ActivMEDIA INCORPORATED

Pioneer

Mobile Robots

with Pioneer Server Operating System Software

Saphira Software Manual

Version 6.1

Saphira Manual Version 6.1f, August 1998.

Contents

Page

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 intelligent mobile robot, such as a Pioneer from *Activ*Media, 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 the 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.

The Saphira client connects to a robot server with the basic components for robotics sensing and navigation: drive motors and wheels, position encoders, and sensors. The server handles the low-level details of robot sensor and drive management, sends information, and responds to Saphira commands through a special communications packet protocol we describe in detail in Chapter 6. Some of the server details are robot-specific, so we encourage you to consult your robot's operation manual and supplementary Saphira materials for details, as well.

The Saphira client library is available for Microsoft Windows NT and 95 and for most UNIX systems (SunOS, Solaris, SGI, OSF, FreeBSD, and Linux). Saphira sources and libraries are written in ANSI C. There is an Application Programmer's Interface (API) of calls to the Saphira library. Programming details are in the following chapters of this manual.

1.2 Colbert Robot Programming Language

With Version 6, Saphira has added a C-like language, Colbert, for writing robot control programs. With Colbert, users can quickly write and debug complex control procedures, called *activities*. Activities have a finite-state semantics that makes them particularly suited to representing procedural knowledge of sequences of action. Activities can start and stop direct robot actions, low-level behaviors, and other activities. Activities are coordinated by the Colbert executive, which supports concurrent processing of activities.

Colbert comes with a runtime evaluation environment in which users can interactively view their programs, edit and rerun them, and link in additional standard C code. Users may program interactively in Colbert, which makes all of the Saphira API functions available in the runtime environment. Future additions to Colbert include a compiler for efficient execution of debugged programs, and multiple-robot coordination.

1.3 Behavior Compiler and Executive

Saphira uses *fuzzy control rules* for implementing low-level control programs, or *behaviors.* Behaviors are defined using standard C structures and functions. To make writing and debugging behaviors easier, Saphira has a *behavior compiler* that translates a simple fuzzy-control-rule syntax into the required C code. As of Saphira 6.1, behaviors are a type of activity, and can be turned on and off from the activities window. The behavior window is output-only and shows more detail on behavior execution.

1.4 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.

1.5 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 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

Optional:

- C-program source-file editor and compiler. *Note: The current Windows95/NT version of Saphira supports only Microsoft's Visual C/C++ software, not Borland's Turbo-C/C++ products*. Necessary for compiling new subroutines in standard C.
- Motif GUI and libraries for FreeBSD/Linux/UNIX. Necessary only to compile new clients; with Colbert, users may instead operate in an application environment that is already compiled

1.6 Saphira Client Installation

The latest information for installing and running Saphira can be found in the `readme` file in the distribution; please examine this file carefully before and during installation. The `update` file has information about major changes in the latest releases of the Saphira system; you should consult it as a general guide for updating older programs.

The Saphira distribution software, including the `saphira` demonstration program, Colbert, simulator, and accompanying C libraries, come stored as a compressed archive of directories and files either on a 3.5-inch, 1.44MB floppy diskette, or at the ActivMedia 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 ActivMedia Internet site; see *Additional Resources* later in this chapter for details.

The Windows95/NT versions are PKZIP'd, and UNIX versions come GZIP'd and TAR'd. To decompress the software into usable files, you will need the appropriate decompression/archive software: PKUNZIP, GUNZIP, or compatible program; 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` or another publicly accessible directory, and set the appropriate permissions for access and use by your

[1] We do not recommend using Macintosh for Saphira development at this time, because the native operating system (System 8) does not fully support multitasking, which is essential for Saphira operation.

robotics groups. Copy the Saphira archive to that directory, then uncompress and untar the Saphira archive. For example, with Linux the command is:

```
tar -zxvf linux61e.tgz
```

For Windows95/NT or Macintosh, 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\ver61`. 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; `ver61` for Saphira version 6.1, for example. The distribution directory for the Windows 95/NT Saphira version 6.1 looks like the one in Figure 1-1/:

```
ver61/
   bin/
      saphira.exe         Saphira/Colbert runtime application
      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
  colbert/                Colbert language samples
   handler/
   src/
   samples/               tutorial examples
   apps/                  application source examples
   basic/
      behavior.beh        behavior examples
   include/               header files
   obj/                   UNIX library files
   maps/                  Saphira example maps
   worlds/                simulator world files
   params/                parameters for different robots
   readme                 explanation text file
   update                 comparison of versions
   license                operation license
```

Figure 1-1. Distribution directory for Window 95/Windows NT in Saphira version 6.1.

IMPORTANT NOTICE!

All Saphira operations require that the environment variable SAPHIRA be set to the top-level directory, e.g., `/usr/local/saphira/ver61` on a Unix system (note that the directory name does not have a final slash), or `c:\saphira\ver61` on an Microsoft 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 top-level directory. You *must* reset it to the top-level directory of the new distribution. All new clients will complain and fail to execute until you do.

A useful method on UNIX systems is to make a soft link to `/usr/local/saphira/ver61` using the file `current`. The environment variable can be set to `/usr/local/saphira/current` and will remain unchanged when installing a new system; only the soft link `current` need be reset.

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/ver61  (bash shell)
setenv SAPHIRA /usr/local/saphira/ver61  (csh shell)
```

In Windows 95 and NT 3.51, assuming the top-level Saphira directory is `c:\saphira\ver61`, add the following line to the file `C:\AUTOEXEC.BAT`:

```
     SET SAPHIRA=C:\saphira\ver61
```

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

The Saphira library is now in a sharable form on both UNIX and MS Windows machines. This means that a Saphira application will link into the library at runtime, rather than compile time. All clients share a copy of the library, take up less space, and are quicker to compile. Saphira applications must be able to find these libraries.

Under UNIX, the distribution contains the file `handler/obj/libsf.so.6.0.x`. You can make the library accessible to an application in two ways. We recommend leaving it in this directory and putting the directory name onto the load library list using the `shell` command:

```
export LD_LIBRARY_PATH=${SAPHIRA}/handler/obj
```

A second method is to copy the library file into the standard library directory, usually `/usr/lib`.

Under MS Windows, the shared library is `bin\sf.dll`. You must copy or move this file to the standard MS Windows system directory. In Windows 95 this is `C:\Windows\System`; in Windows NT it is `C:\Winnt\System32`.

If an application cannot find the shared libraries, it will complain and exit. Also, problems will arise if the application uses older libraries. It is good practice to clean up by deleting older shared libraries after doing an installation.

1.7 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, use the mouse to double-click the `saphira.exe` icon inside the `saphira/ver61/bin/` folder on your Windows 95 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.

The Saphira client window will appear, with a graphics display of the robot internals, a text information display, and an interaction window. Type `help` in the interaction window for a list of command classes that you can query for further information.

Have a robot server or the simulator readied for a Saphira connection. For example, execute the `saphira/ver61/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 interaction window, type `connect serial` to connect on the standard serial port. If your radio modem is connected to a different serial port, use `connect serial <port>`, where `<port>` is the name of the serial port, e.g., `/dev/ttyS1` or `COM2`.

If you're using the simulator, you can connect using `connect local`, which opens a local port to the simulator and starts things up. You should have started the simulator first by executing `pioneer(.exe)` from the same `bin/` directory.

Bxx users can connect using either a TCP/IP connection or a local connection; typically the Saphira server will start listening on a local port. Run the Saphira client on the same machine as the server, and use `connect local` to make a connection.

You also can connect via the Connect menu on the main Saphira window.

After 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 that 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.) The demonstration Colbert program `colbert/demo.act` is loaded automatically in the sample application; it and has more activities you can try out, by starting them from the Function/Activities window.

1.8 Additional Resources

Every new Saphira licensee gets three additional and valuable resources: a private account on our Internet server for downloading Saphira software, updates, and manuals; access to the private Saphira-users newsgroup; and direct access to the Saphira technical support team.

1.8.1 Saphira Software

We have a World Wide Web server connected full-time to the global Internet, where customers obtain Saphira software and support materials:

`http://robots.activmedia.com`

Some areas of the website are restricted to licensed customers. To gain access, enter the username and password that are written on the ***Registration & Account Sheet*** accompanying your Saphira distribution and this manual.

1.8.2 Saphira Newsgroup

We maintain an e-mail-based newsgroup through which Saphira owners can share ideas, software, and questions about the robot. To sign up, send an e-mail message to our automated newsgroup server:

`To:` **`saphira-users request@activmedia.com`**
`From:` *<your return e-mail address goes here>*
`Subject:` *<choose one command:>*
`help` (returns instructions)
`lists` (returns list of newsgroups)
`subscribe`
`unsubscribe`

Our SmartList-based listserver will respond automatically. After you subscribe, send your e-mail comments, suggestions, and questions intended for the worldwide community of Saphira users:

`To:` **`saphira-users@activmedia.com`**
`From:` *<your return email address goes here>*
`Subject:` **<something of interest to all members of saphira-users>**

Access to the Saphira-users newslist is limited to subscribers, so your address is safe from spam. However, the list currently is unmoderated, so please confine your comments and inquiries to issues concerning the operation and programming of Saphira.

1.8.3 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 Saphira? Share your thoughts and questions directly with us:

saphira-support@activmedia.com

Your message goes to our Saphira technical support team; a staff member will help you or point you to a place where you may find help. Because this is a support option, not a general-interest newsgroup like `saphira-user`, we must reserve the option to reply only to questions about bugs or problems with Pioneer.

1.8.4 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**.

1.8.5 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. Originally, it was 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, which also appeared in an April 1997 segment of the same program.[3]

The Saphira system can be thought of as two architectures, with 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 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 that users may pick among various methods for achieving an objective, for example, choosing between a fuzzy control regime or. more direct control of the motors. It is also an *open architecture,* as users may substitute their own methods for many of the predefined routines, or add new functions and share their innovations with other research groups.

In this section, we'll give a brief overview of the two architectures and discuss 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-taskss that are invoked during? 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`**).

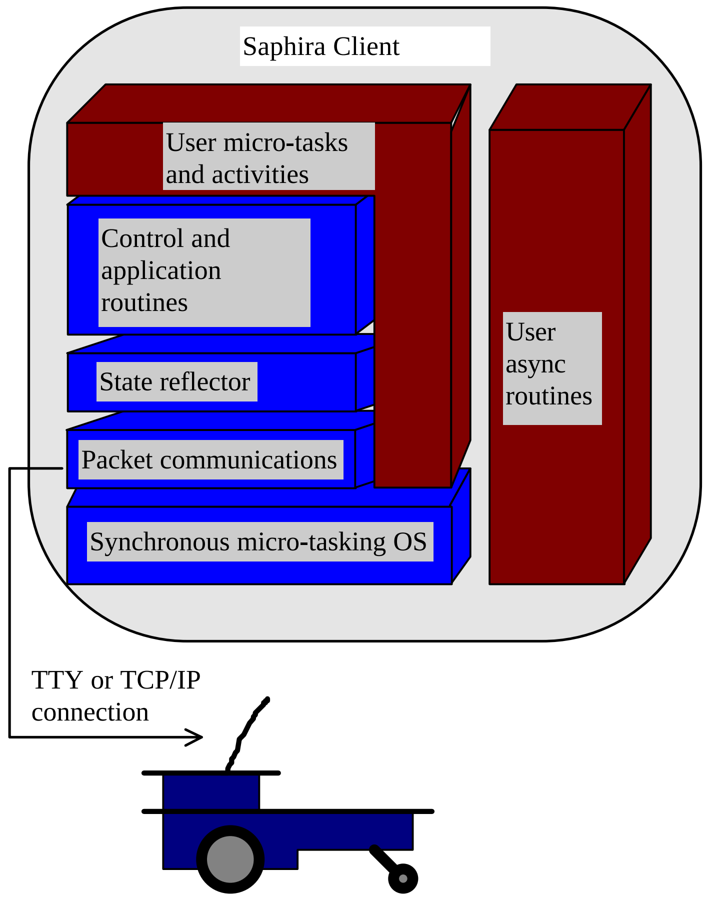


Figure 2-1 Saphira System Architecture.

Blue areas represent routines in the Saphira library, red routines are from the user. All the routines on the left are 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 also can 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's 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 much more convenient to write *activities* in the Colbert language. The activity language has a rich set of control concepts and a user-friendly syntax, both of which make writing control programs much easier. Activities are a special type of micro-task and run in the same 100 ms cycle as other micro-tasks. Activities are *interpreted* by the Colbert executive, so the user can trace them, break into and examine their actions, and rewrite them, without leaving the running application. Developers can concentrate on refining their algorithms, rather than dealing with the limitations of debugging in a compile-reload/re-execute cycle.

Because 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 they 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, such time-consuming operations as 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 need to include only 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 one to four 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 7.3). 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, an Ethernet with TCP/IP, or a local IPC link.

Because 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 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 2-1). It consists of a set of micro-taskss that implement all of the functions required for mobile robot navigation in an office environment. A typical client will use a subset of this functionality.

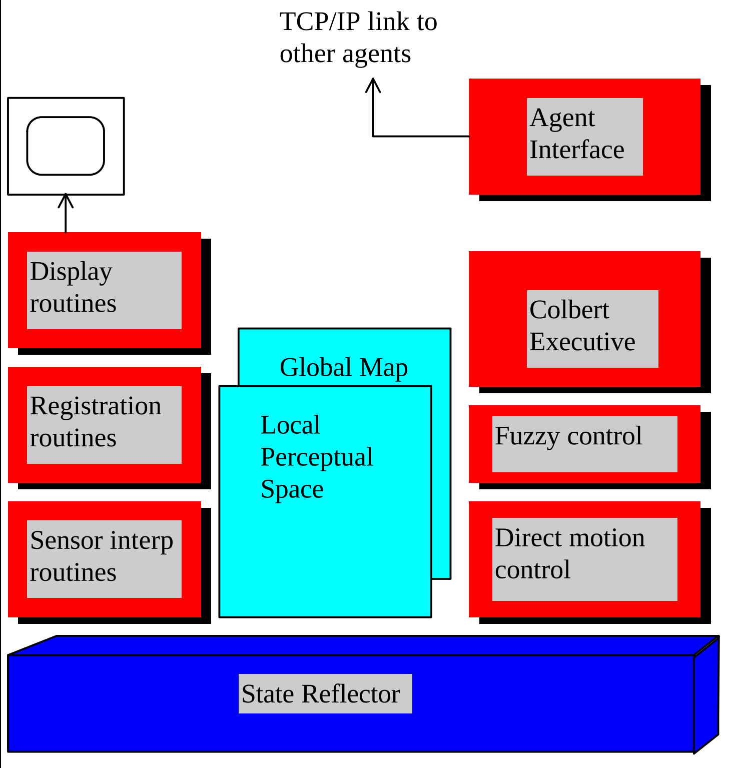


Figure 2-1. Saphira's Control Architecture.

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 registering obstacles to be avoided. The LPS gives the robot a sense of its local surroundings. The main Saphira interface window displays the robot's LPS (see Figure2-1). 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.

Two *direct motion channels* handle rotation and translation of the robot. Any combination of velocity or position setpoints may be used for these channels (see Section 8.4).

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 and activity level, as well as 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.

Version 5.3 includes several major changes in behavior management. Behaviors are no longer invoked with `sfInitBehavior` or `sfInitIntendBeh`; instead, use `sfStartBehavior` (which takes a variable number of arguments for the behavior), or the `start` command from the Colbert interaction window.

Behaviors can be turned on and off by sending them signals, either from the interaction window, or from the Function/Activities window. Behaviors can *not* be controlled from the Function/Behaviors window; the check box that appears there shows only whether a behavior is active or not.

2.2.4 Activities and Colbert

To manage complex goal-seeking activities, Saphira provides a method of scheduling actions of the robot using a new control language, called Colbert. With Colbert, you can build libraries of activities that sequence actions of the robot in response to environmental conditions. For example, a typical activity might move the robot down a corridor while avoiding obstacles and checking for blockages.

Activity schemas are the basic building block of Colbert. When *instantiated*, an activity schema is scheduled by the Colbert executive as another micro-task, with advanced facilities for spawning child activities and behaviors, and coordinating actions among concurrently running activities.

Activity schemas are written using the Colbert Language. The language has a rich set of control concepts, and a user-friendly syntax, similar to C’s, that makes writing activities much easier. Because the language is *interpreted* by the executive, it is much easier to develop and debug activities, because errors can be trapped, an activity changed in a text editor, and then reinvoked, without leaving the running application.

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 work with data reflected 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 also may invoke the graphics routines directly to display relevant information.

2.2.7.1 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; 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 Saphira's capabilities 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 requires only runtime files found on your system, and the relevant loadable libraries from Saphira (`sf.dll` or `libsf.so.6.0.x`). You should have installed these as directed in Section 1.6.

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

2.3.1 Loading an Activity File

The Saphira client in `bin/saphira` has only a bare set of micro-tasks loaded (you can see the source code in `handler/src/apps/saphira.c`). The capabilities of the client are increased by loading in Colbert files, which contain activity schemas and invocations of API functions. A sample activity file, `colbert/demo.act`, is used as an example in the rest of this section (the `.act` extension signifies a Colbert language file). When the `saphira` client starts, it looks for the file `init.act` in the current load directory, which by default is `$(SAPHIRA)/colbert`. The initialization file loads the demonstration file `demo.act`.

To load your own init file, you can either change the load directory by setting the environment variable `SAPHIRA_LOAD`, or change the `init.act` file in the `colbert/` directory.

The `demo.act` file defines several activity schemas, then invokes them and a few predefined behaviors for obstacle avoidance. Please refer to the code for more details.

2.3.2 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 (the simulator defaults to using Pioneer parameters; see Chapter 3). Otherwise, the Saphira client auto-detects the robot server type and loads its parameters when first connected (see Chapter 6 for details), so it isn't necessary to load a parameter file into the Saphira application unless you're using a custom configuration.

You can connect using either the interaction window commands or the menu.

Serial port connection to Pioneer (radio modem or fixed line). In the Saphira interaction window, type `connect serial` to connect on the standard serial port. If your radio modem is connected to a different serial port, use `connect serial <port>`, where `<port>` is the name of the serial port, e.g., `/dev/ttyS1` or `COM2`. The Connect/Serial Port menu item will also work for the standard serial port. You can set the standard serial port and baud rate; see Section 4.6 for details.

Simulator connection. If you've started the simulator, it's listening on a local internal port. Type `connect local`, which opens the local port to the simulator and starts things up. Or, choose the Connect/Local Port menu item.

B14 and B21 users. Bxx users must start up the Saphira server on a Bxx computer; see the instructions that come with the Saphira server software. Usually, the Saphira server will start listening on the local port. Run the Saphira client on the same machine as the server (with telnet from a desktop machine), and use `connect local` in the interaction window or the Connect/Local Port menu item.

If you have a problem connecting with 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. Typical causes for failure of the simulator or the actual robot (and their solutions) include:

(Bxx robots) 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 no other Saphira client or simulator server is running on the same machine.
In rare cases, the communications pipe may be blocked. This can occur if the server or client exits abnormally from a previous connection, without shutting it down properly. Try deleting the pipe file and starting again. If this doesn't work, the only remedy is rebooting the machine.
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.
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.

2.3.3 LPS Display

The Saphira client's display contains most of the items likely to be found in the robot's LPS (see Section 8.6). 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 Display/Local menu item.

The main Saphira window components include:

2.3.3.1 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 (see Chapter 9)

2.3.3.2 Sonar readings

Accumulated sonar readings appear on screen as small open rectangles. Current sonar readings are slightly larger open rectangles. The number of sonar readings accumulated can be set by the user (see Section 8.6.1 for more information about the buffers).

2.3.3.3 Control point

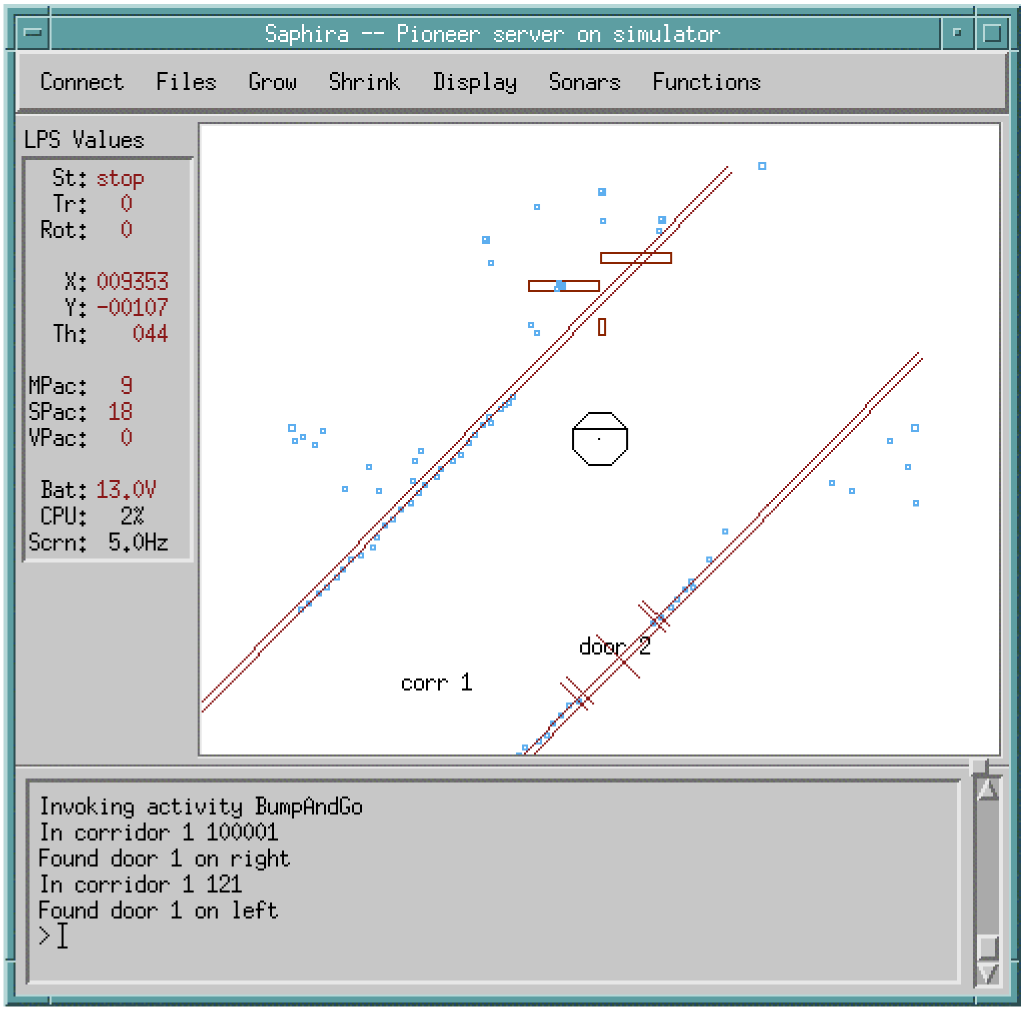
The elongated open rectangle directly in front of the robot icon is its heading control point, as returned by the server in robot-centric coordinates. Normally, this control point is positioned directly ahead of the robot, veering to one side or the other in response to a turn directive from the client.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

2.3.3.4 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.

2.3.3.5 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.



