use both at the same time to control the robot. For example, in the **saphira** sample client, the bump-and-go procedure uses direct motion control, while the obstacle avoidance routines are behaviors. The bump-and-go procedure is inactive until the robot hits something, at which point it takes over motion control and backs the robot up. To suppress behavior execution during this time, the sfBehaviorControl flag is set to 0. When bump-and-go is finished, it resets the flag to 1, and the behaviors resume control.

int sfBehaviorControl

A value of 0 suppress behavior control of motion, although all behaviors are still evaluated. A value of 1 allows the results of behavior evaluation to control the robot motion.

7.5. *Fuzzy variables.*

Fuzzy variables are floating-point numbers in the range [0,1]. Several functions are defined for creating fuzzy variables from single numeric values.

7.5.1. Fuzzy variable creation functions

```
float straight_up (float x, float min, float max)
float down_straight (float x, float min, float max)
float f_greater (float x, float c, float delta)
float f_smaller (float x, float c, float delta)
float f_eq (float x, float c, float delta)
```

The functions *straight_up* and *down_straight* convert numerical values into a fuzzy value based on its inclusion in a range. Both take three arguments: the value itself, the start of the range, and the end of the range. *straight_up* returns 0.0 if the value is below the range, and 1.0 if it is above, and interpolates linearly between them (Figure 7-1). *down_straight* is the opposite: values below the start return 1.0, above 0.0, and intermediate ones are linearly interpolated.

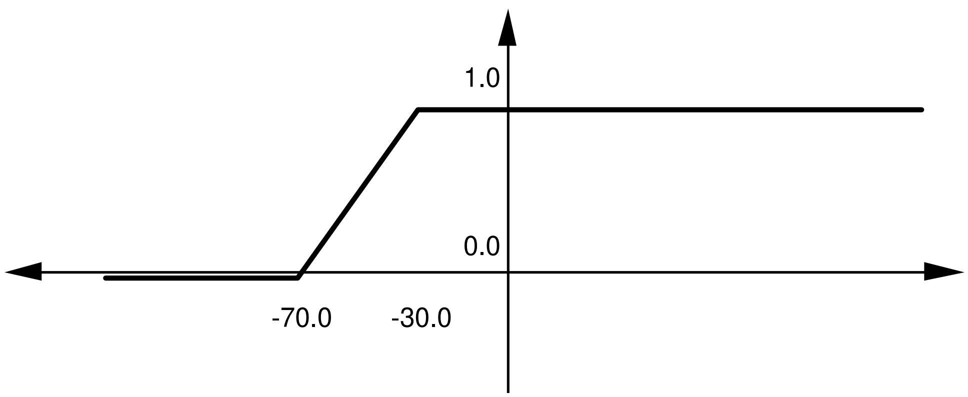

Figure 7-1. The *straight-up* function

The functions *f_smaller*, *f_greater*, and *f_eq* compare two numbers and return a fuzzy value based on whether the first is smaller than, larger than, or equal to the second. The *delta* argument is the range over which the fuzzy value will vary.

7.5.2. Fuzzy variable combination functions

Combine fuzzy variables by using the T-norm functions *max* (for disjunction), *min* (for conjunction), and *unary minus* (for negation). The utility functions *f_and, f_or,* and *f_not* are provided to implement these operators.

```
float f_not (float x)
float f_and (float x, float y)
float f_or (float x, float y)
```

7.6. *Implementing Behaviors*

For reference, we include descriptions of the parts of behaviors defined using structures and functions in C. If you use the behavior syntax to write behaviors, you generally won't have to worry about these details.

7.6.1. Input parameters

The variables that constitute the input to the behavior are contained in a structure called *beh_params*. Each parameter is either a floating point number or a pointer; pointers are used for complex variables such as goal points. The *beh_params* type is an array of such parameters.

```
typedef union /* a param can be either a fp number */
{             /* or a pointer */
 float f;
 void * p;
} param;

typedef param * beh_params;
```

7.6.2. Update function

On each Saphira cycle (100 ms), the behavior updates its state variables (using information from the LPS), and then evaluates its rules. Updating is accomplished by an update function, which takes the *beh_param* structure as an argument.

7.6.3. Init function

When Saphira instantiates a behavior schema, its *init* function is called to set up the initial fuzzy state. The input to the init function is a *beh_params* structure, containing the initial parameters of the behavior. The init function can set any initial state that is needed by the behavior; a clock, for example, if the behavior has a timeout.

7.6.4. Rules

Each behavior rule is defined as a structure *beh_rule*, which consists of a name, and two indices into the fuzzy state: the antecedent value for the rule, and the mean value of the output action. Each rule can recommend only one action, which is the *consequent* value: one of *Accel, Decel, Turn_left,* or *Turn_right:*

```
typedef struct
{
 char *name;            /* name of the rule */
 int *antecedent,             /* activity of this rule */
   *consequent,               /* action to take */
   *parameter;                /* mean value of action */
} beh_rule;
```

For example rule definitions, see below. Note that the consequent value constants are external integers, but they are not declared in the Saphira headers, so they must be declared in the application code.