```
St: stop
Tr:    0
Rot:   0

  X: 009353
  Y: -00107
 Th:    044

MPac:  9
SPac: 18
VPac:  0

Bat: 13.0V
CPU:    2%
Scrn: 5.0Hz
```

```
Invoking activity BumpAndGo
In corridor 1 100001
Found door 1 on right
In corridor 1 121
Found door 1 on left
>I
```

Figure 2.-3. Saphira client LPS in local mode.

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

2.3.4 Artifacts

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

2.3.5 Information Area

The information area is at the left of the main window. It contains four sets of data returned from the robot server.

2.3.5.1 Status (St)

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

2.3.5.2 Velocity (Tr, Rot)

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

2.3.5.3 Position (X, Y, Th)

Absolute robot position in millimeters and degrees. Note that this is *not* the server dead-reckoned position, which 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.

2.3.5.4 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 10 motor packets (`Mpac`) and approximately 25 sonar packets (`SPac`) per second. Vision packets (`Vpac`) currently are not supported.

2.3.5.5 Miscellaneous (Bat, CPU, Scrn)

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. The last value is the LPS update rate.

2.3.6 Text Interaction Area

The interaction area is at the bottom of the window. Here Saphira prints information about the system, and the user can type commands to the Colbert evaluator. A scroll bar allows the user to look at previous information. The small square on the far upper right of the window is a dragging handle for resizing the interaction area.

In the interaction area, you can do the following tasks:

- Load activity files and change the working directory
- Connect and disconnect from a robot server
- Define, start, and stop activities
- Trace and untrace activities
- Get help on API and evaluator functions
- Examine and set internal Saphira variables

The evaluator lets users write and debug programs from the running Saphira application. Usually, the user code will be in a text file that is read into the system with the `load` command, as we did for this example (`colbert/demo.act)`. The code file contains a mixture of activity schema definitions and calls to library functions. The user can invoke the activities from the interaction area with the `start` command, or use the Function/Activities window. During execution, the user can examine the state of Saphira variables, and stop and start other activities. If an error occurs, the offending activity is suspended and a message is printed. The user can change the Colbert text file, reload it, and run the changed activities. There is no need to exit from the application and recompile. Even new C functions can be dynamically linked into the system by loading a shared object file.

2.3.7 Menus

The main client window contains seven pull-down menus.[4] 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:

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

2.3.8 Connect Menu

The Connect menu lets you make and break a connection to the robot server. The menu contains three items: the standard serial port, a local port for the simulator and Bxx robot servers, and a TCP connection. Choosing one of these items causes the client to try to connect to the physical robot or to the simulator. Parameters such as the baud rate and port names can be changed from the interaction window or via library calls (see Section 4.6).

The Disconnect option closes an open connection to the robot.

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

2.3.8.1 Files Menu

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

You can save the current map to a file using the Save Map item, which invokes a file-save dialog. Use Delete Map to erase all artifacts in the current map.

The Load menu does not load Colbert files; to do this, use the Colbert evaluator commands in the interaction area.

2.3.8.2 Grow and Shrink

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

2.3.8.3 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. If the number of motor packets (Mpacs) per second falls significantly below 10, and you have a good connection to the robot server, then a high display-update rate may be the culprit.

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 controls only the display of information; all internal geometric structures remain the same.

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

The Wake option, if on, deposits "breadcrumbs" in the display, showing the last 10 seconds of robot travel.

If it is on, the Occ Grid menu item displays the occupancy grid constructed using the MURIEL algorithm.[5] This item is not implemented on Macintosh or machines without color capability. On some machines, turning on Occ Grid may create a situation in which a large percentage of available CPU time is used for updating the display.

2.3.8.4 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. (This item isn't currently implemented on the robot server; the sonars are always on.)

2.3.8.5 Functions Menu

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

[5] The MURIEL algorithm is described in a paper that can be found at `http://www.ai.sri.com/~konolige/saphira`.

2.3.9 Keyboard Actions

In addition to using Saphira's pulldown menus, you may control some of the functions of the robot server directly from the client keyboard (see **Error! Reference source not found.**). 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 "hotkey" actions.

2.3.10 Behaviors Window

Saphira's Behaviors window shows graphically the state of all current behaviors. It is invoked from the Functions/Behaviors menu in the main window. To understand the contents of this window, you may find it 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 mm 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 Stop at a low priority assures that the robot will stop if it is not doing anything else.

Table 2-1. Keyboard-controlled behaviors for the Saphira client.

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

Figure 2-2 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 changed by signaling its invoking activity in the Activities window.

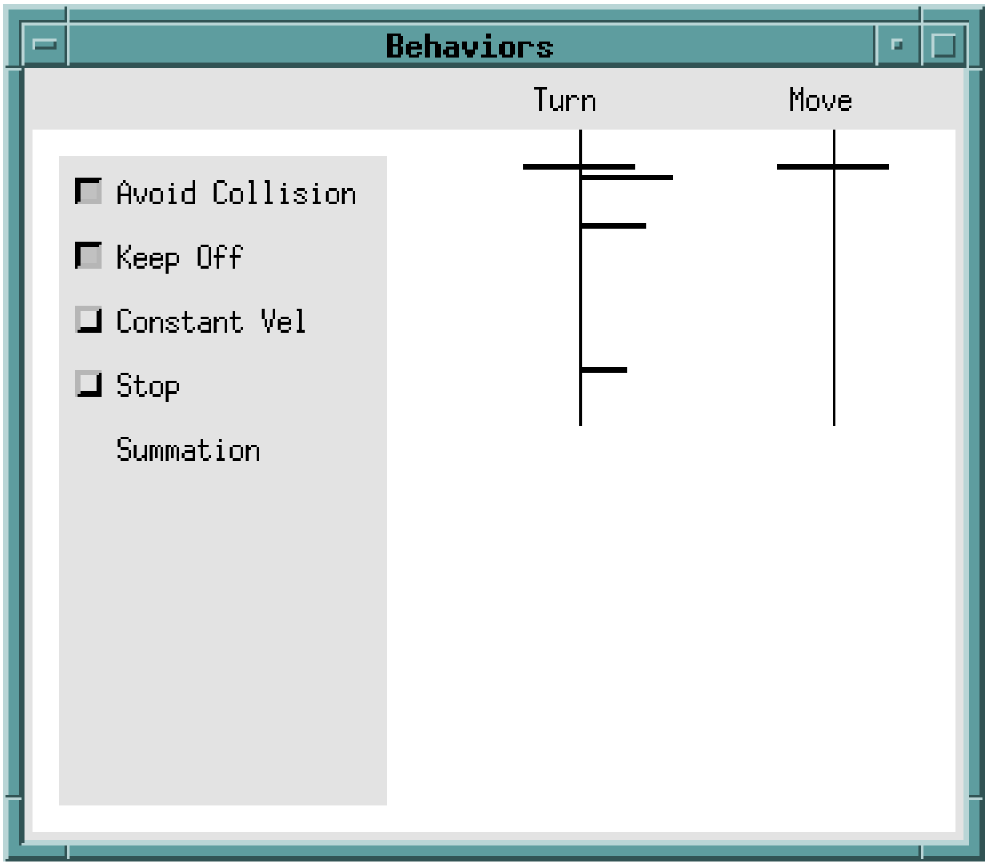


Figure 2-2. Saphira's Behaviors window (Linux/Motif version).

Note: This is a change from version 5.x, in which the buttons were active in the behavior window.

The dark bar next to each behavior name indicates the state of the behavior. Two vertical lines, represent the behavior's outputs for turning and forward/backward movement. For example, the Keep Off behavior in Figure 2-2 is fully active for both turning and moving, as indicated by the horizontal activity bars going through the vertical lines (see the details in Figure 2-3). 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.

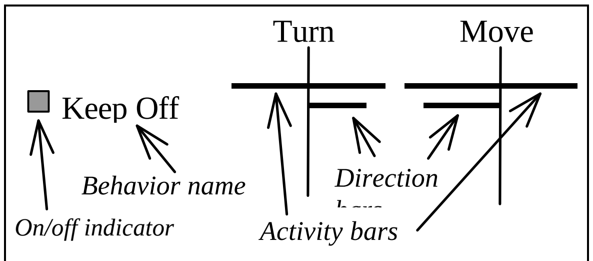


Figure 2-3 Keep Off behavior display expanded

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 2-4 shows a typical behavior window while active. The invocation 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: greater

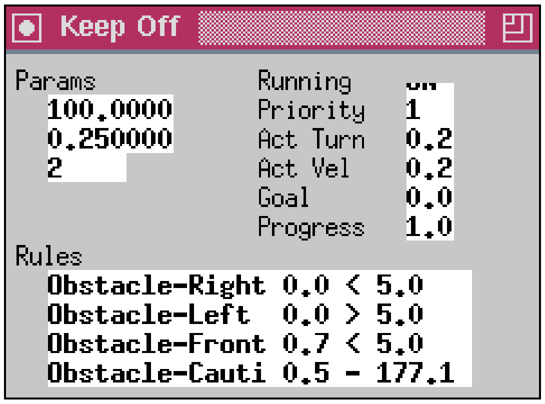
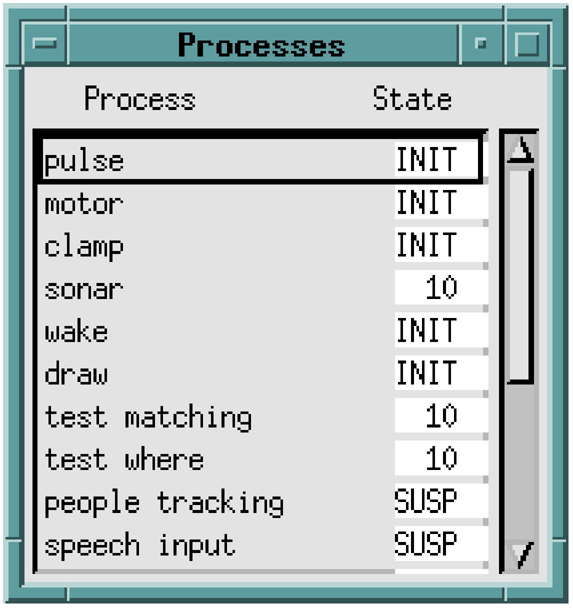

```
Keep Off
Params          Running    on
   100.0000     Priority   1
   0.250000     Act Turn   0.2
   2            Act Vel    0.2
                Goal       0.0
                Progress   1.0
Rules
   Obstacle-Right 0.0 < 5.0
   Obstacle-Left  0.0 > 5.0
   Obstacle-Front 0.7 < 5.0
   Obstacle-Cauti 0.5 - 177.1
```

Figure 2-4. Behavior window for Keep-Off.

than (>) or less than (<) for right or left turn, plus (+) or minus (-) for speed up or slow down. The value indicates the desired control signal; a left turn of 5.0 degrees, for example.

2.3.11 Processes Window

The Processes window displays the states of all micro-taskss in the Saphira client multitasking queue (see Figure 2-5). Open it from the Functions/Processes menu in the main window. The Processes window contains a scrolled list, in which each entry consists of the micro-task name and state. The display is updated in real time as the micro-task state changes.



Process	State
pulse	INIT
motor	INIT
clamp	INIT
sonar	10
wake	INIT
draw	INIT
test matching	10
test where	10
people tracking	SUSP
speech input	SUSP

Figure 2-5. 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 suspend processing; a micro-task's behavior depends on how the micro-task handles the interrupt state. Some micro-tasks 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 interrupt only micro-tasks that you have added to the Saphira application, and for which there is a defined interrupt behavior.

2.3.12 Activities Window

Saphira's Activities window shows the state and relationship of all current Colbert activities (Figure 2-6). Open it from the Functions/Activities menu in the main window.

The Activities window contains a scrolled list similar to the Processes window, and each line contains the activity's name and its state. The state information is updated in real time as the activity state changes.

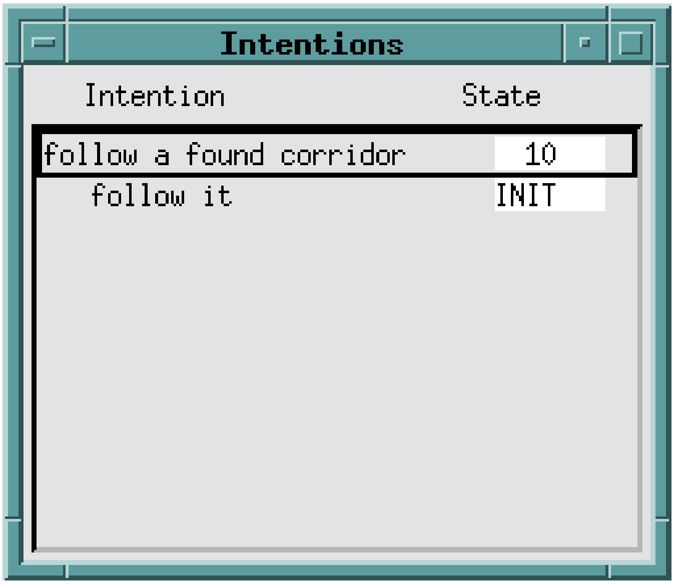
Relationships between activities are indicated by line indentations. For instance, in the example in Figure 2-6, the second activity `follow it` is indented to show that it is a child of the first activity. The two activities combine to invoke a corridor-following sequence for the robot. The top-level activity waits until the robot has found a corridor, then invokes its child activity to select a path to follow down the center of the corridor. In addition, the top-level activity 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` activity.

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

The sample Saphira client contains several activities. Some of these are *wrappers* for a behavior, that is, their sole purpose is to control a single behavior. The reason for this is to provide behaviors with the same facilities as activities, e.g., timeouts, signaling, and hierarchical invocation (see Section 4.8.3).

The activity `BumpAndGo` is an example of an activity that produces *direct action*. It waits until the robot bumps into something and its motors stall out; then it turns off all behavior output and maneuvers the robot in a short back-and-turn sequence to get it out of the stall. This activity is traced, so you'll see the results of its evaluated statements printed in the interaction area. Beware: It's hard to make the robot run into something unless you turn off the obstacle-avoidance behaviors.

Another activity, `follow a found corridor`, has the robot find and follow corridors. The activity monitors the robot environment until it detects a corridor, then starts a subactivity, a behavior, that projects a path for the robot down the middle of the corridor.