7.6.5. Behavior schema

A complete behavior schema is a structure combining its rules, init, and update functions. The rules can be included directly in the definition; here is the example constant velocity function:

```
extern int Accel, Decel, Turn_left, Turn_right;

behavior
constant_velocity =
              { "Constant Vel", cv_setup, cv_check_speed, 1,
                          2, { { "Speed-Up", &cv_too_slow ,
                                        &Accel, &cv_speedup},
                                 { "Slow-Down", &cv_too_fast,
                                        &Decel, &cv_slowdown}
                             }
              };
```

The first argument is the name of the behavior; the second is the init function; the third is the update function; and the fourth argument is the number of parameters. The number of rules is the fifth argument, and the rules themselves are the sixth. Note that all global variables are referenced as pointers in the behavior.

The maximum number of rules in a behavior is 10. The consequent values *Accel* and so on must be declared as external integers.

7.7. Invoking Behaviors

Behaviors are invoked by using the *sfInitBehavior* function, which creates an instance (or closure) binding the behavior schema to a set of parameters. The function then adds the behavior to the list of executing behaviors. Currently, there is a limit of 20 executing behaviors.

```
BEHCLOSURE sfInitBehavior (behavior *b, int priority, int running, ...)
BEHCLOSURE sfInitBehaviorDup (behavior *b, int priority, int running,
...)
```

The first argument of the *sfInitBehavior* function is a pointer to the behavior structure, as defined above. The second is the priority of the behavior closure, relative to others. Lower values get higher priority: 0 is the highest priority and should be used for the most important emergency maneuvers, such as collision avoidance. Saphira treats all behaviors with the same priority equally in terms of competing for control of the robot; ones with larger priority numbers (lower priority) are suppressed by activity of higher-priority behaviors.

The third argument is *TRUE* (or *sfON*) if the closure is initially running, and *FALSE* (or *sfOFF*) if it is not active. A closure that is not running is still in the list of behavior closures, but it is not executed and does not affect the robot's movements. The running state of a behavior can be changed with the *sfSetBehaviorState* function (see below).

The remaining arguments to this function set up the parameters of the closure. The arguments come in pairs, with the type of the argument first, and then its value (Table 7-1). The argument list must end with the constant *sfEND*, even if there are no other arguments.

Here is an example invocation of the pre-defined behavior *sfKeepOff*:

```
sfInitBehavior(sfKeepOff, 1, True,
               sfFLOAT, 100.0, /* caution speed */
               sfFLOAT, 0.4, /* sensitivity */
               sfEND);
```

Table 7-1. Valid Saphira behavior arguments

Argument Name	Argument Type
sfINT	*int*
sfFLOAT	*float*
sfPTR	*void **
sfEND	end of argument list

Normally, only on behavior instance for a given behavior is allowed to run. *sfInitBehavior* check for any currently-running instances, and if one is present, just replaces its parameters. There are cases where you may want more than one instance with different parameters. The *sfInitBehaviorDup* function is similar to *sfInitBehavior*, but does not check for duplicate behavior instantiations.

`void sfKillBehavior (BEHCLOSURE b)`

Removes the behavior instance *b* from the list of currently executing behaviors. If you want to halt the execution of a behavior for some period, then resume it, use the state-changing function below, rather than removing and re-invoking the behavior.

```
void sfSetBehaviorStat e(BEHCLOSURE b, int state)
void sfBehaviorOn(BEHCLOSURE b)
void sfBehaviorOff(BEHCLOSURE b)
```

sfSetBehaviorState sets the state of a behavior closure to one of the *state* arguments defined in Table 7-2. The other two functions are convenience functions for particular states.

Table 7-2. Valid behavior closure state values

State argument	Action
sfON	Behavior starts running.
sfOFF	Behavior stops running.
sfTOGGLE	Behavior's run state is toggled.

`BEHCLOSURE sfFindBehavior (char *name)`

Finds a behavior in the current closure list with name *name*, and returns a pointer to it, if it exists. If not, returns NULL.

7.8. Pre-Defined Saphira Behaviors

Saphira has a number of pre-defined behaviors for obstacle avoidance and goal-directed movement. Most of the complexity of these behaviors is in the update functions, which extract data from the LPS and update a small set of fuzzy variables relevant to the behavior. Besides integrating these behaviors with your own routines, you can use them as templates to create new behaviors. The example code is **`behavior.beh`** in your Saphira distribution software.

Note that the variables in the example are *pointers* to behavior structures and can be used directly in the *sfInitBehavior* function. See the sample application **`saphira.beh and behavior.beh`** for examples.

`behavior *sfConstantVelocity`

Sets the velocity setpoint on the robot server to its first parameter, an integer in millimeters per second.

`behavior *sfStop`

Sets the velocity setpoint to zero. No parameters.

`behavior *sfAvoidCollision`

Slows and turns the robot sharply to avoid immediate obstacles. Takes four parameters, listed in the Table 7-3. Additionally, the default turn direction when it is completely blocked is given by the global variable *sfPreferredTurnDir*, which should be set to either *sfLEFTTURN* or *sfRIGHTTURN*. User programs and other behaviors can set this variable to change the action of this behavior.

Table 7-3. Avoid collision behavior parameters

Parameter	Effect
sfFLOAT	Front sensitivity to obstacles. Value from 0.5 (not sensitive) to 3.0 (very sensitive).
sfFLOAT	Side sensitivity to obstacles. Value from 0.5 (not sensitive) to 3.0 (very sensitive).
sfFLOAT	Turning gain: controls how rapidly the robot turns away from obstacles. Value from 4.0 (slow turn) to 10.0 (fast turn).
sfFLOAT	Standoff. Defines the avoidance "bubble" around the robot. Value from *flakey_radius* (at the robot) to *flakey_radius* + *standoff* (*standoff* mm from the robot).
sfPreferredTurnDir	This global variable controls the default direction of turn when the front is blocked. Values are *sfLEFTTURN* or *sfRIGHTTURN*.

`behavior *sfStopCollision`

Slows the robot sharply to avoid immediate obstacles. This behavior differs from *sfAvoidCollision* in that it doesn't turn the robot; some other behavior must do that. Takes three parameters, listed in the Table 7-4.

Table 7-4. Stop collision behavior parameters

Parameter	Effect
sfFLOAT	Front sensitivity to obstacles. Value from 0.5 (not sensitive) to 3.0 (very sensitive).
sfFLOAT	Side sensitivity to obstacles. Value from 0.5 (not sensitive) to 3.0 (very sensitive).
sfFLOAT	Standoff. Defines the avoidance "bubble" around the robot. Value from *flakey_radius* (at the robot) to *flakey_radius* + *standoff* (*standoff* mm from the robot).

`behavior *sfKeepOff`

Gently steers the robot around and away from distant objects. The behavior takes two parameters and uses the global variable *sfPreferredTurnDir*, described in the Table 7-5. The priority for *sfKeepOff* should always be less than (higher priority number) than that for *sfAvoidObstacle* when they are invoked together.

Table 7-5. Keep off behavior parameters

Parameter	Effect
sfFLOAT	Caution speed. Robot slows to this speed when more distant obstacles are detected. Value in mm/sec.
sfFLOAT	Sensitivity to obstacles. Value from 0.2 (not sensitive) to 2.0 (very sensitive).
sfPreferredTurnDir	This global variable controls the default direction of turn when the front is blocked. Values are *sfLEFTTURN* or *sfRIGHTTURN*.

`behavior *sfGoToPos`

Sends the robot to a given point. Takes three parameters, described in Table 7-6.

Table 7-6. Go to position behavior parameters

Parameter	Effect
sfFLOAT	Speed (in mm/sec). Robot moves at this speed towards goal position.
sfPTR	Goal position. Should be a pointer to a point artifact.
sfFLOAT	Success radius (in mm). Defines how close the robot must be to the goal position before the behavior goal is satisfied.

`behavior *sfAttendAtPos`

Moves the robot near a given goal position and points the robot towards the goal position. Takes three parameters, described in Table 7-7.

Table 7-7. Attend at position behavior parameters

Parameter	Effect
sfFLOAT	Speed. Robot moves at this speed towards goal position. Value in mm/sec.
sfPTR	Goal position. Should be a pointer to a point artifact.
sfFLOAT	Success radius. Defines how close the robot must be to the goal position before the behavior goal is satisfied. Value in mm.

`behavior *sfFollow`

Tells the robot to follows a lane, as represented by a lane artifact. The lane structure is a directed point with a width, although the width is ignored in this behavior since there are explicit parameters for the latitude the robot is allowed in the lane. There is also a goal point, representing a position in the lane that the robot is to achieve.

When active, the behavior draws its lane as a set of dotted lines in the LPS. This behavior takes seven parameters, described in Table 7-8. This behavior sets the *sfPreferredTurnDir* variable depending on how the robot is misaligned with the lane.

Table 7-8. Follow lane behavior parameters

Parameter	Effect
sfPTR	Lane. This is a point or lane artifact representing a line the robot is to follow. Parameters below define allowed deviations from the line.
sfPTR	Goal position. The robot moves along the lane in the direction of the goal until it reaches it. Should be a pointer to a point artifact.
sfFLOAT	Right edge (in mm). Distance the robot is allowed to wander from the right side of the line.
sfFLOAT	Left edge (in mm). Distance the robot is allowed to wander from the left side of the line.
sfFLOAT	Speed off lane (in mm/sec). How fast the robot travels when it is out of the lane.
sfFLOAT	Speed in lane (in mm/sec). How fast the robot travels when it is in the lane.
sfFLOAT	Turn ratio. How important it is to be near the center vs. aligned in the right direction; 0.0: direction overrides; 1.0: center overrides.