```
Intention                State
follow a found corridor   10
  follow it              INIT
```

Figure 2-6. An example Saphira Activities window.

2.3.13 System Environment Variables

Several environment variables can be set to control defaults in Saphira clients. Following is a complete list of them, and their effects. In MS Windows, environment variables are set in `AUTOEXEC.BAT`, or via the user profiles (Windows NT). In UNIX, they are set from a shell using `setenv` or `export`.

Table 2-1. Environment variables used to control defaults in Saphira clients.

Environment Variable	Effect
`SAPHIRA`	Top level of the Saphira distribution. This variable must be set for Saphira clients and the simulator to run correctly. In Unix, there should be no final slash in the path, e.g., `/usr/local/saphira/ver61`.
`SAPHIRA_LOAD`	Initial load directory for the Colbert evaluator. This directory is searched for the file `init.act` when the Colbert evaluator starts. If not set, defaults to the directory from which the client was started.
`SAPHIRA_COMSERIAL`	Serial port for connecting to the robot. Defaults to the primary serial port for the system being used, e.g., `COM1` under MS Windows, `/dev/cua0` under Linux, and so on.
`SAPHIRA_SERIALBAUD`	Baud rate for serial connection. Defaults to 9600.
`SAPHIRA_COMPIPE`	Local communication port for connection to the Saphira simulator. Can be set so that multiple copies of the simulator can run on the same machine, and clients can connect to them. This variable affects both the simulator and the client application. Default depends on the system.
`SAPHIRA_COMSERVER`	Machine name or IP address for TCP/IP connection. Defaults to `NULL`.

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 because 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, the simulator connects to the client using an interprocess communications channel on the same machine. It is also possible to run multiple copies of the simulator on the same machine with different communication channels (handy for class work), or to have the simulator listen on a tty port or a TCP/IP port on a remote 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 **Error! Reference source not found.**. 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.

3.1.1 Listening on Other Ports

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 `SAPHIRA_COMPIPE` 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 from the client side, set the `SAPHIRA_COMPIPE` environment variable to the name of the desired simulator socket before trying to connect. Under UNIX and Windows NT, different users can set these variables in a unique way, so that several users logged in to the same machine can start up their private versions of the simulator.

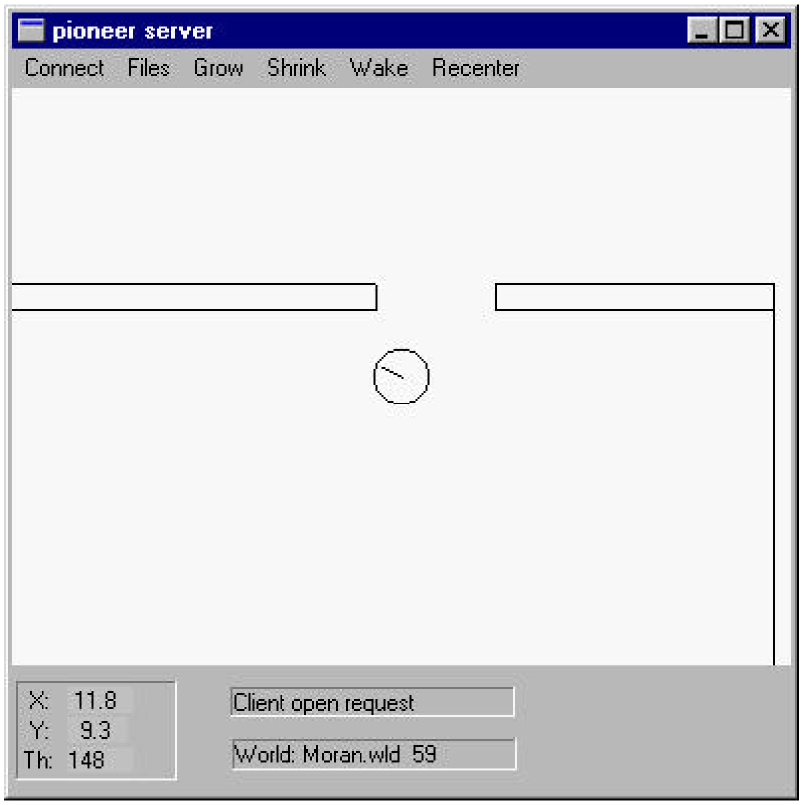
The simulator also can listen on a tty port (for debugging tty access) or TCP/IP socket (for remote machine access). In these cases, the simulator must be started with command-line arguments specifying the type of access. Two choices are available:

```
pioneer tcp
pioneer /dev/tty1
```

The first form starts the simulator and listens on a TCP/IP socket for network connections from a client. On the client side, you must specify the network address or network name of the machine the simulator is running on (using the `set server` command or the `SAPHIRA_COMSERVER` environment variable).

The second form accepts any argument that is not `tcp`. This argument is assumed to be the name of a tty port, and the simulator listens for connections on that port.

In MS Windows, you can start the simulator with command-line arguments by using the Run item in the Start menu.



X: 11.8
Y: 9.3
Th: 148

Client open request

World: Moran.wld 59

Figure 3-1. A sample window of the simulator.

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 use a parameter file to prescribe a variety of simulated robot characteristics, such as placement of sonars and drive-error tolerances. Once constructed, store your parameter file in common text (ASCII) format in the `params/` directory; usually, you add the suffix `.p` to the file name. A sample, annotated parameter file listing is in Appendix A, and the parameter file resides 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 9 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 file name suffix `.wld,` which describes the size and contents of a simulated world. A sample world file can be found in the Chapter 10, 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.

If the simulator is connected to a client, the client can tell the simulator to load a world file via the `sfLoadWorldFile` function.

3.4 Simulator Menus

Several simulator menus control the parameters and actions of the simulated robot. The menu options provide 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 File/Load Params 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 File/Load World 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 also can 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 in which 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, the Connect menu also includes an Exit option.

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, 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.

Usually, 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.

3.6 Compass

The simulator’s compass has a standard deviation of 3 degrees from the robot’s true heading. Compass readings are sent back in the information packet. The simulated compass differs from the real compass in that it does not reflect bias in the magnetic environment, which can be quite severe. In the simulator, magnetic north is always along the positive *x* direction.

4 Using Colbert

This section describes the Colbert language and evaluator. Colbert is a C-like language with a semantics based on finite state machines. It has the such standard C constructs as sequences, iteration, and conditionals, but they are interpreted in a way that makes sense for robot programming. The main construct of Colbert is the *activity schema*, or *act*, a procedure that describes a set of coordinated actions that the robot should perform.

Colbert is an *interpreted* language, which means that you write text files containing Colbert activities, load them into a Saphira client, and then start them up. The Colbert evaluator parses and executes the activities, and reports back results and errors. Having an evaluator is very convenient for development and debugging, because you can try out code without having to recompile and relink an entire client, and then try to get back to the state you're interested in.

The Colbert evaluator has the following capabilities.

Direct execution of control statements from a running Saphira client.

Tracing of activities: users so that can see exactly what statements are being executed.

Signaling between activities: activities may start sub-activities, or interrupt activities that are already running.

Trapping of errors: fatal errors, such as divide by 0, disable just the offending activity and print an error message.

Error correction: buggy activities can be edited with a text editor and reloaded, without exiting the running Saphira client.

A technical paper describing Colbert is available from the website **`http://www.ai.sri.com/~konolige/saphira`** in the Publications section.

4.1 A Colbert Example

We'll introduce the Colbert language with a short example, using the direct motion calls to the robot (see Listing 4-1). The example file is in `colbert/direct.act`.

The first step is to start the Saphira client and connect to a robot or the simulator (see Section 2.3). After you've successfully connected, try typing the following statement in the interaction area at the bottom of the client:

```
> move(500);<cr>
```

This is an example of a *direct motion* command, which tells the robot to move immediately (see Section 8.4). You must type the semicolon to indicate the end of the command, just as in C, otherwise the evaluator will complain about a syntax error. The robot should move forward ½ meter (500 mm). If you execute this command without connecting to the robot, you receive an error message indicating that the command cannot be executed. You can try other commands such as `turn` (a complete list of the direct motion commands is in Section 4.7), or you can type `help movement` to have the list printed in the interaction area. Utility commands such as `help` and `load` do not follow normal C syntax, and a semicolon is unnecessary.

You can enlarge or shrink the interaction area by grabbing the separator handle (located at the left, between the LPS and interaction windows) with the mouse, and moving it up or down.

The next step is to load the sample file. First, check the current load directory with the `pwd` command in the interaction area. By default, it is the directory of the shell from which you started Saphira (the default can be changed by setting the environment variable `SAPHIRA_LOAD`). The load directory can be changed with the `cd` command, or you can give the load path directly in the `load` command, relative to the current directory, or as an absolute path.

For example, if you're in the `bin` directory, use this sequence:

```
load ../colbert/direct.act<cr>
```

(in MS Windows, the forward slashes will be backslashes). Loading the file defines three activities, and

```
/* Colbert example exercising the direct motion calls */
act patrol(int a)               /* go back and forth 'a' times */
{
while (a != 0)
{
turnto(180);
     a = a-1;
move(1000);
turnto(0);
move(1000);
}
}

act square                      /* move in a square */
{
int a;
a = 4;
while(a)
{
a = a-1;
move(1000);
turn(90);
}
}

act aa                          /* call them sequentially */
{
trace patrol;
start patrol(4);
trace square;
start square;
}

sfSetDisplayState(sfGLOBAL, 1);  /* put display into global coords */
start aa;                        /* start up the toplevel activity */
```

Listing 4-1. A direct motion application in the Colbert language.

starts one of them (`aa`), which calls the other two. A listing of the file is in Listing 4-1. Activity schemas are defined in a manner similar to C functions, using the keyword `act`. Just as with C functions, acts take arguments, which are given when the activity is called, or *instantiated*. For example, in the act `aa`, the `patrol` activity is called with an argument of 4, which means that the robot will go back and forth 4 times.

The direct motion commands in `patrol` and `square` are executed by the evaluator, which waits until they complete before moving on to the next statement. The same thing is true of the calls to the `patrol` and `square` activity within `aa`. This is an example of *blocking execution* of motion commands or activities, which is generally desirable for sequences of actions. In other cases, you may want to start several activities in parallel (e.g., a monitoring activity and a direct-action activity). In this case, Colbert provides a *nonblocking* instantiation mode.

In addition to direct motion calls, activities can reference standard C variables and functions. A number of library variables and functions are available initially, and more can be added through the use of the `sfAddEvalXXX` functions. The C syntax of the evaluator has some limitations; for example, you can't embed assignments within a C expression.

Unlike standard C files, Colbert files allow you to execute statements from within the file. In the example, the last two statements are executed. One is a call to a library function for setting the state of the LPS display. The other starts up the `aa` activity. So, loading the file has the effect of defining three acts, then setting the display state and starting the top-level activity.

4.2 Evaluator Interaction Area

The interaction area is at the bottom of the Saphira client window. This area is always present in a Saphira client for output of messages (`sfMessage` and `sfSMessage` library calls). If the micro-task `sfRunEvaluator` has been invoked, then it is also available for user text input to the Colbert evaluator. The sample client `bin/saphira` invokes the evaluator.

At the beginning of a session, several lines are written to the interaction window, showing the Saphira top-level directory and the current working directory for loading Colbert files. There is an input prompt (>). You may type input at this prompt, and edit it using standard editing commands, e.g., the delete and backspace keys. The characteristics of text editing are set by the `XKeysymDB` file and the X resources databases in UNIX. If you have trouble getting text editing to work in UNIX systems, please check with a local X guru.

The evaluator accepts commands and activity definitions from the user. Commands are always just a single line, but you can extend a line by typing a backslash (\) as the last character, and continuing on subsequent lines. A carriage return (`<cr>)` is needed to input the line. The cursor need not be at the end of a line in order to use a carriage return`.` At the command line, a terminating semicolon (;) is optional for all statements.

For convenience, some of the utility commands do not adhere to C syntax. For example, the `load` command accepts its string argument without quotes, so you can type `load src/test.act`, for example.

You have access to a history list of previous input. You can cycle through previous lines by using the Ctrl-P (back) and Ctrl-N (forward) keys. After you retrieve a line, you may edit it. Text may be selected, cut, and pasted using the standard mouse keys. As in C, case is significant.

A scroll bar on the right sight of the interaction area lets you scroll back through previous messages. Currently, no limit is imposed on the amount of text kept.

4.3 Evaluator Help

The evaluator has a simple help facility to remind you of commands.

`help` provides a list of help topics

`help topic` provides help on the specified topic

`help <fn>` provides helps on an API function, or list of API functions containing `fn`

Topics include utility commands (file loading, directories), communication, direct motion commands, and information on particular API functions. Not all API functions have associated help text; we are adding them in future versions.

Help text can be added using the `sfAddEvalHelp` function, and retrieved with `sfGetEvalHelp`. Both these functions are available in Colbert and from compiled C code.

4.4 Syntax Errors

As much as possible, Colbert uses ANSI C syntax. But it also extends this syntax with new commands and constructions for robot control, and omits some parts, such as embedded assignments and arrays. If the parser cannot understand the input, it will print an error message in the interaction area, and abort the loading of any file currently in progress.

Determining the reason for a syntax error is a difficult problem, and the parser does not even try to do this. Instead, it will print the token that it was trying to parse when the error occurred, as well as the line number in the file, if a file was being loaded. For example, the ill-formed C expression:

```
1 + ) 2;
```

produces the error message:

```
*** Parsing error at token ")"
```

because the parser could not fit the token) into the C expression it was trying to form. These are the most common sources of syntax errors:

C constructions not supported by Colbert. These include embedded assignments, variable initializations, the comma operator, arrays, etc. (see Section 4.9.1).

Colbert keywords that are not ANSI C keywords. There are many of these (e.g., `connect`, `waitfor`); see Section 4.9.3.

Functions not defined in Colbert. Most C library functions are not initially available in Colbert, although you can make them accessible (see Section 4.10.2). Using one of these functions will give a syntax error.

4.5 Evaluator File Loading

Colbert source files may be input from text files, using the `load` command. Any errors in the source are indicated in the interaction window, and file loading is aborted at that point. Load files can contain definitions of activities, as well as commands to be executed, including any commands that can be typed in the user interaction area. So, for example, it's possible to load a file that loads other files.

This command, for example, loads file from the current load directory:

```
load [<file>]
```

`file` is actually a path from the current directory; e.g., `colbert/demo.act` is a legal filename. C syntax does not apply to filenames, so any non-blank characters are allowed. Without arguments, the command prints a list of loaded shared object files.

This command unloads a shared object file:

```
unload [<file>]
```

It is used only under MS Windows for unloading DLLs. Without arguments, it unloads the last shared object.

Colbert source files can have an arbitrary extension (except for `.so` or `.dll`), but by convention their extension is `.act`. This extension must be included in the filename.

Evaluator files can be changed and reloaded as often as desired. If an activity schema is redefined by reloading, then all instances of the schema are changed. This has implications for how the user should handle instantiated activities that are being debugged. The state of the activity is not changed by the redefinition, so the activity will continue execution at its current line. This may not make sense if the line numbering has changed. However, the standard states (`sfINTERRUPT`, `sfSUSPEND`, and so on) are the same for all activities, so these are "safe" states for redefinition. In general, it's probably best to suspend an activity if you're going to change its definition.

Redefining an instantiated activity does *not* change its arguments or internal state. Normally, this is what you would like, because the activity can resume operation with the same arguments and internal variables. However, if the number of arguments or their ordering is redefined, or internal variable declarations are changed, then the instantiated activity may be confused as to how to find the values of these entities. In this case, it is better to remove the activity and restart it.

4.5.1 Loading Shared Object Files

Some API functions will work only in compiled C code, and cannot be called from the evaluator. These include such functions as `sfAddEvalVar` and `sfAddEvalStruct`, which access underlying C constructs. In addition, application code which performs significant computation should be compiled as C code for efficiency.

The loader will load compiled C code in the form of shared object files (`.so` extension in UNIX, `.dll` in MS Windows). These files are loaded and dynamically linked with the running Saphira system. (See Sections 4.10 and Chapter? 6 for information on how to compile shared object files, and for some examples.) The loader recognizes the extension and calls appropriate dynamic loading routines. If present, the function `sfLoadInit()` is evaluated after the file is loaded.

Under MS Windows, it is impossible to relink a DLL file that is in use by an application. Therefore, you must unload the DLL file first, using the `unload` command. For convenience, `unload` with no arguments unloads the most recently loaded file.

Table 4-1. Colbert commands to query and set the load directory.

Command	Effect
`pwd`	Prints working directory, value of variable `sfLoadDirectory`.
`cd <path>`	Changes the working directory according to path. Path may be an absolute or relative path. Prints the new working directory. Affects `sfLoadDirectory`

4.5.2 Load Directory

Files are loaded based on the current load directory. The following commands query and set this directory (see Table 4-1). The argument to `cd` does not use C syntax, and can contain any non-blank characters.

By default, the initial load directory is the directory of the shell that Saphira was started in. The default load directory can be changed by setting the environment variable SAPHIRA_LOAD to a directory. The load directory is also available to programs as the API variable `sfLoadDirectory`, whose type is a string. Setting this variable causes the load directory to change.

When started, the evaluator will look for a file `init.act` in the initial load directory, and load it in. This file is used for automatically configuring Saphira on start-up.

4.5.3 Sample Application Files

Sample files that mimic the behavior of the old `saphira` and `direct` clients are available in the `colbert/` directory.

Table 4-2.

Command	Effect
demo.act	Invokes several behaviors, along with some activities: bump-and-go for getting out of stall situations, and follow-corridor for following a found corridor. Some of these activities and behaviors are started in a suspended state; double-click on them in the Activities window to start them.
direct.act	Defines some simple direct motion activities, and starts them up. Must be connected to a robot, or you'll receive an error message when starting the direct motion commands.
packet.act	Communicates directly with the robot using the packet protocol.

4.6 Communication and Connection Utilities

Colbert offers several utility commands for setting communication modes and for connecting and disconnecting with the robot (see Table 4-3).

Table 4-3 Colbert commands for connecting to and disconnecting from the robot.

Command	Effect
`connect serial [<port>]`	Connects via `<port>` or the current serial port (`sfComSerial`) at the specified baud rate (`sfSerialBaud`).
`set serial [<port>]`	Sets or returns the serial port (`sfComSerial`). If `<port>` is given, sets the serial port to this value. The first serial port of the machine (`COM1`, `/dev/ttya`, etc.) is the default.
`set baud [<rate>]`	Sets or returns the baud rate of the serial connection (`sfSerialBaud`). If the argument `<baud>`.is given, sets it to this value. The default rate is 9600 baud. With PSOS 4.3, the Pioneer server now supports 19200 baud. Other baud rates may be used for specialized applications.
`connect local [<pipe]`	Connects via the local communication port with name `<pipe>` or the default name (`sfComPipe`). This is the normal connection for the simulator, or for the Bxx robot servers. The default name of the connection can be changed by setting `sfComPipe` to another string.
`set local [<pipe>]`	Sets or returns the local connection name. This command is useful when running multiple simulators on the same machine, because each simulator can be assigned a unique local connection name.
`connect server [<netaddr>]`	Connects to a robot server via TCP/IP to a remote machine specified by `<netaddr>` or the default address (`sfComServer`). The robot may be a Bxx or simulator server on a remote machine.
`set server [<netaddr>]`	Sets or returns the remote server net address (`sfComServer`). These addresses may be network names (e.g., `flakey.ai.sri.com`) or numbers (e.g., `128.18.65.12`).
`disconnect`	Disconnects from the currently connected robot server.
`exit`	Exits from the Saphira executable, disconnecting from any robot server first.

Parameters to the connection commands are usually held in library variables and can also be accessed (set and queried) by using the variables.

4.7 Direct Motion Commands

The evaluator provides a set of direct motion commands that can move and rotate the robot. These commands are Colbert language statements, and can be typed in the interaction window.

The direct motion commands are *not* C functions, and do not return any value. They also have a syntax for specifying a timeout value and a non-blocking mode. The general form of a direct motion command is:

```
command(int arg) [timeout n] [noblock];
```

where `timeout n` specifies a time in 100 ms increments for the command to complete, and `noblock` means that the command will be executed without blocking, i.e., control will continue with the next statement. Some motion commands are implicitly non-blocking: `speed` and `rotate`. In the interaction window, all direct motion commands are issued non-blocking, whether or not `noblock` is specified. Non-blocking motion commands can be checked for completion with the `sfDonePosition` and `sfDoneHeading` commands.
More information on direct motion control, as well as C library functions, can be found in Section 8.4. These API calls are available from the evaluator, and are an alternate way of issuing direct motion commands.

Table 4-4.

Command	Effect
`move(int mm);`	Move the robot `mm` millimeters forward (positive) or backwards (negative). Blocking.
`turn(int deg);`	Turn the robot `deg` degrees clockwise (negative) or counter-clockwise (positive) degrees from the current heading. Blocking.
`turnto(int deg);`	Turn the robot to the heading `deg` degrees. Positive values are counter-clockwise, negative values are clockwise. Blocking.
`speed(int mms);`	Move the robot at a speed of `mms` millimeters per second forward (positive) or backwards (negative). Non-blocking.
`rotate(int degs);`	Move the robot at a rotational speed of `degs` degrees per second counter-clockwise (positive) or clockwise (negative). Non-blocking.
`halt;`	Halts all direct motion commands.

4.8 Activity Schemas

Activity schemas are Colbert language programs for controlling the robot. They are interpreted using the Colbert evaluator. Activities execute similarly to normal C functions, evaluating statements in order, with sequences, loops, and conditionals. However, the underlying execution model is quite different: it is a finite-state machine. Each statement of the activity is a node that can potentially wait for a condition to hold before going on to the next statement, or can change the flow of execution. For example, every primitive action (`move`, `turn`, and so on) that is invoked causes the program to stay at that statement until the action is completed or times out.

4.8.1 Act Definition

Activities are defined as sets of statements in the Colbert language (see below). As the example in Listing 4-2 shows, the syntax is similar to that of C functions, with the keyword *`act`* as the first token:

```
 act actname(parameters)
{
   variable declarations
   update statements
   body statements
}
```

Listing 4-2. The syntax of Colbert activities is similar to that of C functions.

Listing 4-3 shows a sample activity that moves the robot in a square. One internal variable, `a`, keeps track of the four legs of the square. The main body of the act is a `while` loop that decrements `a`, turns the robot 90 degrees, and moves it along on the next leg. The act's parameter, `len,` specifies the length of each side of the square.

```
act square(int len)              /* move in a square */
{
int a;

 update
  { sfSMessage("a is: %d", a); }

a = 4;
while(a)
{
a = a-1;
move(len);
turn(90);
}
}
```

Listing 4-3. A sample activity schema definition in Colbert.

We haven't explained yet how the act is executed; the next subsection explains this in detail. But note that the `move` and `turn` actions halt the activity until they are completed. The Colbert evaluator accomplishes this by calling the activity periodically to check and see if it can proceed. On each call, the `update` statement is evaluated. This statement prints the following sequence of messages in the interaction area:

```
a is 0
a is 4
a is 4
...
a is 3
a is 3
...
```

Initially, the variable `a` is set to 0 when the act is first started. The `update` code then prints this value in the interaction area, and the body code starts. In the first statement, `a` is set to 4, and then the `turn` action starts. While the robot turns, the activity is polled by the Colbert evaluator; each time it is polled, the `update` code is evaluated, and the value of `a` is printed.

4.8.2 Colbert Evaluator and Activity States

Activity schemas, once instantiated, are called *activities*, or *acts*. Each act is a micro-task that runs in the normal 100 ms control cycle of Saphira. But unlike standard micro-tasks, acts have special facilities for robot control, including task completion, timeouts, hierarchical invocation, and signaling.

When an activity schema is invoked, it is added to the micro-task schedule. On each cycle of the scheduler, the act is given to the Colbert evaluator for evaluation. The act's *current state* is the statement that will be executed next. The evaluator evaluates statements starting from the current state, until it hits a break condition, at which point it returns control to the scheduler. Thus, each act gets a small amount of computation time on each cycle, and its current state keeps track of where execution should resume. The state of an activity may be retrieved with `sfGetTaskState` function (see Section 4.8.4).

Break conditions are designed to fit naturally into the execution cycle of acts. Typically, an act will perform a few simple computations, then invoke a robot action, behavior, or activity, and wait for its completion. In the `square` activity, both `turn` and `move` caused the act to wait. Acts will wait for two reasons:

A direct motion action, a behavior, or a sub-activity is blocking execution. Direct motion actions are discussed in Section 4.7, Direct Motion Commands; behaviors and subactivities are invoked with the `start` command, discussed in the next section. Most such actions are implicitly blocking until completion, unless the `noblock` keyword is given on invocation.

An explicit wait is issued with the `waitfor` statement. This statement waits for its condition to hold before continuing execution, for example, waits until either `a` or `b` is nonzero:

waitfor (a || b);

Besides explicit and implicit waiting conditions, an act can be suspended or interrupted by signals from other acts or itself. These special states are described in Section 4.8.3 on signaling.

To prevent an act from taking too much computation time, *single breaks* also occur in many situations. A single break causes the act to return control to the scheduler, but does not initiate a waiting condition. In the next micro-task cycle, the act continues execution at the current state. Single breaks are issued at the end of the following statements:

goto

the last statement in a `while` body

the condition of a `while` statement being false

start

signal

Single breaks ensure that an act does not evaluate large numbers of statements before returning control to the scheduler. For example, it is impossible to go through a loop without encountering at least one break. On the other hand, sequences of ordinary statements, such as variable assignments, will all execute in the same cycle, thus making act evaluation efficient.

Evaluation of acts is similar to the execution of finite-state machines. In fact, you can view activity schemas as a shorthand for finite-state machines, with special syntax for sequences, conditionals, and iterations. Figure 4-1 shows the finite-state execution model of the `patrol` activity. The states of the finite-state machine map to states of the activity; the wait conditions are represented by transition arcs that are satisfied when the wait condition holds. One of the most interesting characteristics of the Colbert language is its ability to represent finite-state machines in a compact, readable form.

The current state of an act is an integer, because acts are micro-tasks (see Section 8.5). The state is an index into the body of the act and shows where the next statement to execute is. The Saphira system maintains a mapping between these internal states and the source lines of the activity schema definition, so that it can indicate source lines during tracing. Activity states are set by sending them signals (see Section 4.8.3), and the state can be examined using the library functions `sfGetTaskState, sfTaskFinished`, and `sfTaskSuspended` (see Section 4.8.4).

4.8.3 Invocation and Signaling

Activities and behaviors are invoked with the `start` command, which has the following form:

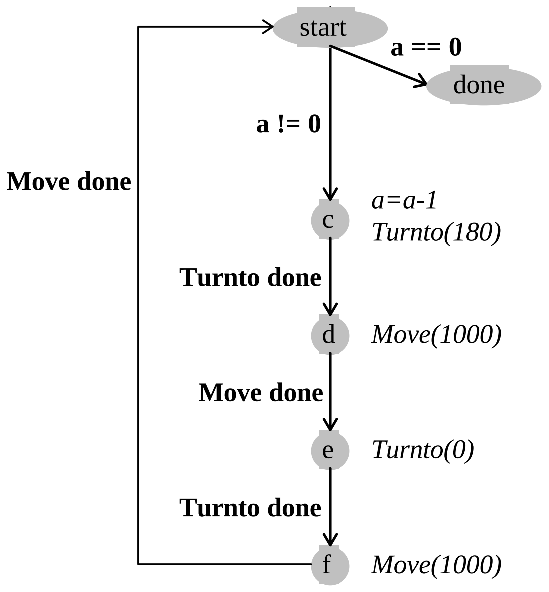
```
start <schema> [iname <symbol>] [timeout <int>] [priority <int>]
               [noblock] [suspend];
```

The `<schema>` argument is required, and is the name of the activity or behavior schema. All of the other arguments are optional, and cause modification of the invoked activity or behavior.

Table 4-5.

Command	Effect

iname <symbol>	An instance name to give the executing program. All references to the activity are through its instance name, for example, the activity can be signaled using this name. By default, the instance name is the schema name. If you start two instances of the same schema, you must give them different instance names.
timeout <int>	A timeout in 100 ms units. After this amount of time, if the activity or behavior is still executing, it is terminated.
priority <int>	Behaviors only: specifies the behavior priority.
noblock	Doesn't wait for completion of this activity or behavior, go on to the next statement of the act.
suspend	Invokes the activity or behavior, but leave it suspended. The act or behavior is added to the list of micro-tasks, but it does not start executing.



From C code, activities can be started using the function sfStartActivity (see the Saphira API reference).

Once started, an act may be signaled by other acts, by itself, or by the user through the interaction area or the Activities window. Sending an act a signal causes it to go into a special state. For example, a suspending act or behavior can be restarted by sending it a resume signal. We can illustrate the utility of signals with a monitoring example. Suppose we want to program the robot to patrol until it sees some object in front; then it should stop patrolling and approach the object. To accomplish this task, we'll set up two activities: the patrol activity of the previous example, and a supervisory activity that checks if there is something in front of the robot, and if so, approaches it (see Listing 4-4).

```
act approach()
{
int x;
start patrol(-1) timeout 300 noblock;
checking:
if (sfGetTaskState("patrol") == sfTIMEOUT || sfIsStalled())
     fail;
x = sfObjInFront();
if (x > 2000) goto checking;
suspend patrol;
move(x - 200);
succeed;
}
```

Listing 4-4. An activity that monitors another.

This activity starts off by invoking patrol with a negative argument, so it continues indefinitely. However, instead of causing the approach to wait for its completion, the patrol activity is invoked with two special parameters. The first, timeout 300, causes patrol to quit after 30 seconds (300 cycles) have elapsed. The second, noblock, allows the execution of approach to continue in parallel with patrol. The former now goes into a monitoring loop, in which it checks for objects in front, for a motor stall, and for the state of the patrol activity. If it determines that patrol has timed out, or if a motor stalls (indicating the robot ran into something immovable), then approach exits in a failure state. The activity executive keeps track of the dependencies among activities; in this case, because approach called patrol, exiting approach automatically exits patrol. Thus, if the motor stalls, all activity started by approach will be suspended.

If, on the other hand, approach determines that an object is less than 2 meters in front (by calling the perceptual routine sfObjInFront, which returns the distance to the nearest object), then it suspends the patrol activity, and moves to within 20 cm of the object. The patrol activity must be suspended, otherwise the move action will conflict with the actions being issued by patrol. After the robot moves near the object, the approach activity exits with the success state.

In this example, two activities execute concurrently, and coordination is achieved by *signals* that are sent between them. Activities can examine each others' state, and take appropriate action. As the monitoring activity, approach has the responsibility of checking the state of patrol to see if it has timed out, and also of checking for other conditions that would cause the suspension of patrol and the initiation of new activities. Finally, if approach is itself part of a larger activity, then by exiting with success or failure, it can signal other activities of its result.

Signals are sent by one of the following commands. If the optional argument is given, it is the instantiation name of the activity or behavior to signal. If not, the activity signals itself. It may seem strange for an activity to send itself some of these signals, e.g., interrupt, but it does make sense, because the effect of the signal is also communicated to the children. The only signal that can't be sent to self is `resume`, since an activity can't send a signal when it is suspended.

Table. 4-6. Colbert commands that send signals.

Command	Signal
stop [<symbol>]	sfSUSPEND
suspend [<symbol>]	sfSUSPEND
succeed [<symbol>]	sfSUCCESS
fail [<symbol>]	sfFAILURE
interrupt [<symbol>]	sfINTERRUPT
resume <symbol>	sfRESUME
remove [<symbol>]	sfREMOVE
trace [<symbol>]	
untrace [<symbol>]	

The `stop` and `suspend` commands both put the activity or behavior into a suspended state, where no evaluation is performed. The `interrupt` command is similar, but instead it signals an interrupt state, in which the activity can perform special processing before suspending. Within an activity, processing for interrupts is indicated by the special `oninterrupt` label. For example, in the code fragment in Listing 4-5, the activity will remove the follow-corridor behavior before suspending itself:

```
...
   start sfFollowCorridor(e, p) priority 2 iname follow noblock;
...
 oninterrupt:
   remove follow;
   suspend;
```

Listing 4-5. Colbert code fragment

The `resume` command resumes an activity or behavior. For activities, processing resumes at the `onresume` label. If no such label exists, the activity resumes at its first statement.

`succeed` and `fail` are special commands for stopping an activity. The activity is considered to be finished: no more processing takes place, as in the case of suspension. But other activities can check for these states to determine if the activity accomplished its job or not. When an activity "falls through" and finishes its last statement, it will enter the `sfSUCCESS` state by default.

Normally, sub-activities (those started from other activities) are not removed from the active process list when they finish. This is so that other activities can check on their progress, determine if they finished, and so on. An activity can be explicitly removed from the active process list by giving it the special state `sfREMOVE` with the `remove` command. It's a good idea to remove activities and behaviors when they're done. Top-level activities (those with no parents) are removed automatically when they finish.

The `trace` and `untrace` signals change the tracing state of activities (see Section 4.8.7).

Note that all of the signaling commands can be issued in the user interaction area, which is the normal way to start, stop, and trace activities. These commands are also used inside activities, as a means of coordinating their action.

4.8.4 Accessing Activity States

Because Colbert lacks a special construct for referring to the state of an activity; the library functions `sfGetTaskState`, `sfTaskFinished` and `sfTaskSuspended` are used:

```
int sfGetTaskState(char *iname);
int sfTaskFinished(char *iname);
int sfTaskSuspended(char *iname);
```

`sfGetTaskState` returns the state of the micro-task whose instance name is `iname`. This micro-task may be an activity, behavior, or simple micro-task. If no such micro-task exists, the result is `sfINACTIVE`. Note that the instance name is a string, because `sfGetTaskState` is a C function (see Listing 4-4 for an example).

States that are less than or equal to 10 are special states: initial, suspended, finished, or interrupted states. A micro-task that has completed its activity will be in one of the *finished* states: succeeded, failed, or timed out. The function `sfTaskFinished` returns 1 if the micro-task is in one of these states, and 2 if it is not present or is in the state `sfREMOVE`. If the micro-task is present and not finished, then `sfTaskFinished` returns 0.

A suspended micro-task has the state `sfSUSPEND` if it is suspended indefinitely, or a negative integer if it is suspended for a number of cycles. The function `sfTaskSuspended` returns 1 if a micro-task is in a suspended state, and 0 otherwise.

4.8.5 Hierarchical Invocation

Like other micro-tasks, acts can run concurrently, accomplishing different goals for the robot. The previous section showed an example of a monitoring activity running in parallel with a movement activity. Here, both activities are active and performing a certain task. In other cases, it may be useful to *sequence* a set of activities, waiting for one to complete before starting another. A parent activity controls the sequence by starting each subactivity in turn.

Colbert supports an execution model in which activities may be invoked as children of an executing activity. The technical term for this is *hierarchical task decomposition*, an important method for robot control. Consider the task of moving an object from one place to another. It's natural to decompose this into three subtasks: picking up the object, going to the destination, and dropping the object. In Colbert, we would do

```
act move_object(int dest)
{
start pickup;
start goto(dest);
start drop;
}
```

Listing 4-6. An activity with subactivities.

this using the activity in Listing 4-6.

The subactivities `pickup`, `goto`, and `drop` are executed in turn. The `move_object` activity stops at each until it finishes, then goes on to the next. This default execution model is the same as for primitive actions.

The hierarchical structure of activities is important for signals. Any signal sent to a parent is reflected to its active children. For example, if an activity is interrupted, all of its children also receive interrupt signals. This means that any behaviors or direct motion commands are suspended. Similarly, if an activity is resumed, all of its suspended children are also resumed. Hierarchical invocation makes it easy to turn sets of activities on and off.

4.8.6 Activity Window

Activities and behaviors can be controlled from the activity window, invoked from the Functions/Activities menu. The activity window shows the state of all activities and behaviors in the system.

Double-clicking on the activity or behavior will change its state from running to interrupted/suspended, or from suspended to resumed.

4.8.7 Tracing and Error Recovery

Activities can be traced by sending them the `trace` signal. For a traced activity, as each statement is evaluated, its value is printed in the user interaction area, along with the source line of the statement. The source line is an offset from the beginning of the activity schema definition.

To cut down on the amount of output, the executive prints information only when the state of a traced activity changes. For example, nothing is printed while an activity is waiting for completion of a direct motion command. Information is printed when the command finishes, and the activity goes on to the next statement.

The evaluator traps all fatal errors—all fatal user errors, for instance—in micro-tasks. An error message is printed, and the offending command is exited. In the case of an error caused by a statement in an activity, the line number of the activity (relative to the top of the activity) is printed, and the activity is suspended.

4.9 Colbert Language

The Colbert language is C-like, in that it has a syntax that is close to that of ANSI C. It has many but not all of C's expression and statement constructs, and additional constructs that are specific to Saphira, such as the direct motion commands, and the invocation of activities and behaviors.

Colbert is *not* meant to be a replacement for writing code in C. You cannot define new C functions in Colbert (*acts* are like functions, but are executed differently). For any complicated computation, we recommend writing a C function, compiling it into a shared object, and then loading it into the evaluator (see Section 4.10).

Most of the Saphira library functions, variables, and structures are available in Colbert. Few C library functions (such as the trigonometric functions) exist, but these can easily be added by the user via shared object files.

4.9.1 Major Changes from ANSI C

The typing system is slightly different. The basic types are `int`, `float`, `void`, and `string` (essentially, `char *`). No `double` or `char` type is available. Structures are permitted, but only by explicitly importing them from a native C shared-object file.

No arrays or array operators exist.

The type `double` is not available; instead, all floating-point numbers are single precision (`float`).

No `typedef` operator exists, and new structures cannot be defined in Colbert; they must be imported from native C object files.

The following operators are not defined: `?:`, `op=`, >>, <<, ++, `--`, `,`.

Explicit type casting is not permitted (although implicit casting is performed).

The `for` and `switch` statements are not defined.

Variables may not be initialized when defined.

No embedded assignments are allowed, e.g., `if ((x = a) > 2) { … }`

New functions are not defined in Colbert but may be imported from native C object files.

Only a few standard C library functions are initially available, although others can be made available by telling the evaluator about them with `sfAddEvalFn` (see below). This and the other `sfAddEvalXXX` functions are available only in C code, so you must compile and load a shared object file to link in C library functions.

Some of these limitations may be removed in future releases. As Colbert provides for dynamic linking of C object files, these restrictions aren't absolute: Native C functions can be loaded. For example, to reference an array, you can define a C function that takes an offset and array as its argument, and returns the array element.

4.9.2 Comments

Standard C comment syntax is used:

```
/* a comment */
```

4.9.3 Keywords

In addition to many of the ANSI C keywords, Colbert defines several new keywords that cannot be used as variables, labels, or other names. A list of these names follows:`act`, `behavior`, `fail`, `halt`, `help`, `iname`, `interrupt`, `load`, `move`, `noblock`, `priority`, `remove`, `resume`, `rotate`, `speed`, `string`, `succeed`, `suspend`, `timeout`, `trace`, `turn`, `turnto`, `untrace`, `update`, and `waitfor`.

4.9.4 Types

The type system of Colbert is simplified from ANSI C. Table 4-7 shows the predefined types.

Table 4-7. predefined types in Colbert.

Type	C index
`int`	`sfINT`
`float`	`sfFLOAT`
`void`	`sfVOID`
`string`	`sfSTRING`
`act`	`sfACTIVITY`
`behavior`	`sfBEHAVIOR`
`void *`	`sfPTR`

The first six are basic types. Note that the type `double` doesn't exist; all floating-point numbers in Colbert are single precision. This decision was made to keep all types the same size on 32-bit machines. For the same reason, Colbert has neither a `char` or `bitfield` type. Users can always provide access to C data with non-Colbert types by writing native C functions to convert them to Colbert types.

`string` is equivalent to `(char *)`, but is atomic, i.e., `*str` is illegal if `str` is a string. The last, `sfPTR`, is a convenience definition for a generic pointer.

The `sfACTIVITY` and `sfBEHAVIOR` types are special basic types for activities and behaviors, similar to functions. Activity schemas are defined with the `act` keyword (see Section 4.9.12). Behavior types are not input by Colbert; instead, behaviors are defined using the behavior compiler (see Chapter 4), and made available in Colbert with the `sfAddEvalVar` or `sfAddEvalConst` functions (see Section 4.10.3).

Function pointer types exist, but the user has no access to them from Colbert, so they are omitted here.

It is often necessary to refer to Colbert types from C code; for example, when defining C functions for Colbert. All types in Colbert have a corresponding C index, an integer, so they can be referred to from standard C code. For examples of the use of these indices, see Sections 4.10.2, 4.10.3, and 4.10.4.

4.9.5 Expressions

Expressions use ANSI C syntax. The following are valid expressions:

```
2.3
"a string"
1 + 4.3
(2 > a) || !b
fn(arg1, arg2)
exp.slot
exp->slot
*exp
&exp
sizeof(type)
```

Expressions typed at the command window, and followed by a semicolon, are evaluated and the result printed, with the type of the result given in parentheses.

Pointer arithmetic is not implemented.

The comma (`,`) and question mark (`?:`)operators are not implemented.

The type of an expression is determined by the type of its components. Colbert performs implicit type casting in the following cases.

In arithmetic operations and comparisons, all numbers are converted to floating-point if any one of the components is floating-point. Pointers are converted to integers.

In logical operations, floating point numbers and pointers are cast to integers.

In assignments, the value to be assigned is cast to the type of the variable being assigned.

In function evaluation, the arguments are cast according to the function prototype.

4.9.6 Variables

Variables are defined using ANSI C syntax. The type of the variable is given by one of the five predefined types, or by a type imported with `sfAddEvalStruct`. Pointers and pointers to pointers, and so on, are legal; but no special modifiers such as `const` or `extern` are permitted.