`behavior *sfFollowCorridor`

Tells the robot to follow a corridor, as represented by a corridor artifact. The corridor structure is a directed point with a width; the width is used to set up a lane down the center of the corridor for the robot to follow. There is also a goal point, representing a position in the lane that the robot is to achieve.

When active, the behavior draws its lane as a set of dotted lines in the LPS. This behavior takes two parameters, described in Table 7-9. This behavior sets the *sfPreferredTurnDir* variable depending on how the robot is misaligned with the corridor.

Table 7-9. Follow corridor behavior parameters

Parameter	Effect
sfPTR	Corridor. This is a corridor artifact the robot is to follow. The path of the robot is bounded by a lane set in from the sides of the corridor.
sfPTR	Goal position. The robot moves along the corridor in the direction of the goal until it reaches it. This should be a pointer to a point artifact.

`behavior *sfFollowDoor`

Tells the robot to go in a doorway, as represented by a door artifact. The direction is whether to go in or out of the doorway; this could be decided automatically by the position of the robot, but isn't because the robot may already be on the correct side.

When active, the behavior draws its lane as a set of dotted lines in the LPS. This behavior takes two parameters, described in Table 7-10. This behavior sets the *sfPreferredTurnDir* variable depending on how the robot is misaligned with the lane through the doorway.

Table 7-10. Follow door behavior parameters

Parameter	Effect
sfPTR	Door. This is a door artifact the robot is to go in or out of. The path of the robot is bounded by a narrow lane perpendicular to the door.
sfINT	Direction (*sfIN* or *sfOUT*). "In" means into the room, "out" means out of the room and into the corridor.

`behavior *sfTurnTo`

Turns the robot to point in the direction of a goal position. The robot always turns in the direction that makes the smallest turn. Parameters are given in Table 7-11.

Table 7-11. Turn to parameters

Parameter	Effect
sfPTR	Goal position. The robot turns until it points towards this goal. Should be a pointer to a point artifact.
sfFLOAT	Success angle (in radians). If the robot is within this angle of pointing towards the goal, it will have succeeded.
sfFLOAT	Turn speed. How fast the robot turns to the goal. Value of 0.5 is slow speed, 2.0 is fast.

8. Intentions and PRS-lite

Saphira includes a reactive planning system that is derived from SRI's work on the Procedural Reasoning System. We call Saphira's version PRS-lite, because it has only a subset of the functionality of the full PRS system. In compensation, it is small and fast enough that it can be used to control the robot within same 100 millisecond cycle time as the rest of the Saphira system.

The main task of PRS-lite is to coordinate behaviors and interpretation processes. In this sense, it acts like a high-level supervisor, checking the state of the LPS, deciding when to switch behaviors on and off, and which interpretation processes to invoke. So, for example, a routine such as patrolling the corridor could be written in PRS-lite. This routine would check that the robot was in a corridor, decide which way to move, invoke a corridor-following behavior, check its progress, reverse the direction of the robot when it arrived at the end of the corridor, and so on. It might even invoke contingency behaviors if the corridor is blocked or if the robot loses its way outside the corridor.

PRS-lite is not a planner, however. It does not have even a limited capability of doing means-ends analyses (PRS can, though). Saphira does have a topological planner and a plan executor (written in PRS-lite), but they are not documented in this version of the system.

8.1. Intention Schemas

PRS-lite routines are called *intention schemas*. In the full version of PRS, intention schemas achieve a goal and can be invoked to satisfy that goal. In PRS-lite, the goals are implicit, but the term *intention schema* is retained.

Intention schemas are much like Saphira processes: they are finite-state machines that are called every 100 milliseconds to update their state. They additionally incorporate other capabilities, including parameters, timeouts, invocation of sub-intentions, and success/failure reporting.

The states of all active intentions are displayed in Saphira's Intentions window, invoked from the Function menu of the Saphira interface (see Chapter 3, "The Saphira Client").

A good way to explain the workings of an intention is by example. The following intention schema *sfFindAndFollow* checks whether the robot is in a corridor, and then has it follow that corridor until it ends or until the robot is not facing along the corridor.

8.1.1. Initialize intention

```
point follow_target;
#define       CHECK_CORRIDOR    20
#define       FOLLOW_CORRIDOR   10
#define        WAIT_REMOVE      11
```

In this first part, we set up a global point that is used by the "follow corridor" behavior as a target, as well as define some convenient names for the states of the intention. Note that these states are all 10 or greater, meaning they are user defined.