```
int a;
ptr tonowhere;
float **f;
robot *r;
```

Variables can be declared at the beginning of acts, and at the top level of a Colbert source file. Top-level variables have global extent and are accessible by all Colbert activities. Variables declared within an act are local to that act and function as static variables. Each invocation of an activity schema gets its own copy of the local variables.

All variables are initialized to 0.

Colbert variables also can be declared by linking them to a native C variable with the `sfAddEvalVar` function. These variables need not have an explicit Colbert declaration, although it is legal to give them one. The value of the Colbert variable reflects the value of the C variable.

4.9.7 Statement Grouping

Statements are grouped by using curly braces, as in this example:

```
{ <stmt1> <stmt2> <stmt3> }
```

Grouping is useful in Colbert-specific forms, such as `update` in act definition, that take only a single statement.

The empty statements `;` and `{}` are valid statements.

4.9.8 Conditional Statements

Colbert uses the standard `if` statement for conditionals:

```
if (<c_exp>) <stmts> [ else <stmts> ]
```

Colbert requires a special form for waiting until a certain condition holds:

```
waitfor <c_exp> [timeout <n>];
```

This statement effectively causes the Colbert executive to suspend further sequential execution of the act until the condition `c_exp` becomes nonzero. On each cycle of the scheduler, `c_exp` is evaluated, and if it is 0, control states at the `waitfor` statement. The `waitfor` statement without the optional `timeout` parameter is equivalent to this sequence:

```
while (!c_exp) {}
```

The timeout parameter is very handy for preventing blocks in an activity. After `n` cycles, if the condition still has not been satisfied, the `waitfor` completes, and execution continues with the next statement.

Unlike suspension, `waitfor` does not affect any child activities, which keep executing normally.

4.9.9 Iteration and Branching Statements

The only iteration construct in Colbert is the `while` statement:

```
while (<c_exp>) <stmts>
```

`c_exp` is evaluated, and if false, a single break occurs, so control returns to the scheduler. On the next cycle, execution continues with the statement after the `while`.

Control may also be transferred in an act using the `goto` statement and labels, as in this sequence:

```
<label>:
goto <label>;
```

Labels may occur *only* at the top level of an activity schema. `goto`'s cause a single break when they are executed, so that control returns to the scheduler.

4.9.10 Assignment Statements

Values may be assigned to any expression that represents a storage location. This includes variables and locations described by pointers and structure members. Implicit type conversion is made to convert the value to the type of the storage location. The following are valid assignment operations:

```
point p;
int a;
int *b;
float *c;
a = 2;
b = &a;
*b = 3;
p.x = 1.0;
c = &p.x;
*c = *b;
```

If a Colbert variable is linked to a native C variable by the `sfAddEvalVar` function, then changing the value of the Colbert variable will also change the value of the linked C variable.

4.9.11 Function Statements

Function expressions are also considered as statements.

4.9.12 Activity Schemas

Activity schemas are defined using the special keyword `act`. They are similar to function definitions, but are interpreted by the Colbert executive as a special type of micro-task.

The activity name is any symbol. The symbol cannot be declared as a variable or function. If the name was previously assigned to an activity schema, the old definition is replaced by the new one. Note that any instances of the schema running as micro-tasks are unaffected by the redefinition; you must re-invoke the activity schema to get the new definition.

The activity schema takes a set of parameters, which are variables local to the activity. If no parameters are provided, the parentheses may be omitted.

Optional local variables are declared only at the beginning of the activity schema.

An optional update block is a statement that is evaluated every time the activity is invoked by the scheduler. Typically, the update block is used to set the values of variables to reflect a change in the state of the robot. `goto`, labels, and iteration are illegal in the update statement.

Body statements are executed in accordance with the finite-state machine semantics described in Section 4.8.2. Labels are allowed only at the top level of body statements. Some special labels indicate places to start execution on exceptional conditions.

Table 4-8.

Command	Description
oninterrupt	Branch location for an interrupt signal.
onresume	Branch location when an activity resumes because it was sent an `sfRESUME` signal.

4.9.13 Direct actions

Direct actions are statements that result in robot motion (see Table 4-9). These statements may appear anywhere a statement is allowed in an activity schema. The general form is:

```
command(int arg) [timeout n] [noblock];
```

where `timeout n` specifies a time in 100 ms increments for the command to complete, and `noblock` means that the command will be executed without blocking; that is, control will continue with the next statement. Some motion commands are implicitly non-blocking: `speed` and `rotate`.

Table 4-9. Direct action statements in Colbert.

Command	Effect
move(int mm);	Moves the robot `mm` millimeters forward (positive) or backwards (negative). Blocking.
turn(int deg);	Turns the robot `deg` degrees clockwise (negative) or counter-clockwise (positive) degrees from the current heading. Blocking.
turnto(int deg);	Turns the robot to the heading `deg` degrees. Positive values are counterclockwise, negative values are clockwise. Blocking.
speed(int mms);	Moves the robot at a speed of `mms` millimeters per second forward (positive) or backwards (negative). Non-blocking.
rotate(int degs);	Moves the robot at a rotational speed of `degs` degrees per second counter-clockwise (positive) or clockwise (negative). Non-blocking.
halt;	Halts all direct motion commands.

4.9.14 Activity and Behavior Invocation and Signaling

Activities and behaviors are started with the `start` statement (see Listing 4-7)

```
start <aname> [([c_exp]*)] [ noblock ]
                          [ iname <symbol> ]
                          [ priority <int> ]
                          [ suspend ]
                          [ timeout <int> ] ;
```

Listing 4-7. In Colbert, activities and behaviors begin with the `start` statement.

The activity should be invoked with as many arguments as in its definition. If there are no arguments, then the argument's parentheses may be omitted. All of the optional keywords can occur in any order. `noblock` causes execution to continue with the next statement, after a single break and without waiting for the activity to complete. The activity can be given an instance name so that other activities can refer to it; by default, this name is its schema name. If another activity or behavior has this instance name, an error is signaled. The `priority` keyword is only for behaviors, which compete for control on the basis of their priorities. A `timeout` specifies the maximum number of 100 ms cycles the activity or behavior will be allowed to execute. If present, `suspend` invokes the activity or behavior but leaves it in a suspended state, pending a `resume` signal.

Note that the activity schema must be defined with the `act` command before the `start` command is executed, or an error will result. Currently the only way to invoke an activity is to use the activity schema name. For behaviors, the behavior schema must be defined with the behavior compiler and loaded into Colbert, and its name made available with `sfAddEvalVar` or `sfAddEvalConst`.

Activities and behaviors are sent signals with the `signal` statement:

```
signal [ <symbol> ];
```

The optional argument specifies the instantiation name of an activity or behavior to signal. If no such activity or behavior exists, an error is issued. The available signals are shown in Table 4-10.

Table 4-10. Signals for activities and behaviors in Colbert.

Command	Signal	Description
stop [<symbol>]	sfSUSPEND	Suspends execution of the activity. Execution can be resumed with the `resume` command.
suspend [<symbol>]	sfSUSPEND	Same as `stop`.
succeed [<symbol>]	sfSUCCESS	Causes the activity to finish in the `sfSUCCESS` state.
fail [<symbol>]	sfFAILURE	Causes the activity to finish in the `sfFAILURE` state.
interrupt [<symbol>]	sfINTERRUPT	Interrupts the activity, branching to the `oninterrupt` label, if it exists. If not, the activity stays in the `sfINTERRUPT` state, and no further execution occurs.
resume <symbol>	sfRESUME	Resumes a suspended or interrupted activity. If the `onresume` label exists, starts at this point; otherwise starts at the beginning of the activity.
remove [<symbol>]	sfREMOVE	Causes the scheduler to reap the activity.

trace [<symbol>]		Traces the activity.
untrace [<symbol>]		Untraces the activity.

4.10 Loading Native C Code

Native object files can be dynamically loaded into the execution environment, giving access to Saphira internals and new C functions. Because Colbert has only a limited implementation of C, in many cases you must load C object files to accomplish a task. For example, the only way to define new structures is to load in an appropriate C object file.

4.10.1 Format of Native C Files

In general, a C source file will contain user-defined functions and variables, and a special function, `sfLoadInit,` that will be called when the file is loaded. `SfLoadInit` will contain calls to the `sfAddEvalXXX` functions, which will make native C functions and variables defined in the file (or already loaded in the system) available to Colbert. Listing 4-8 shows an example load file (in `handler/src/apps/testload.c`).

```
/*
 * test load file for dynamic loading
 */
#include "saphira.h"

int nopen = 0;

int myfn(int a)
{
  return a+1;
}

struct mystruct
{
  int a;
  float b;
  void *c;
} m;

int ind_mystruct;

EXPORT void sfLoadInit(void)  /* evaluated on load */
{
  float a = 1.3;
  a = sqrt(a);
  printf("Opened! %d %f\n", nopen++, a);
  sfSMessage("Opened: %d", nopen);
  sfAddEvalFn("myfn", myfn, sfINT, 1, sfINT);
  sfAddEvalConst("sfFollowCorridor", sfBEHAVIOR, sfFollowCorridor);
  sfAddEvalConst("sfLEFT",  sfINT, 0);
  sfAddEvalVar("sfCurrentEnvironment", sfPTR,
                (fvalue *)&sfCurrentEnvironment);

  ind_mystruct = sfAddEvalStruct("mystruct", sizeof(struct
                   mystruct), (char *)&m, 3,
                                   "a", &m.a, sfINT,
                                   "b", &m.b, sfFLOAT,
                                   "c", &m.c, sfPTR);
  sfAddEvalVar("m", ind_mystruct, (fvalue *)&m);
}
```

Listing 4-8. Example load file in Colbert.

The function `sfLoadInit`, if present, is invoked when the object file is loaded into Saphira. In this case, it prints a message, then makes a structure, a variable, a function, a behavior, a constant, and another variable visible to Colbert. Details on how to make C functions and variables available in Colbert are contained in the next few sections.

When a file is unloaded or reloaded, the function `sfLoadExit()` is called to help clean up anything that could cause problems. For example, any activities that access C functions or variables defined in the file should be removed, or they will cause an error.

4.10.2 Making Native C Functions Accessible

Native C functions, including Saphira library functions, are made accessible in Colbert with the `sfAddEvalFn` function. `sfAddEvalFn` is *not* callable from Colbert, because Colbert has no way to access the underlying C environment. It must always be compiled and loaded from a shared object file, usually as a call in the `sfLoadInit` function (see the example in the previous section). The format of `sfAddEvalFn` is:

```
sfAddEvalFn(char *name, void *fn, int rtype, int nargs, ...)
```

`name` is the name of the function as seen by Colbert. `fn` is a function pointer to the C function being made available. The return type, `rtype`, is the C index of a Colbert type (see Section 4.9.3). The predefined types are shown in Table 4-11.

Table 4-11. Predefined function types in Colbert.

Type	C index
`int`	`sfINT`
`float`	`sfFLOAT`
`void`	`sfVOID`
`string`	`sfSTRING`
`act`	`sfACTIVITY`
`behavior`	`sfBEHAVIOR`
`void *`	`sfPTR`

In addition, pointers to types can be defined with the function `sfTypeRef(int type)`. For example, to define a pointer to an integer, use:

```
sfTypeRef(sfINT)
```

The function `sfTypeDeref` performs the inverse operation, giving the type of the reference of a pointer; but this is less useful in defining argument types.

`nargs` is the number of arguments of the function, currently a maximum of seven. If the function takes a variable number of arguments, then use a negative number here, where `|nargs|` is the number of required arguments of the function. Each argument to the function is described by the C index of its type. For example, the library function void `sfSMessage(char *format, …)` is made accessible with this command:

```
sfAddEvalFn("sfSMessage", sfSMessage, sfVOID, -1, sfSTRING);
```

`sfSMessage` has one required argument, a string, and returns `void`.

4.10.3 Making Native C Variables Accessible

Native C variables in user code and the Saphira library are made accessible in Colbert with the `sfAddEvalVar` function. This function can only be called from loaded C object files, not from the Colbert evaluator. It must always be compiled and loaded from a shared object file, usually as a call in the `sfLoadInit` function (see the example in the Section 4.10.1). The format of `sfAddEvalVar` is:

```
sfAddEvalVar(char *name, int type, (fvalue *)&cvar);
```

`name` is the name of the variable as seen by Colbert. `cvar` is the variable being made available. Note that a pointer to the variable is required, and it is cast to the type `fvalue *`. This is so that the Colbert executive can change the value of the variable. The type of the variable, `type`, is the C index of a Colbert type (see Section 4.9.3). The predefined variable types are shown in Table 4-12:

Table 4-12. Predefined variable types in Colbert.

Type	C index
`int`	`sfINT`
`float`	`sfFLOAT`
`void`	`sfVOID`
`string`	`sfSTRING`
`act`	`sfACTIVITY`
`behavior`	`sfBEHAVIOR`
`void *`	`sfPTR`

In addition, pointers to types can be defined with the function `sfTypeRef(int type)`. For example, to define a pointer to an integer, use:

```
sfTypeRef(sfINT)
```

The function `sfTypeDeref` performs the inverse operation, giving the type of the reference of a pointer; but this is less useful in defining argument types.

A C variable made available in Colbert is both accessed and changed by the appropriate Colbert expression. For example, `sfRobot` is available in the `bin/saphira` executable, having been defined by:

```
sfAddEvalVar("sfRobot",  sfSrobot, (fvalue *)&sfRobot);
```

The type `sfSrobot` is also defined (see the next section). The robot's current global position is available in Colbert as the members `sfRobot.ax`, `sfRobot.ay`, and `sfRobot.ath`. For example, the following Colbert statement will increment the global *x* position by 1 meter (this is just an example; the recommended way to change the global position is with `sfMoveRobot`):

```
sfRobot.ax = sfRobot.ax + 1000.0;
```

Besides variables, *constants* can be defined in Colbert, with `sfAddEvalConst`. The format is similar to that for adding variables:

```
sfAddEvalConst(char *name, int type, (fvalue)val);
```

where `val` is a constant expression. For example, these are some of the predefined constant loaded into `bin/saphira`:

```
sfAddEvalConst("sfConstantVelocity", sfBEHAVIOR,
                sfConstantVelocity);
sfAddEvalConst("sfVSLOWLY", sfFLOAT, 3.0);
sfAddEvalConst("sfSLOWLY", sfFLOAT, 4.0);
```

The only way to start a behavior from Colbert is to define it in C, and then make it accessible with `sfAddEvalConst` or `sfAddEvalVar`.

4.10.4 Making Native C Structures Accessible

Native C structures are made accessible in Colbert with the `sfAddEvalStruct` function. New structures cannot be defined in Colbert; they must already exist in a loaded C object file. `sfAddEvalStruct` can only be called from loaded C object files, not from the Colbert evaluator. It must always be compiled and loaded from a shared object file, usually as a call in the `sfLoadInit` function (see the example in the Section 4.10.1). The format of `sfAddEvalStruct` is:

```
sfAddEvalStruct(char *name, int size, (char *)&s, int num, ...);
```

`name` is the name of the structure as seen by Colbert. `size` is the size in bytes of the structure. `s` is an instance of the structure being made accessible; here a pointer to it is passed to the function. `num` is the number of members in the structure. The rest of the arguments are triplets, each defining one structure member. The format of the triplets is:

```
(char *)slotname, &s.slot, int type
```

where `slotname` is the Colbert name for the member, `s.slot` is the instance structure member, and `type` is the C index of the member type. The available types appear in Table 4-13.

Table 4-13. Predefined Structure types in Colbert.

Type	C Index
`int`	`sfINT`
`float`	`sfFLOAT`
`void`	`sfVOID`
`string`	`sfSTRING`
`act`	`sfACTIVITY`
`behavior`	`sfBEHAVIOR`
`void *`	`sfPTR`

In addition, pointers to types can be defined with the function `sfTypeRef(int type)`. For example, to define a pointer to an integer, use:

```
sfTypeRef(sfINT)
```

The function `sfTypeDeref` performs the inverse operation, giving the type of the reference of a pointer; but this is less useful in defining argument types.
For example, the `sfRobot` structure is defined as shown in Listing 4-9.

```
      int sfSrobot;
          robot r;
          sfSrobot = sfAddEvalStruct(
             "robot", sizeof(robot), (char *)&r, 22,
             "x", &r.x, sfFLOAT,    /* local coords */
             "y", &r.y, sfFLOAT,
             "th", &r.th, sfFLOAT,
             "ax", &r.ax, sfFLOAT,  /* global coords */
             "ay", &r.ay, sfFLOAT,
             "ath", &r.ath, sfFLOAT,
             "control", &r.control, sfFLOAT, /* heading control */
             ...
             "status", &r.status, sfINT, /* status int here */
```

Listing 4-9. The Colbert structure `sfRobot`.

The structure name in Colbert is `robot`, and it has 22 available members, each with its own name and type. Not all members need be declared to Colbert. The `sfAddEvalStruct` returns a new type index, which is stored in the C variable `sfSrobot`. This index is used in making the definition of the Saphira variable `sfRobot` available in Colbert. As with the other `sfAddEvalXXX` functions, `sfAddEvalStruct` must be compiled into a shared object file, and then loaded into Colbert.

C indexes for pointer types are constructed using the functions `sfTypeRef` and `sfTypeDeref`. For example, in C code, to get a type index for a pointer to the robot structure, use:

```
sfTypeRef(sfSrobot)
```

The size of a structure is returned in Colbert by the `sizeof(typename)` function. The currently-loaded structures are printed with the `help structs` command.

The `robot` and `point` types are predefined in the `bin/saphira` executable.

4.10.5 Compiling and Loading C Files

Chapter 6 has more detailed information about the particulars of compiling native C files and making them into shared object files. Under UNIX, the object files must be converted into a shareable object file (.so). The shareable object file is loaded with the load command, as in this example:

```
load ..../testload.so
```

A dynamically loaded file may be recompiled and reloaded at any point.

Under MS Windows, C code is compiled into a Dynamic Link Library (DLL). The DLL is then loaded into Saphira, again with the `load` command. DLLs cannot be relinked or reloaded, unless they are first unloaded. From Colbert, use the `unload` command to unload a DLL that you are going to relink.

5 Behavioral Control

You can control robot motion in two different ways. The *direct motion* commands were introduced in Sections 2.2.2 and 4.7. Direct motion control is appropriate for moving the robot through simple sequences of action, e.g., the `BumpAndGo` activity backs and turns the robot when it bumps into something. But in certain cases, the trajectory of the robot must satisfy complicated demands from the task and various maintenance policies. For example, in navigating from one room to another in an office environment, the trajectory is defined in large part 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.

One approach to complex control is to decompose the problem into a set of small actions to accomplish particular goals, which can then be combined into a more comprehensive control strategy. Each such small action, with its associated goal, is called a *behavior*. A behavior looks at some set of sensor information and outputs a desired action, based on its goal. For example, an obstacle-avoidance behavior might look at the current sonar readings and decide to slow down or turn the robot.

As of Saphira 6.x, behaviors are treated as a type of activity, and are invoked and disabled using the same commands as activities. In particular, while the behavior window still exists for displaying information about behaviors, you cannot turn a behavior on and off from this window. Instead, use the Activities window. Also, we recommend starting and controlling behaviors using Colbert, which provides a convenient interface to behavior activation and a uniform view of behaviors, direct actions, and activities.

5.1 Behaviors and Fuzzy Control

Every behavioral-control scheme must decide on representations for the output action and must include a method for arbitrating among competing outputs, when several behaviors want to control the robot. In Saphira, we use *fuzzy control rules* to define output actions, and competing outputs are merged based on priorities and degree of activation of a behavior. A fuzzy control rule maps 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 mentioned here.

This section describes how to define and execute behaviors in the Saphira system. Behaviors are specified using the *behavior grammar*, which simplifies the task of writing behavior control rules. Specifications in the behavior grammar are translated into C code by the `bin/bgram` program, and the resulting source code can be compiled and loaded into Saphira.

The Saphira library also has a number of precompiled behaviors available for obstacle avoidance and goal seeking (see Section 5.9).

5.2 Invoking Behaviors

We introduce behaviors with an example invocation of a predefined behavior. You can invoke behaviors in two ways: with the Colbert `start` command from the interaction area or an activity, or with the `sfStartBehavior` function from C code.

The behavior `sfGoToPos` moves the robot to a goal position. It takes three arguments: the speed at which the robot is to move (in mm/sec), a point artifact representing the goal position, and a success radius (in mm). In the interaction area, type:

```
point *goal;
goal = sfCreateLocalPoint(1000, 0, 0);
sfAddPoint(goal);
start sfGoToPos(200, goal, 100);
```

The first two statements create a point artifact situated 1 meter in front of the robot. The `sfAddPoint` function adds it to the pointlist, so that its position is updated as the robot moves. Finally, the start command invokes the `sfGoToPos` behavior at 200 mm/sec to the goal point, with success defined as being at most 100 mm from the goal. If the robot is connected, it will start to move towards the goal point, and stop when it gets near.

5.2.1 Presenting Behaviors to Colbert

Behaviors are defined using the behavior compiler, which generates a C code file that can be compiled and loaded into Saphira (see Section 5.4). Behaviors are represented as C structures and can be presented as constants to Colbert. If myBehavior is a behavior, then the following construct will make `myBehavior` available to the Colbert evaluator:

```
sfAddEvalConst("myBehavior", sfBEHAVIOR, myBehavior)
```

The `sfAddEvalConst` call should be included in an `sfLoadInit` function in the C file.

5.2.2 Invoking Behaviors from C Code

From C code, you can invoke behaviors using the `sfStartBehavior` function. The arguments are similar to those of the `start command.`

```
sfprocess sfStartBehavior(behavior *b, char *iname, int timeout,
                          int priority, int suspended, ...)
```

The first argument of the `sfStartBehavior` function is a pointer to the behavior structure, as defined below. The second is the instance name of the behavior. In Saphira 6.x, behaviors are also micro-tasks and so are referred to by their instance name. The `timeout` value is the number of 100 ms cycles the behavior will run; use 0 for no timeout.

The next argument 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 `suspended` argument is 0 if the closure is started in an active state, and 1 if it is suspended. A closure that is suspended is present but not active and does not affect the robot's movements. The suspended state of a behavior can be changed by using Colbert signals (see Section 4.8.3), or with the library function `sfSetTaskState` (see Section 8.5.3).

The remaining arguments to this function set up the parameters of the closure. They must be the same number and have the same type as the parameters specified in the behavior definition. Listing 5-1 shows an example invocation of the predefined behavior `sfKeepOff.`

```
sfStartBehavior(sfKeepOff, “keep off”,  /* instance name */
                           0,           /* no timeout */
                           1,           /* priority */
                           0,           /* no suspension */
                           100.0,       /* caution speed */
                           0.4);        /* sensitivity */
```

Listing 5-1. Invocation of the predefined behavior `sfKeepOff`.

5.3 *Behaviors as Activities*

Behaviors are a special type of activity. They have special properties, such as a priority and various internal state variables (`goal`, `turn`, and `velocity` activity). On the other hand, they are similar to basic actions in that they cannot invoke subactivities.

Behaviors can be suspended or killed by sending them signals, using the task signal facility (see Section 4.8.3). All of the behavior functions from version 5.3, such as `sfInitBehavior` and `sfKillBehavior`, are not available in 6.x. Active behaviors appear in the Function/Activities window. Just as with other activities, they can be interrupted and resumed by double-clicking in this window.

Conceptually, there are two types of activities: those that achieve some goal and those that act to maintain a state. Goal-achieving behaviors can terminate on their own, like direct actions. They do this by setting the goal state in their Activity section (see Section 5.8.5). A behavior whose goal state is greater than 0.8 is considered to be successful and is terminated by the behavior executive.

Like direct actions, behaviors may be started in either blocking or non-blocking mode from within an activity. Blocking mode is generally useful only with goal-achieving behaviors, which will terminate when their goal state is sufficiently fulfilled. Blocking mode is useful for sequencing goal-achieving behaviors. Non-blocking mode is useful for starting a set of behaviors executing concurrently.

When an activity is suspended, all of the behaviors it or its subactivities invoke are also suspended. On resumption, these behaviors are resumed. If an activity terminates, all of its behaviors are terminated. One common mistake in invoking behaviors from activities occurs when the activity terminates unintentionally. For example, the following activity, will start the three behaviors, then succeed and promptly terminate all three:

```
act startb()
{
   start b1() noblock;
   start b2() noblock;
   start b3() noblock;
}
```

The net effect is that the behaviors never really get executed.

5.4 Behavior Grammar

The *behavior grammar* is a convenient syntax for defining behaviors. The BNF for the grammar is given below. For reference, Listing 5-2 provides 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_right = up_straight(phi, 10, 50);
too_left  = straight_down(phi, -50, -10);
near_goal = straight_down(dist, radius, radius*2);
Activity
Turn Not near_goal
Speed Not near_goal
Goal near_goal
EndBehavior
```

Listing 5-2. Example behavior grammar; the sequence sends a robot toward a goal position.

Sample behaviors can be found in the file handler `/src/basic/behavior.beh`. You can refer to these for reference and ideas on how to write behaviors.

5.5 Behavior Grammar in BNF

Listing 5-3 provides 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]
```

Listing 5-3. Complete rules for the behavior grammar, in the form accepted by the YACC or BISON parsers

5.6 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 `bin/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 the bump-and-go procedure finishes, it resets the flag to 1, and the behaviors resume control:

```
int sfBehaviorControl
int sfHasDCHead
```

A value of 0 for `sfBehaviorControl` suppresses 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.

`sfHasDCHead` controls whether the `DCHEAD` or `DHEAD` commands are used to control robot turning. `DCHEAD` was implemented on PSOS 4.3 and later. Set `sfHasDCHead` to 1 to use `DCHEAD` commands; it can result in smoother and more responsive turning. The default value is 0.

5.7 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.

5.7.1 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; it interpolates linearly between them (see Figure 5-1). `down_straight` is the opposite: values below the start return 1.0; those above, 0.0. Intermediate ones are linearly interpolated.

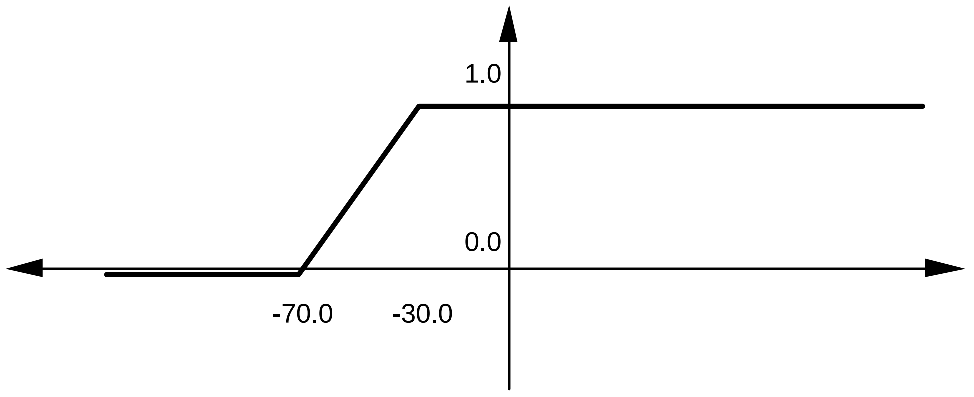


Figure 5-1. The `straight-up` function.

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.

5.7.2 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)
```

5.8 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.

5.8.1 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;
```

Listing 5-4.

5.8.2 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.

5.8.3 `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.

5.8.4 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;
```

Listing 5-5.

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.

5.8.5 Activity

The activity section of a behavior defines how it operates in the larger context of other behaviors. The activity section comprises four fuzzy state variables, given in Table 5-1.

Table 5-1. Behavior state variables and their definitions.

Variable	Effect
`Turn`	Controls the rotation channel of the robot. If it is 0.0, this behavior has no effect on robot rotation. If it is 1.0, then it competes fully with other behaviors of the same priority for control of rotation. Default is 0.0
`Speed`	Controls the velocity channel of the robot. If it is 0.0, this behavior has no effect on the speed of the robot. If it is 1.0, then it competes fully with other behaviors of the same priority for control of the robot’s speed. Default is 0.0.
`Goal`	Indicates whether the behavior is achieving a goal. A value of 0.0 indicates no goal achievement. A value greater than 0.8 signals that the behavior has achieved its goal.
`Progress`	Indicates whether the behavior is successfully moving towards a goal. Not currently used.

The `Turn` and `Speed` state variables control how much effect the behavior will have on these actions of the robot, relative to other behaviors of the same priority. The `Goal` variable is used to determine whether the behavior has succeeded in achieving its goal. When the `Goal` is greater than 0.8, the behavior is considered to be successful and terminates in the state `SUCCESS`.

5.8.6 Behavior schema

A complete behavior schema is a structure combining its rules, `init`, and `update` functions (the activity section is part of the `update` function). The rules can be included directly in the definition; Listing 5-6 shows the 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}
                             }
               };
```

Figure 5-6. The behavior schema for the constant velocity function.

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.

5.9 Predefined Saphira Behaviors

Saphira has a number of predefined behaviors for obstacle avoidance and goal-directed movement. Most of the complexity of these behaviors are 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 `handler/src/basic/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 `sfStartBehavior` function. See the sample behavior definition file `behavior.beh` for examples.

This sequence sets the velocity setpoint on the robot server to its first parameter, an integer in millimeters per second:

```
behavior *sfConstantVelocity
```

This one sets the velocity setpoint to zero. It doesn't permit parameters:

```
behavior *sfStop
```

This structure slows and turns the robot sharply to avoid immediate obstacles:

```
behavior *sfAvoidCollision
```

It takes four parameters, which are listed in Table 5.2. Additionally, the default turn direction, when it is completely blocked, is given by the global variable `sfPreferredTurnDi`*r*, 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 5.2. Behavior parameters for avoiding a collision.

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`.

This structure slows the robot sharply to avoid immediate obstacles:

```
behavior *sfStopCollision
```

This behavior differs from `sfAvoidCollisio`*n* in that it doesn't turn the robot; another behavior must do that. The structure takes three parameters, which are listed in Table 7-4.

Table 5.3. Behavior parameters for stopping a collision.

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).

This structure gently steers the robot around and away from distant objects:

```
behavior *sfKeepOff
```

The behavior takes two parameters and uses the global variable `sfPreferredTurnDir`, described in Table 5.4. The priority for `sfKeepOff` should always be less than (higher priority number) than that for `sfAvoidObstacle` when they are invoked together.

Table 5.4. 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`.

This structure sends the robot to a given point:

```
behavior *sfGoToPos
```

It takes three parameters, described in Table 5.5.

Table 5.5. 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.

To move the robot near a given goal position and point the robot towards the goal position, use the following structure:

```
behavior *stAttendAtPos
```

It takes three parameters, described in Table 5.6.

Table 5.6. 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.

Use this structure to tell the robot to follow a lane, as represented by a lane artifact:

```
behavior *sfFollow
```

The lane structure is a directed point with a width, although the width is ignored in this behavior, because explicit parameters for the latitude the robot are allowed in the lane. A goal point represents 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 5.7. This behavior sets the `sfPreferredTurnDir` variable according to how the robot is misaligned with the lane.

Table 5.7. 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 centered/aligned in the right direction; 0.0: direction overrides; 1.0: center overrides.

This structure tells the robot to follow a corridor, as represented by a corridor artifact:

```
behavior *sfFollowCorridor
```

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. A goal point, represents 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 5.8. This behavior sets the `sfPreferredTurnDir` variable depending on how the robot is misaligned with the corridor.

Table 5.8. 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.

To tell the robot to go in a doorway, as represented by a door artifact, use this sequence:

```
behavior *sfFollowDoor
```

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 5.9. This behavior sets the `sfPreferredTurnDir` variable depending on how the robot is misaligned with the lane through the doorway.

Table 5.9. 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.

Use this structure to turn the robot to point in the direction of a goal position:

```
behavior *sfTurnTo
```

The robot always turns in the direction that makes the smallest turn. Table 5.10 shows the p

Table 5.10. `TurnTo` 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 degrees). 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.speed, 2.0 is fast.

6 Creating Load Files and Clients

This chapter describes how to create Saphira clients, and provides examples of the three types of clients:

Loadable clients. Loadable clients are created by loading files into a base system, typically `bin/saphira`. The files may be Colbert language interpreted files, or compiled C code in shared object files.

Stand-alone clients. Stand-alone clients are created by compiling C code and linking it with the Saphira libraries to create a stand-alone executable.

Foreign clients. These clients are called from a program written in another language, e.g., PROLOG or LISP. The foreign language executable loads and executes routines from the Saphira libraries and compiled user C code.

The chosen method is up to the user. With Colbert, the user stays within an interactive debugging environment and can debug and re-execute procedures without the burdensome debug-recompile-reload-re-execute cycle. Colbert sources and shared object files are also much easier to distribute and share than C source for clients. So, the interactive method is the one we recommend for most development tasks. For mature applications, it may be useful to create a new client, with all user functions preloaded.

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 routines in C or C++ and compiles them into object files, then these object files, together with the Saphira libraries, are loaded into the LISP or PROLOG system.

C or C++ programs can be compiled into object files using standard compilers, such as `gcc` or MS Visual C++. The header files in `handler/include` 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 either a shared object file, or an executable client. Shared object files are loaded into Colbert, and clients are stand-alone systems for controlling the robot. User clients may also invoke the Colbert evaluator; for instance, the sample client `bin/saphira` calls the evaluator as a micro-task.

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 and shared object files 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 and shared objects are found in the `handler/src/apps` directory; they are explained in more detail below.

6.1 Host System Requirements

Saphira libraries are available for most UNIX systems (including SunOS 4.1.3, Solaris 2.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 Gnu `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 create *stand-alone* clients that use any of the graphics or user interface routines, you will need the following libraries and headers:

X11R5 or later

Motif 2.0 or later

These libraries are *not* required if you are simply compiling shared objects for loading into the Colbert evaluator, because the library functions are already present in the client.

For MS Windows, the libraries have been compiled with MS Visual C 4.x tools. A `DLL` file and an associated `LIB` file are available. 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; incompatibilities between MSVC and Borland `LIB` files may arise.

6.2 *Compiling and Linking C Source Files*

To compile a loadable shared object file or 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/ver61` in a UNIX system, for example.

After installing the Saphira distribution, follow these steps to create a client or a shared object file:

Write a C or C++ program containing your code, including calls to Saphira library functions.

Compile the program to produce an object file.

Link the object file together with the relevant Saphira library to create an executable or shared object file.

As of Saphira 6.0, all the Saphira library routines are contained in a shared library. In MS Windows, this is `sf.dll`; in UNIX systems, it is the shared library `libsf.so.6.x.y`, where `x` and `y` are the major and minor versions of Saphira. The symbolic link `libsf.so` points to the current shared object library.

In MS Windows, shared libraries (DLLs) cannot be relinked unless no application is using them. If you have loaded a DLL, then make changes to the source code and try to relink it, you will get an error saying that the DLL file is busy. The `unload` command can be used to unload the DLL from Saphira so the link can proceed.

6.2.1 Writing C or C++ Client Programs

To develop a stand-alone Saphira application, or to load C routines into Colbert, you write one or more C or C++ programs that contain your own functions, and make calls to the Saphira library routines. It may help to review Chapter 2 for an explanation of micro-tasks and asynchronous user routines.

For a stand-alone client, the main file will always follow the structure in UNIX systems, as shown in Listing 6-1.

```
#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);
}
```

Listing 6-1. xxxxxx

The Saphira library headers, as well as other relevant system and graphics headers, are loaded by the `handler/saphira.h` file. This file is always included, whether creating a stand-alone client, or loadable shared object files. 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`. An 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 File/Exit item is chosen from the menu.

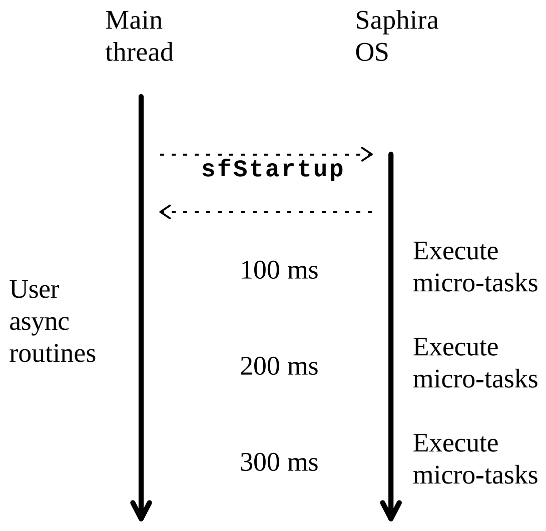
Programming in MSVC is similar, except that the form of the `main` function changes to MS Windows programming standards (see Listing 6-2)

.

```
#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;
}
```

Listing 6-2.

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. 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 6-1 is a graphical view of the execution process. The main client thread starts up, and invokes the Saphira OS with the `sfStartup` function. After start-up, 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 chapter.

6.2.2 Compiling and Linking Client Programs under UNIX

After 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

Figure 6-1. Concurrent execution of Saphira OS and user asynchronous tasks.

compilers provided with UNIX systems should also work, however.

The compiler and linker are typically called using the `make` facility. The file `handler/src/apps/makefile` is used to make all of the sample clients and load files. Listing 6-3 shows a portion of this makefile:

```
#############################################################
# 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)btech $(BIND)saphira $(BIND)async $(BIND)packet $(BIND)nowin
touch all

$(OBJD)saphira.o: $(SRCD)saphira.c
$(CC) $(CFLAGS) -c $(SRCD)saphira.c $(INCLUDE) -o $(OBJD)saphira.o

$(BIND)saphira: $(OBJD)saphira.o
$(CC)  $(OBJD)saphira.o -o $(BIND)saphira \
-L$(SAPHIRA)/handler/obj -lsf -L$(MOTIFD)lib $(LLIBS) -lc -lm