8.1.2. Find a corridor

```
void                     /* for following corridors */
sfFindAndFollow(void)
{
 static process *p = NULL;
 static point *e;
 beh_params par;         /* need this because we haven't defined a
```

```
                              convenience function */
switch(process_state)
 {
 case sfINIT:
 case sfRESUME:
  add_point_check(&follow_target);
  process_state = CHECK_CORRIDOR;
  break;
```

The above code defines the initial and resumed states of the intention. We then add the target point to the pointlist. Next, we set the process state to check for a corridor:

```
 case CHECK_CORRIDOR:
  if (current_environment != NULL &&
       current_environment->type == CORRIDOR)
      {
       e = current_environment;
       process_state = FOLLOW_CORRIDOR;
       follow_target.x = 10000.0; follow_target.y = 0;
       par = beh_alloc(7);
       FOL_LANE(par) = current_environment;
       FOL_TARGET(par) = &follow_target;
       p = intend_beh(&follow_corridor,"follow it",0,par,2);
      }
  break;
```

Whenever the robot is in a corridor, the *current_environment* variable is set to an artifact representing that corridor by the registration processes. If the robot is not in a corridor or other recognizable place, this variable is NULL. The intention checks *current_environment*, and when it is not NULL it saves the environment artifact and sets up the *follow_corridor* behavior. It does this by first constructing a target point 10 meters in front of the robot (note how easy this is with robot-centered coordinates), and then by invoking the *intend_beh* function. This is a special PRS-lite function that initializes a behavior and a supervising intention at the same time. The argument 0 is a timeout condition (that is, no timeout), and the argument 2 is the priority level of the behavior. The state is thereby changed to follow a corridor.

8.1.3. Follow the corridor

```
 case FOLLOW_CORRIDOR:
  if (current_environment != e ||
       finished(p)       ||
       aligned_in_lane(e,&follow_target,80.0,110.0) < 0.5 )
      {
       process_state = WAIT_REMOVE;
       p->state = sfREMOVE;
       p = NULL;
      }
  break;
```

When the current environment changes, the behavior accomplishes its goal, or the robot is no longer aligned towards the target, the intention sets the state of the *follow_corridor* intention to *REMOVE*, and changes its own state.

```
 case WAIT_REMOVE:
  process_state = CHECK_CORRIDOR;
  break;
```

The intention waits one cycle while the *follow_corridor* intention actually gets removed, and then it goes back to waiting for a corridor environment.

8.1.4. Finished

```
  case sfINTERRUPT:      /* clean up here */
   if (p) p->state = sfREMOVE;
   process_state = sfSUSPEND;
   break;
  }
}
```

The *INTERRUPT* state is reached when the user or another intention requests an interrupt. The intention responds by getting rid of any *follow_corridor* actions and then suspending.

8.2. Intention Parameters

An *intention closure* or simply intention is the instantiation of an intention schema to a particular set of parameters. Functions which manipulate intentions use a pointer to the intention structure. If *p* is a pointer to an intention structure, then *p->params* is the *beh_param* structure holding its parameters.

8.3. Intention Schema Instantiation

An intention schema is a function with no arguments, returning *void*.

```
process *sfInitIntention (void (*fn)(void), char *name, int timeout, ...)
```

The *sfInitIntention* function instantiates and returns an intention closure, giving it the name *name*, and a timeout value of *timeout* (see below; a timeout of 0 means indefinite execution). The parameters of the intention are listed at the end of the function call in a manner similar to the *sfInitBehavior* function). The parameter list must end with *sfEND*.

8.4. Intention Termination and Removal

Like processes, intentions have a *state* that is an integer. The intention starts in the *sfINIT* state and any user-defined states must be 10 or greater. Unlike a process, an intention can terminate and remain inactive but not removed. In addition to the states defined for processes, there are special states for intentions to indicate their termination condition.

```
int finished (process *p)
```

The *finished* function returns 1 if the intention closure *p* has terminated. Otherwise, it returns 0. The termination states for intentions are *SUCCESS*, *FAILURE*, and *TIMEOUT*. (The state of an intention may be accessed with *p->state*, where *p* is a pointer to the intention closure.)

Timing-out is a way to limit the amount of time an intention has to perform its task. A timeout parameter can be specified when an intention is instantiated. It is the number of 100-millisecond cycles the intention may execute before Saphira terminates it with a *TIMEOUT* status.

If an intention schema is instantiated from within another intention, the instantiation is called a *child* intention. The field *p->dad* of the intention closure is a pointer to the parent.

An intention is removed from the list of active intentions when any one of the following three conditions exist:

1. Its state is *sfREMOVE.*
2. It has terminated (*finished(p)* is true), and it has no parent, or its parent has terminated.
3. Its parent is removed.

These rules guarantee that an intention with a termination condition of *SUCCESS*, *FAILURE*, or *TIMEOUT* can be checked by an active parent. If there is no active parent, these terminated intentions will be removed. Note that an active parent is responsible for removing its terminated children by setting their state to *sfREMOVE*.