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

testload.so: $(OBJD)testload.o
$(LD) $(SHARED) $(OBJD)testload.o -o testload.so
```

Listing 6-3. A portion of th e `makefile` for Saphira applications.

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 OSes, the Motif (`MOTIFD`), X11 (`X11D`), and system libraries (`LLIBS`) are set correctly, but in some cases this may not be true. 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 `saphira.o` from the `saphira.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.

The file `testload.so` is an example of a shared object file, which is loadable under Colbert. The C source is compiled as usual, but the linking step is different. Instead of creating an executable file, the `LD` command is invoked to create a shared object file (with the extension`.so`). You must include the shared object flags `SHARED`, defined in `os.h` for each particular OS.

6.2.3 Compiling and Linking Client Programs under MSVC

With Microsoft Windows, the sample Saphira clients are MS Visual C++ 4.x projects. All of the sample clients in the `handler/src/apps` directory have two `.mak` files, one for `btech`, and one for all the rest. 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)`. At this time there seems to be no way to specify relative path names, so if you use a different distribution directory (something other than `c:\saphira\ver61`), 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.

The Saphira library file `handler/obj/sf.lib` must be included in the project files.

The project must be compiled in 32-bit mode, not in 16-bit mode.

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.

Executables should be linked with the multithreaded libraries, in the Code Generation item of the C/C++ tab of Build/Settings.

To make a loadable shared file for Colbert, select the *dynamic load library* (`.dll`) project type. The library header file (`.lib` extension), containing linkage information for the load library, is not needed by Colbert.

6.2.4 Debugging C Code under UNIX

The Colbert interaction window is a handy facility for debugging clients, because you can query the values of variables, start and stop activities, and so on. Often, it may be necessary to invoke a more heavy-duty debugging apparatus, especially for complicated C programs. The Gnu debugger `gdb` can be useful, especially when started in Emacs. Here are a few tips for interacting with the Gnu debugger.

To start up, give `gdb` the name of the client executable (usually `saphira`). At the debugger prompt, type `run` to start the client. Before running the program, the Saphira libraries (`libsf.so`) aren't loaded, so you can't set breakpoints in Saphira functions. Similarly, user load files aren't yet present. After the client is running and you have loaded any shared object files into Colbert, you can set breakpoints by interrupting back to the debugger prompt. All the Saphira library exported functions and variables can be examined, and you can set breakpoints in the library functions. The Saphira library has been compiled with the `-g` option, so its symbols are available to the debugger. However, the source code is not in the distribution, so you can't step through library functions.

If you loaded a user shared object file into Colbert, say `testload.so`, you won't see its symbols, even if you used the `-g` option on compilation. That's because user shared objects are read by the dynamic loader, and the debugger has no way of tracking these loads. So it must be explicitly told of user shared object files with the `sharedlibrary` command. For example, giving the debugger command `sharedlibrary testload.so` will make all the symbols in this file available to the debugger, assuming it was compiled with the `-g` option.

6.2.5 Debugging C Code under MS Windows

You can use the MSVC debugger to set breakpoints and step through compiled C code loaded into Colbert as DLLs. All of the exported library symbols can also be examined, although source code is not available.

To invoke the debugger, start from an MSVC project creating the DLL in question (use the Debug build option). Use the `Execute` command; you will be prompted for the name of an executable file, which should be the Saphira client. After the client is started, load the DLL into it via Colbert's `load` command. The MSVC debugger will halt the client on breakpoints, and you can examine the state of the computation.

6.3 Client Examples

In this section, we provide examples of the ways of writing Saphira clients. 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. Most of the examples exist as loadable Colbert files and compilable stand-alone clients.

`saphira.c`

This is the source for the basic client `bin/saphira`. It invokes very basic micro-tasks for communication and display, and starts the Colbert evaluator.

`demo.act/c`

A demonstration client that invokes behaviors, activities, and perception micro-tasks, as well as user-interface functions on the mouse buttons.

`testload.c`

Source for a shared object file to be loaded into Colbert.

`direct.act/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.

`packet.act/c`

This client bypasses the state reflector for Saphira, providing its own packet communication handler.

`async.c`

This client 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.

6.3.1 The Basic `Saphira` Client

The basic client, `bin/saphira`, is used as the typical development environment. It starts up basic micro-tasks for communication and control. It also starts the Colbert evaluator for user interaction, which loads the Colbert file `init.act` from the working directory, if it exists.

Like all Saphira C source files, this 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 from the one 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 start-up or connect event takes place, Saphira calls the registered user function.

The start-up callback can be 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 start-up callback.

In this application, `myStartupFn` is invoked when the Saphira OS is initialized, and it sets the display rate to 5 Hz (see the `sfSetDisplayState` function in the API reference). `myConnectFn` is invoked when the client connects to the robot server (using the Connect menu or `connect` command); here it is empty because no special processing is to be done on connect. You don't need to register this callback if you don't do any special processing on connect; it's here for illustration purposes.

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

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. The mouse callback simply returns 0, invoking the default mouse-click action. Note that the mouse callback could have been omitted; we include it here simply to illustrate how to invoke a mouse callback

```
#include "saphira.h"

void myConnectFn(void);     /* prototypes */
void myStartupFn(void);
int myKeyFn(int ch);
int myButtonFn(int x, int y, int b, int m);

#ifdef IS_UNIX              /* UNIX main function */
void main(int argc, char **argv)
{
  /* set up user button and key processing */
  sfButtonProcFn(myButtonFn);
  sfKeyProcFn(myKeyFn);
  sfOnConnectFn(myConnectFn);
  sfOnStartupFn(myStartupFn);
  /* start up, don't return */
  printf("starting...\n");
  sfStartup(0);
}
#endif

#ifdef MS_WINDOWS
int PASCAL WinMain (HANDLE hInst, HANDLE hPrevInstance,
      LPSTR lpszCmdLine, int nCmdShow)
{
  /* set up user button and key processing */
  sfButtonProcFn(myButtonFn);
  sfKeyProcFn(myKeyFn);
  sfOnConnectFn(myConnectFn);
  sfOnStartupFn(myStartupFn);
  sfStartup(hInst, nCmdShow, 0);
  return 0;
}
#endif


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

int myButtonFn(int x, int y, int b, int m)
{  return 0; }            /* do default handling */

int myKeyFn(int ch)       /* any user processing of keys here */
{
  switch(ch)
  {
    case SPACEKEY:
    sfSetVelocity(0);     /* stop the robot */
    sfMessage("Stopped!");
    return 1;
  }
  return 0;               /* return 0 for default handling */
}
void myConnectFn(void)    /* start those processes */
{}
```

Listing 6-4

6.3.2 The Demo Client

This is the most complex client example; it makes use of activities and predefined micro-tasks and behaviors to implement a handler for the robot. We include 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.

The `demo` client comes in two forms: a loadable Colbert language file (`demo.act`), and a compilable native C code file. We encourage you to use the Colbert language, as it's more understandable and easier to work with and modify.

The Colbert file, shown immediately below, is loaded into the evaluator by using the `load` command in the interaction window. Colbert files can contain functions to evaluate at the top level of the file. On load, the file starts by invoking several sets of predefined micro-tasks for behavior control (`sfInitControlProcs`), registration of the robot to a map (`sfInitRegistrationProcs`), sensor interpretation and object recognition (`sfInitInterpretationProcs`), and an environment-tracking procedure (`sfInitAwareProcs`). These library functions are all accessible in Colbert and are invoked as the file is read.

The second set of statements initializes a variable, and then starts up four behaviors for obstacle avoidance and movement. The movement behaviors are invoked in a suspended state, so that they won't cause the robot to move until they're resumed (from the Activities menu or with the `resume` command).

```
/*
 * demo.act
 * Demonstration of behaviors and activities
 *  using the Colbert evaluator
 */

sfInitControlProcs();     /* for behavior control */
sfInitRegistrationProcs();/* register robot using sensed artifacts */
sfInitInterpretationProcs();     /* find walls and doors */
sfInitAwareProcs();              /* figure out where we are */

/* Start up some behaviors */

sfPreferredTurnDir = sfLEFTTURN;
start sfAvoidCollision(3, 3, sfSHARPLY, 100) priority 0;
start sfKeepOff(100, .25, sfLEFTTURN) priority 1;
start sfConstantVelocity(200) priority 2 suspend;
start sfStop priority 3 suspend;
```

Listing 6-5.

The second part of the file defines two activity schemas, one for following a corridor, the other for reacting when the robot bumps into something and the motors stall.

`FindAndFollow` is a corridor-following activity based on the fuzzy control behavior `sfFollowCorridor`. It starts out by waiting for the current environment of the robot to be a corridor (`sfInitAwareProcs` has the job of updating the environment variables). When this occurs, it fires up the `sfFollowCorridor` behavior with a goal position 10 meters ahead of the robot. Note that the behavior is started in `noblock` mode, which means the execution of `FindAndFollow` continues in parallel with the behavior. The activity is now in monitor mode, checking whether the behavior finishes or the corridor ends. If so, it removes the behavior and goes back to checking for a new corridor. Note the use of `waitfor` at several points to block execution until certain conditions hold.

`FindAndFollow` is started up from within the file using the `start` command. If the activity is interrupted (say by double-clicking in the *Activities* window), then it first removes the corridor-following behavior, then suspends itself.

```
/* Define an activity to follow the current corridor */

act FindAndFollow
{
  point *e;                      /* old environment */
  point *p;                      /* point to go to */
 NOCORRIDOR:                     /* here we have no corridor */
  waitfor (sfCurrentEnvironment != NULL &&
                 sfCurrentEnvironment->type == CORRIDOR);
                                         /* wait until we have a corridor */
  e = sfCurrentEnvironment;
  p = sfCreateLocalPoint(10000, 0, 0); /* point ahead of robot */
  sfAddPoint(p);
  start sfFollowCorridor(e, p) priority 2 iname follow noblock;
                                /* follow corridor to point */
  waitfor (sfCurrentEnvironment != e || sfTaskFinished("follow"));
  remove follow;                 /* remove this behavior */
  goto NOCORRIDOR;               /* resume checking for corridor */
 oninterrupt:
  remove follow;
  suspend;
}

start FindAndFollow suspend;
```

Listing 6-6.

The `BumpAndGo` schema uses direct actions, rather than behaviors, to rescue the robot from stall situations. The `update` facility of activity schemas is used to calculate a `stalled` variable on each cycle. The act allows behaviors to control the robot until it detects a stall condition; then, it turns off behavior execution and starts issuing direct action commands. In the file, the `BumpAndGo` activity is initiated in the active state.

```
/*
 * This activity detects bump collisions on Pioneer
 */

act BumpAndGo
{
  int stalled;                   /* static local variables */
  int recovering;

 update
                                         /* code executed on every cycle */
   stalled = sfStalledMotor(0) + sfStalledMotor(1);

 NOCONTACT:
  untrace;
  sfBehaviorControl = 1;  /* behaviors on */

  waitfor stalled;
  sfBehaviorControl = 0;  /* behaviors on */
  [...]
}

start BumpAndGo;                 /* start it up */
```

Listing 6-7.

The stand-alone client version, `demo.c`, has most of the same functions. However, because Colbert activities are available only in the evaluator, the `FindAndFollow` and `BumpAndGo` activities are not present.

The code starts by defining the `main` function, setting callbacks as in the `saphira` client, and then calling `sfStartup` to initiate the Saphira system. The start-up callback simply sets the display update rate to 5 Hz. On connection to the robot, the registration and interpretation micro-tasks are started up, just as in the Colbert file. In addition, a user micro-task is invoked. This micro-task is defined below.

```
/*
 * The demo client
 */
#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, int m);

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 */
{
sfInitRegistrationProcs();
sfInitInterpretationProcs();
sfInitControlProcs();
sfInitAwareProcs();
sfInitProcess(test_control_proc,"User Process");
}
```

Listing 6-8.

The user micro-task (`test_control_proc`) is very simple; it starts up several behaviors, then puts itself into a suspended state. You can change the state of the invoked behaviors from Saphira’s Function/Activities 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;
sfStartBehavior(sfConstantVelocity, 0, 3, 0,
300.0);
sfStartBehavior(sfStop, 0, 4, 0);
sfStartBehavior(sfAvoidCollision, 0, 0, 0,
3.0, /* front sensitivity */
3.0, /* side sensitivity */
sfSHARPLY, /* turn gain */
100.0); /* standoff */
sfStartBehavior(sfKeepOff, 0, 1, 0,
100.0, /* caution speed */
0.25); /* sensitivity */
process_state = SUSPEND;
break;
case RESUME:
sfMessage("Resumed");
break;
}
}
```

Listing 6-9.

6.3.3 The `testload.so` Loadable Object File Example

Native C code can be loaded into Colbert and executed by compiling the code and linking it to create a shared object file in UNIX, or a dynamic load library in MS Windows. The sample file `testload.c` contains a Saphira library and user function calls in C source. As in the stand-alone client examples, the header file `saphira.h` must be included at the beginning of the source file. The rest of the file contains C function, variable, and structure definitions. The difference between loadable objects and a stand-alone client is that they don't have a no `main` function; instead, the `sfLoadInit` function is called after loading the file, and it typically makes the objects in the file available to the Colbert evaluator, through use of `sfAddEvalXXX` function calls. For information on the effect of these calls, see Section 4.10.1.

Under UNIX, the loadable object source is compiled normally, and the resultant object file is converted to a loadable object file (with the extension `.so`) using the `LD` command and the `SHARED` link flags (see Section 6.2.2). Under MS Windows, the project type is set to Dynamic Load Library rather than Application.

```
/*
test load file for dynamic loading
*/

#include "saphira.h"

int nopen = 0;

int
myfn(int a)
{
return a+1;
}

struct mystruct
{
```

```
int a;
float b;
void *c;
};

struct mystruct m;
int ind_mystruct;

EXPORT void
sfLoadInit(void)                 /* this should be evaluated on open */
{
float a = 1.3;
a = sqrt(a);
printf("Opened! %d %f\n", nopen++, a);
sfSMessage("Opened: %d", nopen);
sfAddEvalFn("myfn", myfn, sfINT, 1, sfINT);
sfAddEvalConst("sfFollowCorridor",   sfBEHAVIOR, sfFollowCorridor);
sfAddEvalConst("sfLEFT",  sfINT, 0);
sfAddEvalVar("sfCurrentEnvironment", sfPTR,
                    (fvalue *)&sfCurrentEnvironment);

ind_mystruct = sfAddEvalStruct("mystruct", sizeof(struct mystruct),
                                  (char *)&m, 3,
"a", &m.a, sfINT,
"b", &m.b, sfFLOAT,
"c", &m.c, sfPTR);
sfAddEvalVar("m", ind_mystruct, (fvalue *)&m);

}
```

Listing 6-10.

6.3.4 The `Direct` Client

Using direct motion commands, the `direct` client moves the robot back and forth along a two meter line. The `direct` client comes in two forms: a loadable Colbert language file (`direct.act`), and a stand-alone native C code file. We encourage you to use the Colbert language, as it's more understandable and easier to work with and modify.

The two activities, `patrol` and `square`, are straightforward realizations of the robot routines in Colbert. A third activity, `aa`, turns on tracing and sequences the two activities. Statements at the end set global mode on the display, and initiate the `aa` activity. Note that you should be connected to the robot before loading this file, otherwise an error will occur when the direct actions are attempted.

```
/*########################################
direct.act  --- exercising the direct motion API
*########################################
*/
act patrol(int a)                /* go back and forth 'a' times */
{
while (a)
{
a = a-1;
     turnto(180);
move(1000);
turnto(0);
move(1000);
}
}

act square                       /* move in a square */
```

```
{
int a;
a = 4;
while(a)
{
a = a-1;
move(1000);
turn(90);
}
}

act aa                                 /* call them sequentially */
{
trace patrol;
start patrol(4);
trace square;
start square;
}

sfSetDisplayState(sfGLOBAL, 1);  /* put display into global coords */
start aa;                        /* start up the toplevel activity */
```

Listing 6-11.

The stand-alone client is `direct.c`. Instead of loading a shared object file into the basic Saphira client, here we create a stand-alone executable that incorporates the Saphira libraries and user code.

In the `main` function, start-up and connection callbacks are registered, and then the Saphira system is started. The `patrol` activity is implemented as a micro-task, only part of which is shown here. Note the explicit completion testing for the direct actions, in contrast to the Colbert implicit waits. Other limitations of micro-tasks relative to activities also exist, e.g., no parameters and no timeouts. The micro-task is initiated using the `sfInitProcess` function.

```
#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)
{
sfSetMaxVelocity(200);           /* robot moves at this speed */
sfInitProcess(patrol,"patrol");
}

void main(int argc, char **argv)
{
```

```
sfOnConnectFn(myConnectFn);/* register a connect function */
sfOnStartupFn(myStartupFn);/* register a startup function */
sfStartup(0);  /* start up the Saphira window */

}
```

Listing 6-12.

6.3.5 The `Packet` Client

This client handles low-level communication with the robot server. It takes advantage of the low-level Saphira communication routines, which parse packets and put the information into the state reflector structures. The Saphira OS is active, allowing concurrent execution of micro-tasks and activities. But the default packet and motor control handlers (packets and motor micro-tasks) are turned off, so that the user program can take over these functions. The `packet` client comes in two forms: a loadable Colbert language file (`packet.act`), and a compilable native C code file. We encourage you to use the Colbert language, as it's more understandable and easier to work with and modify.

The example starts out by defining an activity schema for packet communications, `DoPackets`. This activity first turns off the default Saphira packet and motor micro-tasks, which are invoked by `sfStartup`. It then waits until the client connects to the robot, and tells the robot server to open its motor control and start traveling forward at 300 mm/sec. If the connection does not succeed in 10 seconds, the activity exits with failure. Note the use of a timeout in the `waitfor` statement to accomplish this.

After this initialization, the activity reads packets in a `while` loop, calling the default packet processor `sfProcessClientPacket` for each packet. Default processing updates the client state reflector in `sfRobot`, so that position integration values are available to the client. Every 10 cycles, new commands are sent to the robot server to keep the information packets coming, and to keep going at the requested velocity. Also, the robot position is printed in the information area. Note that, because Colbert has no explicit type casts, and the `sfSMessage` function does not handle floats correctly, the robot coordinates are first implicitly cast to integers via an assignment, and then printed out.

At the end of the activity, the robot velocity is set to 0, and the client disconnects from the robot. At the top level of the file, the `connect` function acts to connect the client to the robot simulator, and then the `DoPackets` activity is invoked.

```
/*#########################################
packet.act  --- routines for connecting and
reading packets
*#########################################
*/

act DoPackets()
{
int i;  int x;  int y;

remove packets;
remove motor;
waitfor(sfIsConnected) timeout 100;
  if (!sfIsConnected) fail;
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 */

i = 0;
while (i<100)
{
if (sfWaitClientPacket(1000)) /* wait 1 second for a packet */
{
```

```
i = i+1;
sfProcessClientPacket(sfReadClientByte());
}
if ((i % 10) == 0)
{
sfRobotCom(sfCOMPULSE); /* keep asking */
sfRobotComInt(sfCOMVEL, 300);    /* keep it going... */
sfSMessage("%d packets received", i);
x = sfRobot.ax;
y = sfRobot.ay;
sfSMessage("X: %d  Y: %d", x, y);
}
}

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

connect local;                   /* connect to simulator */

/* for the Pioneer on a tty line,
use 'connect serial <port>' */

start DoPackets();
```

Listing 6-13.

The stand-alone client is similar, but uses a micro-task instead of the activity. As in every stand-alone client, the start-up function is registered, and then the `sfStartup` function is invoked to initiate the Saphira OS.

In the start-up function, the display state is changed to show global movement of the robot, and the task `myTask` is instantiated. Then, the two default Saphira micro-tasks that handle packets and motor control are removed, so that the user task can perform these functions. Finally, the `sfConnectToRobot` function is called to connect the client to the robot server.

The `myTask` micro-task waits until the robot is connected, then opens the motor controller and tells it to move forward at 300 millimeters per second. Execution now proceeds as in the `packet.act` activity; the only difference is that the micro-task must explicitly sequence its operations by changing state. After the packets are received, the task stops the robot and disconnects from the server.

```
#include "saphira.h"
void myStartupFn(void);          /* forward refs */
void myTask(void);

void main(int argc, char **argv)
{
int i = 0;
sfOnStartupFn(myStartupFn);      /* register a startup function */
sfStartup(0);             /* start up the Saphira window, wait */
}

void myStartupFn(void)
{
sfSetDisplayState(sfGLOBAL, TRUE);     /* use the global view */
sfInitProcess(myTask, "myPackets");
sfRemoveTask("packets");  /* get rid of default packet process */
sfRemoveTask("motor");    /* get rid of default motor control */

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

sfConnectToRobot(sfLOCALPORT, sfCOMLOCAL);   /* this is for the simulator */
/* sfConnectToRobot(sfTTYPORT, sfCOM1);  this is for Pioneer */
}

void myTask(void)
{
static int i = 1;
switch (process_state)
{
case sfINIT:
if (sfIsConnected) process_state = 10;
break;

case 10:                          /* connected */
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 */
process_state = 20;
break;

case 20:
/* read 100 packets */
if (i > 100) process_state = 30;
while (sfWaitClientPacket(0)) /* poll for packets */
{
i++;
sfProcessClientPacket(sfReadClientByte());
}
if (i % 10 == 0)
{
sfRobotCom(sfCOMPULSE); /* keep asking */
sfRobotComInt(sfCOMVEL, 300);     /* keep it going... */
sfSMessage("%d packets received", i);
sfSMessage("X: %f  Y: %f", sfRobot.ax, sfRobot.ay);
}
break;

case 30:
sfRobotComInt(sfCOMVEL, 0);       /* stop the robot */
sfDisconnectFromRobot();
process_state = sfSUCCESS;
break;
}
}
```

Listing 6-14.

6.3.6 The `Async` Client

This client demonstrates asynchronous control of the robot; that is, control outside the micro-task loop. As in the `direct` client, the start-up 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 but continues executing the user's program in the `main` function. It's important that the Saphira OS be operating, because its default micro-tasks handle communication and motor control to the robot server, which keeps the state reflector current. The direct action calls of the user program depend on these micro-tasks.