8.5. Invoking Behaviors

There is a special facility for invoking behaviors from intentions. An intention "wrapper" is placed around the behavior, monitoring its state and adding a timeout capability. Behaviors are invoked with the function *intend_beh*.

```
process *intend_beh (behavior *b, char *name, int timeout,
                beh_params params, int priority)
```

The *intend_beh* function instantiates a "dummy" intention and a behavior, using the *params* argument for the behavior. The priority of the behavior is given by *priority*, and a timeout can be specified; a timeout of 0 is indefinite execution.

The syntax of this function is awkward, as it forces you to construct a *beh_param* structure. In future versions of Saphira, the function will take a variable number of parameter arguments, as does *sfInitIntention*.

8.6. Packet Communication Functions

Saphira contains several functions that help you manage communications between your client application and the Pioneer server directly (PSOS; see Chapter 4), rather than going through the Saphira OS. Do not use these functions to parse information packets or send motor control commands if you start up the Saphira OS with **`sfStartup`**.

```
int sfConnectToRobotint port, char *name)
```

(This Saphira function tries to open a communications channel to the robot server on port type *port* with name *name*. Returns 1 if successful; 0 if not.

Table 8-2 Port types and names for server connections

Port types	*sfLOCALPORT*	connect to simulator on the host machine
	sfTTYPORT	connect to Pioneer on a tty port
Port names	*sfCOMLOCAL*	local pipe or mailslot name
	sfCOM1	tty port 1 (/dev/ttya or /dev/cua0 for UNIX; COM1 for MSW; modem for Mac)
	sfCOM2	tty port 2 (/dev/ttyb or /dev/cua1 for UNIX, COM2 for MSW, printer for Mac)

This function also sets the global variables *sfRobotName*, *sfRobotClass*, and *sfRobotSubclass* according to the information returned from the robot; see the table below. Assuming the environment variable **`SAPHIRA`** is set correctly, it will autoload the correct parameter file from the **`params`** directory, using first the subclass if it exists, and then the class.

Table 8-3 Robot names and classes

*(char *)sfRobotName*	See robot descriptions for information on how to set the name. The simulator returns the name of the machine it is running on.
*(char *)sfRobotClass*	Robot classes are *B14*, *B21*, and *Pioneer*
*(char *)sfRobotSubclass*	Subclasses are subtypes, e.g., in Pioneer-class robots the subclass is the OS type, currently PSOS41.

```
void sfDisconnectFromRobot (void)
```

Sends the server a *close* command, then shuts down the communications channel to the server.

```
void sfResetRobotVars (void)
```

Resets the values of all internal client variables to their defaults. Should be called after a successful connection.

```
void sfRobotCom (int com)
void sfRobotComInt (int com, int arg)
void sfRobotCom2Bytes(int com, int b1, int b2)
void sfRobotComStr (int com, char *str)
void sfRobotComStrn (int com, char *str, int n)
```

These Saphira functions packetize and send a client command to the robot server. Use the command type appropriate for the type of argument. See Section 5.2 for a list and description of currently supported PSOS commands.

The string commands send stings in different formats: **`sfRobotComStr`** sends out a null-terminated string (its *str* argument), and **`sfRobotComStrn`** sends out a pascal-type string, with an initial string count; in this case *str* can contain null characters.

The function **`sfRobotCom2Bytes`** sends and integer packed from two bytes, and upper byte **`b1`**, and a lower byte **`b2`**.

```
int sfWaitClientPacket (int ms)
int sfHaveClientPacket (void)
```

Use *sfWaitClientPacket* to have Saphira listen to the client/server communication channel for up to *ms* milliseconds, waiting for an information packet to arrive from the server. If Saphira receives a packet within that time period, it returns 1 to your application. If it times out, Saphira returns 0. This function always waits at least 100 ms if no packet is present. To poll for a packet, use *sfHaveClientPacket*.

```
void sfProcessClientPacket (void)
```

When invoked by the *sfProcessClientPacket* function, Saphira waits for a client packet to be received, and then parses it into the *sfRobot* structure and sonar buffers.

```
int  sfClientBytes (void)
int  sfReadClientByte (void)
int  sfReadClientSint(void)
int  sfReadClientUsint (void)
int  sfReadClientWord (void)
char *sfReadClientString (void)
```

These functions return the contents of packets, if you want to dissect them yourself rather than using *sfProcessClientPacket*. *SfClientBytes* returns the number of bytes remaining in the current packet. The other functions return objects from the packet: bytes, small integers (2 bytes), unsigned small integers (2 bytes), words (4 bytes), and null-terminated strings.

9. Saphira Vision

Current versions of Saphira have both generic vision support and explicit support of the Fast Track Vision System (FTVS), which is available as an option for the Pioneer 1 Mobile Robot. The FTVS is a product from Newton Labs, Inc., adapted for Pioneer. The generic product name is the Cognachrome Vision System. Details about the system, manuals, and development libraries can be found at Newton Labs' Web site: **`http://www.newtonlabs.com`**.

With Saphira, the FTVS intercepts packet communication from the client to robot server, interprets some commands from the client, and sends new vision information packets back to the client. Saphira includes support for setting some parameters of the vision system, but not for training the FTVS on new objects, or for viewing the output of the camera. For this, please see the FTVS user manual about operating modes. In the future, we intend to migrate some of the training functions to the Saphira client. We also intend to have Saphira display raw and processed video.

Saphira also includes built-in support for interpreting vision packet results. If your robot has a vision system, Saphira will automatically interpret vision packets and store the results as described below.

9.1. Channel modes

The FTVS supports three channels of color information: A, B, and C. Each channel can be trained to recognize its own color space. Each channel also supports a processing mode, which determines how the video information on that channel is processed and sent to Saphira. A channel is in one of three modes:

1. BLOB_MODE 0
2. BLOB_BB_MODE 2
3. LINE_MODE 1

[Note: these definitions, as well as other camera definitions, can be found in
`handler/include/chroma.h`

To change the channel mode from a Saphira client, issue the command:

```
sfRobotComStr (VISION_COM,"pioneer_X_mode=N")
```

where the mode N is 0, 1, or 2, and the channel X is a, b, or c (small letters). On startup, the vision system channels are set to BLOB_MODE.

The processing performed in BLOB_MODE, BLOB_BB_MODE, and LINE_MODE are explained in the FTVS manual.

In line mode, several FTVS parameters affect the processing:

```
line_bottom_row          /* first row for line processing */
line_num_slices          /* how many row we process */
line_slice_size          /* how many pixels thick each row is */
line_min_mass            /* number of pixels needed to
                            determine a line segment */
```

These parameters can be set using, for example,

```
  sfRobotComStr (VISION_COM,"line_bottom_row=0")
```

9.2. Vision Packets

If the FTVS is working properly, it will send a vision packet every 100 ms to the Saphira client. In the information window, the **`VPac`** slot should read about "10", indicating that 10 packets/second are being delivered. If it reads "0", the vision system is not sending information.

Saphira parses these packets into a vision information structure:

```
   struct vinfo {
```

```
    int type;                    /* BLOB, BLOB_BB or LINE MODE */
    int x, y;                    /* center of mass */
    int area;                    /* size */
    int h, w;                    /* height and width of bounding box */
    int first, num;              /* first and number of lines */
  };
```

In BLOB_MODE, the **x, y**, and **area** slots are active. The x,y coordinates are the center of mass of the blob in image coordinates, where the center of the image is 0, 0. For the lens shipped with the FTVS, each pixel subtends approximately 1/3 degree:

```
#define DEG_TO_PIXELS 3.0       /* approximately 3 pixels per degree */
```

This constant lets a client convert from image pixel coordinates to angles. The area is the approximate size of the blob in pixels. If the area is zero, no blob was found.

In BLOB_BB_MODE, the bounding box of the blob is also returned, with h and w being the height and width of the box in pixels.

In LINE_MODE, the slots **x**, **first**, and **num** are active. The value **x** is the horizontal center of the line. **first** is the first (bottom-most) row with a line segment, and **num** is the number of consecutive rows with line segments. If no line was found, num is zero.

Global variables hold information for each channel, as follows:

```
extern struct vinfo sfVaInfo, sfVbInfo, sfVcInfo;
```

For example, to see if channel A is in BLOB_MODE, use

```
sfVaInfo.type == 0
```

9.3. Sample Vision Application

The sample Saphira client which enables the FTVS can be found as the source file **handler/src/apps/btech.c** and **/chroma.c**. The compiled executables are found in the **bin/** directory. These files define functions to put the channels into BLOB_BB_MODE, to turn the robot looking for a blob on channel A, to draw the blob on the graphics window, and to approach the blob.

void setup_vision_system(void)

Sets up parameters of the vision system, putting all channels into BLOB_BB_MODE and initializing line parameters.

int found_blob(int channel, int delta)

Returns the X-image-coordinate of a blob on channel (0=A, 1=B, 2=C), if the blob's center is within delta pixels of the center of the image. If no blob is found with these parameters, it returns -1000.

void draw_blobs(void)

Process for drawing any blobs found by the vision system. The blob is drawn as a rectangle centered at the correct angular position, and at a range where a surface two feet on a side would produce the perceived image size. The size of the rectangle is proportional to the image area of the blob.

void find_blob(void)

Intention for turning left until a blob is found in the center of the image on channel A, or until 20 seconds expires.

void search_and_go_blob(void)

Intention for finding a blob (using find_blob) on channel A, then approaching it. Uses sonars to detect when it is close to the blob.

10. Sample Parameter File.

The following is a sample parameters file used by Pioneer simulator and Saphira client to describe the physical robot and its characteristics. See also the **pioneer.p** file in the **params** folder of your Saphira software.

```
;; Parameters for the Pioneer robot
;;
;; counts/rev = 6300 (5-inch wheels)
;;
AngleConvFactor  0.0009973     ; radians per encoder count diff (2PI/6300)
DistConvFactor   0.079        ; $13F4 ticks per 5*PI (inches) distance
VelConvFactor    2.0          ; mm/sec per encoder count per 1/50 sec
RobotRadius      220.0        ; radius in mm
RobotDiagonal    90.0         ; half-height to diagonal of octagon


;; These are for seven sonars: five front, two sides
;;
;; Sonar parameters
;;              SonarNum N is number of sonars
;;              SonarUnit I X Y TH is unit I (0 to N-1) description
;;              X, Y are position of sonar in mm, TH is bearing in degrees
;;
RangeConvFactor  0.229        ; sonar range mm per 1 usec tick
;;
SonarNum 7
;;         #  x   y   th
;;-------------------------------
SonarUnit 0 120 80 90
SonarUnit 1 100 100 20
SonarUnit 2 130 40 10
SonarUnit 3 130  0  0
SonarUnit 4 130 -40 -10
SonarUnit 5 100 -100 -20
SonarUnit 6 120 -80 -90
SonarUnit 7  0  0  0
```

11. Sample World Description File

Worlds for the simulator are defined as a set of line segments using absolute or relative coordinates. Comment lines begin with a semicolon. All other nonblank lines are interpreted as directives.

The first two lines of the file describe the width and height of the world, in millimeters. The simulator won't draw lines outside these boundaries. It's usually a good idea to include a "world boundary" rectangle, as is done in the example below, to keep the robot from running outside the world.

Any entry in the world file that starts with a number is interpreted as creating a single line segment. The first two numbers are the X, Y coordinates of the beginning and the second two are the coordinates of the end of the line segment. The coordinate system for the world starts in the lower left, with +Y pointing up and +X to the right (Figure B-1).

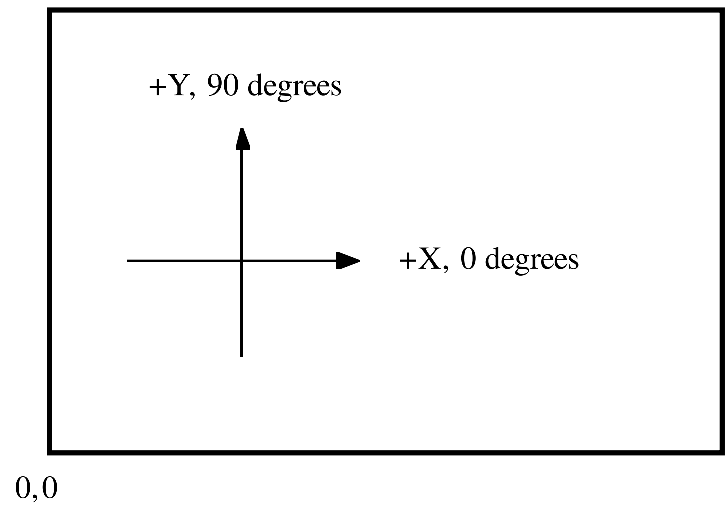

Figure B-1. Coordinate system for world definition files

The position of segments may also be made relative to an embedded coordinate system. The *push x y theta* directive in the world file causes subsequent segments to use the coordinate system with origin at x,y and whose x axis points in the direction. The *theta*. *push* directives may be nested, in which case the new coordinate system is defined with respect to the previous one. A *pop* directive reverts to the previous coordinate system.

The *position x y theta* directive positions the robot at the indicated coordinates.

The following is a fragment of the **`simple.wld`** world description file found in the **`worlds`** directory of Saphira.

```
  ;;; Fragment of a simple world

width 38000
height 30000

 0 0 0 30000                    ; World frontiers
 0 0 38000 0
 38000 30000 0 30000
 38000 30000 38000 0
```

```
push 10000 14000 0

;; upper corridor        ; length = 14,600; width = 2,000
  0 12000 3000 12000                        ; EJ 231 - J. Lee
  3900 12000 4200 12000                     ; EJ 233 - D. Moran
  5100 12000 8000 12000                     ; EJ 235 - J. Bear
  8900 12000 9200 12000                     ; EJ 237 - E. Ruspini
  10000 12000 12000 12000                   ; EJ 239 - J. Dowding
  12800 12000 14600 12000


;; Starting position

position 17500 14000 -90
```

12. Saphira API Reference

Direct Motion Control

Drawing and Color

Fuzzy Variables

Intentions

Map File

Occupancy

OS and Window Functions

Packet Functions

Processes

Processes; Predefined

Sensor Interpretation

Sonars

State Reflection

Vision

Index

Warranty & Liabilities

The developers and marketers of Saphira software shall bear no liabilities for operation and use with any robot or any accompanying software except that covered by the warranty and period. The developers and marketers shall not be held responsible for any injury to persons or property involving the Saphira software in any way. They shall bear no responsibilities or liabilities for any operation or application of the software, or for support of any of those activities. And under no circum stances will the developers, marketers, or manufacturers of Saphira take responsibility for or support any special or custom modification to the software.

Saphira Software Manual Version 5.3, January 1997