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

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 deg/sec */
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);         /* stop */
sfPause(4000);
sfDisconnectFromRobot();  /* we're gone... */
}
```

Listing 6-15.

6.3.7 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 specify a console application instead of window application, and use the `main` function instead of `WinMain`. In `sfStartup`, you must still pass three arguments, but the first two, which are window parameters, should be `NULL`.

Note that, even though windows are not being displayed, the Saphira OS is operating, and the basic set of micro-tasks are managing communication and control.

```
#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 */
sfConnectToRobot(sfLOCALPORT, sfCOMLOCAL);
                               /* this is for the simulator */
while (!sfIsConnected) sfPause(100);
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... */
}
```

Listing 6-16.

7 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 7-1.

High-level robotics applications developers do not need to know many details about a particular robot

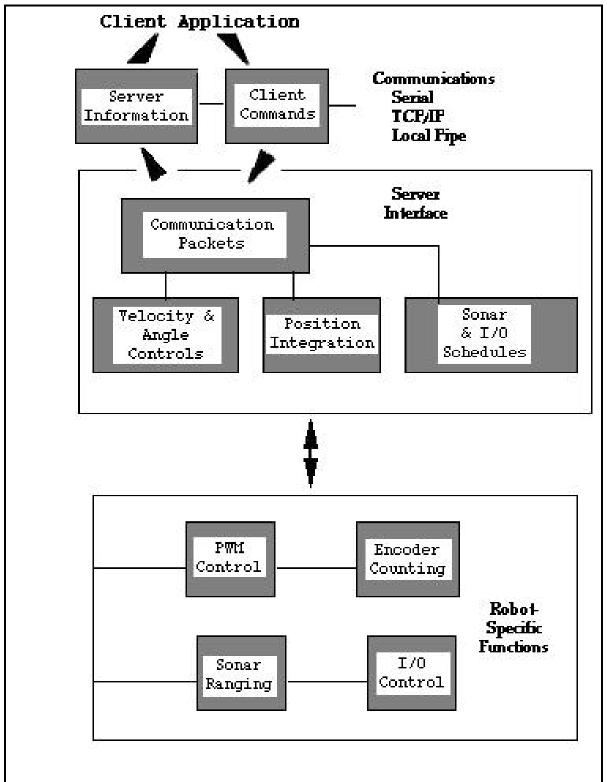


Figure 7-1. Saphira client-robot server architecture.

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.

7.1 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 7.1): 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 a two-byte checksum.

Table 7.1 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 less than 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

7.1.1 Packet Data Types

Packetized client commands and server information blocks use several data types, as defined in Table 7.2. There is no convention for sign; each packet type is interpreted idiosyncratically by the receiver. Negative integers are sign-extended.

Table 7.2 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

7.1.2 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 an odd number of data bytes exists, the last byte is XORed 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 C-code fragment in Listing 6-17 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);
}
```

Listing 6-17. C-code fragment to computer checksum.

7.1.3 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 does it have 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 such housekeeping functions as 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.

7.2 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 (see Table 7.3 and earlier sections of this chapter).

Table 7.4 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 7.3 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 less than 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

7.2.1 Client Command Argument Types

Three different types of client command arguments exist: 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.

7.2.2 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 7.4. PSOS 4.2 supported client commands.

Command Name	Number	Argument Value(s)	Description
`sfSYNC0`	0	none	Start connection; server echoes these
`sfSYNC1`	1	none	Synchronization commands back to
`sfSYNC2`	2	none	client.
`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	bytes 0-4, 0-200	Set pulse-width for RC servo control. First argument is RC servo number, second is width of pulse in 10 us increments (i.e., 0 to 2000 us).
`sfCOMSTEP`	64	none	Single-step mode (simulator only)

7.3 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 7-5. 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 7-5. 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 degrees.
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 degrees
PTU	unsigned int	Pulse width of last RC servo command received
Compass	byte	Compass reading, 0-179 (x2 for actual reading)
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	into	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.

7.4 Start-Up and Shutdown

Before exerting any control, a client application must first establish a connection to the robot server via an RS-232 serial link (9600 baud), an interprocess connection (UNIX pipe, for example, or MS Windows mailslot), or TCP/IP network. Over that established communication link, the client then sends commands to and receives back operating information from the server.

Connection is usually done through the library function `sfConnectToRobot`, which takes two arguments. The first is the connection type, the second is the port name. Table 7-6 lists the types and some special port names available in the Saphira library.

Table 7-6. Port types and names for client/server connections

Port types	**Descripton**
`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	
`sfComPipe`	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)
`sfComServer`	Host name/IP address of simulator server running on another machine

`sfConnectToRobot` performs three tasks:

Synchronizes the communication channel by sending and receiving three `SYNC` packets.

Reads an autoconfiguration packet sent by the server to identify the characteristics of the robot.

Sends a motor open command to the server.

Instead of using `sfConnectToRobot`, the user can perform these tasks with low-level library calls, detailed in the next few sections.

7.4.1 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, in succession, a series of three synchronization packets through the host communication port—`sfSYNC0`, `sfSYNC1`, and `sfSYNC2`. The server responds to each, forming a succession of identical synchronization packets. The client should listen for the returned packets and issue the next synchronization packet only after it has received the echo.

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

7.4.2 Autoconfiguration

The Saphira-aware servers (PSOS v4.1 or later) send configuration information back to the client in the last sync packet (`sfSYNC2`). Following the sync byte are three null-terminated strings that represent the robot name, robot class, and robot subclass (see Table 7-7). You can read these strings with the library function `sfReadClientString`. The function `sfConnectToRobot` reads the strings and sets the appropriate Saphira variables to their values.

Table 7-7. Robot configuration information.

Name	**Description**
`sfRobotName`	Given name for Pioneer-class robots, computer name for Bxx--class robots, simulator" for the simulator
`sfRobotClass`	Pioneer, B14, or B21
`sfRobotSubclass`	`pion1` (Pioneer 1) or `pionat` (Pioneer AT) 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.

7.4.3 Opening the Servers—`sfCOMOPEN`

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

7.4.4Keeping 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.

7.4.5 Closing the Connection—`sfCOMCLOSE`

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

7.4.6 Movement Commands

Table 7-8. Server motion command types

Rotation	Translation
`sfCOMHEAD` absolute heading	
`sfCOMDHEAD` differential heading from control pt	
`sfCOMDCHEAD` differential heading from current	`sfCOMVEL` forward/back velocity
`sfCOMRVEL` rotational velocity	

`sfCOMVEL2`	left and right wheel velocities

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 7-8 summarizes the command modes available.

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.

Table 7.9. Motion command arguments

Command	Argument(s)	Typical Invocation
`sfCOMHEAD`	degrees (int) [0, 360]	`sfRobotComInt(sfCOMHEAD, 320)`
`sfCOMD[C]HEAD`	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 arguments for these commands are given in Table 7.9, below. The heading commands are with respect to the robot's internal coordinate system (see the section below).

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.

7.5 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 (see Figure 7-2). This function is recomputed each time a new heading command is received, making on-the-fly orientation changes possible.

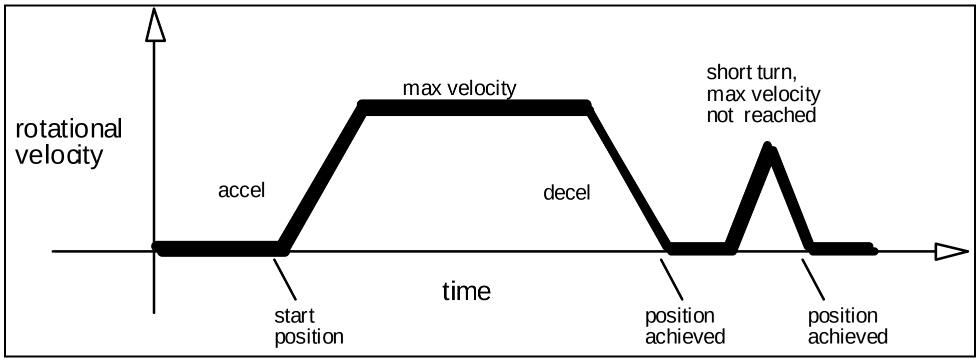


Figure 7-2. Trapezoidal turning velocity profile.

7.5.1 Position Integration

Your robot 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 meters and one revolution, depending on the surface (carpets tend to be worse than hard floors).

Also, moving too fast or too slow tends to exacerbate 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`, `sfCOMDHEAD`, and `sfCOMDCHEAD` turn the robot with respect to its internal dead-reckoned angle (see Figure 7-3). On start-up, 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 robot 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).

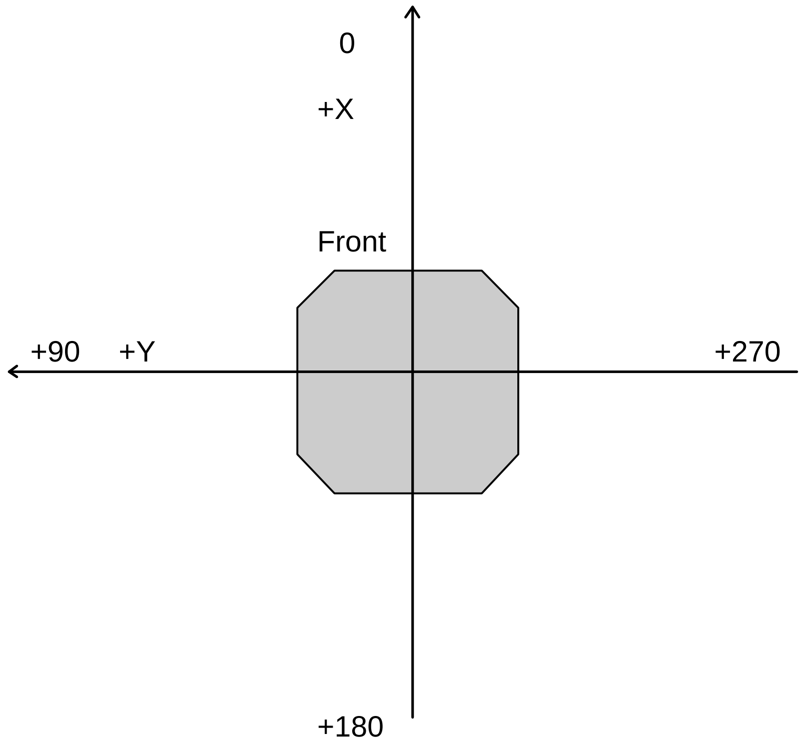


Figure 7-3 Internal coordinate system of a Saphira-aware server.

7.6 Sonars

When opened by the appropriate client command (see `sfCOMOPEN,` above), a Saphira-aware robot server automatically coordinates and begins firing its robot sonars in a predefined default sequence; it 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 that can be, at most, 12 bytes long. Each byte is 1 + sonar number. For example, the following string starts the sonar polling sequence 0, 1, 0, 5:

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

Note that sonar numbers can be repeated. If the string is empty, all sonars are turned off.

8 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 sample source files in `handler/src/apps` as examples of working Saphira applications.

Most of these functions and variables are available in the Colbert evaluator. Those that are not are indicated in the text.

8.1 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 format 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 multitasking 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 it 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. These functions are not available in Colbert.

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

These functions register callbacks for Saphira events: when the Saphira OS first starts up, when it connects to a robot, and when it disconnects. The functions are only used in stand-alone client code that calls `sfStartup`. The variable `sfIsConnected` is also useful in Colbert activities to check if the robot server is currently connected to the client. The user should not change the value of this variable.

The variable `sfIsExited` is set to 1 when the user requests Saphira to exit from the Connect/Exit menu item. This variable is useful for async user code, which calls `sfStartup` in non-blocking mode and then continues execution. The code can check the `sfIsExited` flag to see if there is an exit request.

None of these callbacks is obligatory; in user code, usually the connect callback is registered. The start-up callback should include any relevant initialization code, such as menu or directory settings, in this function. The connect callback should start micro-tasks, behaviors, and other Saphira control routines. The disconnect callback can be used to clean up after the Saphira client disconnects from a robot.

Use the `sfSetDisplayState` function to change the state of a display mode in the Saphira window interface:

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

If you call this function before connecting to the robot (in the start-up 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 8-1), whenever a new connection is made with the robot.

Table 8-1. 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.

`sfMessage` writes the null-terminated string `str` into the message section of the information area in the Saphira main window, followed by a carriage-return:

```
void sfMessage (char *str)
```

Use `sfSMessage` to format the string much as you would C's standard `printf` function, which accepts optional arguments that are to be inserted into the string. :

```
void sfsMessage (char *str, …)
```

A problem in the Colbert evaluator prevents floating-point numbers from being printed using `sfSMessage`. As a workaround, convert them to integers before calling `sfSMessage`. (The `sfKeyProcFn` registers an optional user key process callback, with the prototype of `myKeyFn:`

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

It is called by Saphira whenever the user presses a key when the main Saphira window is active. The argument `ch` 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 override any of Saphira's built-in key processing routines (see Table 8-1).

Not available in Colbert. The `sfButtonProcFn` registers an optional user button process callback, with the prototype of `myButtonFn:`

```
 void  sfButtonProcFn (int (*fn)())
int   myButtonFn (int x, int y, int b, int m)
int   sfLeftButton, sfMiddleButton, sfRightButton
int   sfShiftMask, sfControlMask, sfAltMask
float sfScreenToWorldX (int x, int y)
float sfScreenToWorldY(int x, int y)
```

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`. The shift mask argument `m` is an integer that has bits set indicating which modifier keys were pressed. Return 0 if you don't handle the mouse click; return 1 if you do, to override 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.

Not available in Colbert.

8.2 Predefined Saphira Micro-Tasks

We've provided a variety of predefined Saphira micro-tasks for control of the robot. You may initiate these micro-task sets using the API functions described here, or invoke them individually using the `sfInitProcess` API call (see Section 8.5)

Both the micro-task function and the instantiation name given by the `init` function are described here. The instantiation name is used to refer to the running micro-task, and is shown in the Function/Processes window. To remove a micro-task with instantiation name `iname`, you can type `remove iname` in the interaction window or an activity, or use `sfRemoveTask("iname")` from C code.

`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 shown in Table 8-2.

Table 8-2. Basic communication, display, motor, and sensor control processes

Function	Name	Description
`pulse_proc`	`pulse`	Sends communication pulse every 1 second
`motor_proc`	`motor`	Coordinates keyboard and behavior motor commands
`clamp_proc`	`clamp`	Rotates the world around the robot
`sonar_proc`	`sonar`	Adds new sonar readings to the sonar buffer
`wake_proc`	`wake`	Draws a wake of the robot's motion
`draw_proc`	`draw`	Updates Saphira display window
`process_waiting_packets`	`packets`	Parses information packets from robot server

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

`sfInitBasicProcs` is invoked by `sfStartup`, so the user should not have to call this function. Not available in Colbert.

`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.

Table 8-3.

Function	Name	Description
`execute_current_behaviors`	`execute`	Evaluates behaviors and outputs a motor control

`void sfInitInterpretationProcs (void)`

Starts up processes for interpretation of sonar results.

Table 8-4.

Function	Name	Description
`occgrid_proc`	`occupancy grid`	Computes an occupancy grid
`side_segment_proc`	`side segs`	Forms linear artifacts robot motion
`test_wall_proc`	`test wall`	Performs wall recognition
`test_wall_break_proc`	`test wall break`	door and junction recognition

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

`void sfInitRegistrationProcs (void)`

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

Table 8-5.

Function	Name	Description
`test_match_proc`	`test matching`	matching of linear and point artifacts
`test_environment_proc`	`test where`	identification of current situation

`void sfRunEvaluator (void)`

This micro-task starts up the Colbert evaluator, which is the executive for activities. The evaluator also accepts input from the interaction window. The basic client `bin/saphira.c` starts this process. If you define a stand-alone client, and want to run Colbert, then start this micro-task (using `sfInitProcess`) in your start-up callback.

8.3 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.

The state reflector has three important data structures.

The `sfRobot` structure holds motion and position integration information, as well as some sensor readings (motor stall sensors, digital I/O ports).

The sonar buffers hold information about current and past sonar returns.

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 variable `sfRobot` holds basic information reflected from the robot server. Table 8-6, 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.

This variable is defined in Colbert, as well as the robot structure, and most of the fields are available; type

Table 8-6. Definition of the `sfRobot` structure.

`help robot` for a list of fields.

`sfRobot` field	Units	Description
`x, y, th`	mm, mm, degrees	Robot's location in robot coordinates; always (0, 0, 0)
`ax, ay, ath`	mm, mm, degrees	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

8.3.1 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 is stalled and 0 if it isn't. The argument `which` is `sfLEFT` or `sfRIGHT`.

8.3.2 Sonar buckets

The current range reading of 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. Sonars start at number 0. This variable is not defined in Colbert, which doesn't have arrays; instead use the convenience function `sfSonarBucket`.

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. Several 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 the section

```
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 */
```

on the LPS, below.

Listing 8-1.

```
sdata *sfSonarBucket(int num)
 int   sfSonarRange(int num)
 float sfSonarXCoord(int num)
 float sfSonarYCoord(int num)
 int   sfSonarNew(int num)
```

The first function returns a pointer to the data structure of the `num`'th sonar, or `NULL` if no such sonar exists.

The next 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 isn't returned twice.

8.4 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.

It checks that the setpoints are actually sent to the robot server, given the unreliability of the communication channel.

It implements a set of checking functions for determining when the motion commands are finished.

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. This is done automatically by `sfStartup`, so the user need not call it explicitly.

`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. The argument is in mm, either positive (forward) or negative (backward). 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 in degrees for heading. On a Pioneer robot, you should use at least 100 mm for the distance completion, and 10 degrees for angle. Otherwise, the robot may not move enough to trigger the completion function. 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.

8.5 Saphira Multitasking

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 a limited amount of work.

Compute-intensive processes that take a long time to complete, but that 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 that they execute as a concurrent thread, as we describe below.

Colbert activities and behaviors are also micro-tasks and are defined using the Colbert language or behavior compiler (see Chapters 1 and 4). Some of the micro-task control functions described below are useful for these tasks, as well. To distinguish behaviors and activities from other micro-tasks, we call the latter *simple micro-tasks*.

8.5.1 Micro-task Definition

Simple micro-tasks are functions with no arguments together with state information. Micro-tasks 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. After they are initialized, Saphira will call them with an initial state of `sfINIT`. The micro-task 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 (see Table 8-7).

Table 8.7. Saphira multiprocessing reserved process state values.

State	Explanation
`sfINIT`	Initial state
`sfSUSPEND`	Suspended state
`sfRESUME`	Resumed state
`sfINTERRUPT`	Interrupted state
`sfREMOVE`	Requests the scheduler to remove this micro-task
`sfSUCCESS`	Micro-task succeeded (default ending)
`sfFAILURE`	Micro-task failed
`sfTIMEOUT`	Micro-task timed out
`-n`	Suspend this micro-task for n cycles

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

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

The `sfSUCCESS` and `sfFAILURE` states are used to indicate the successful or unsuccessful completion of a micro-task. The micro-task may set these as appropriate, or signal other micro-tasks to set them. No further processing takes place unless the micro-task is resumed.

Simple micro-tasks do not have timeouts, but activities and behaviors do. In these cases, a state of `sfTIMEOUT` means that the micro-task has timed out before completing its job.

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

Listing 8-2 provides an example of a typical interpretation micro-task function. It starts by setting up 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;
        }
}
```

Listing 8-2. Example of a typical interpretation micro-task function.

8.5.2 State Inquiries

The state of a micro-task can be queried with the following functions.

---

int sfGetProcessState(sfprocess *p)
int sfGetTaskState(char *iname)
int sfSuspended(sfprocess *p)

```
 int sfTaskSuspended(char *iname)
 int sfFinished(sfprocess *p)
 int sfTaskFinished(char *iname)
```

These functions come in two varieties: those that take a micro-task pointer as an argument, and those that take an instantiation name. The latter first look up the micro-task in the task list, using the instantiation name.

`sfGetProcessState` returns the state of the process as an integer, if it exists; otherwise, it returns 0.

`sfSuspended` is 1 if the micro-task is suspended and 0 if it is active.

`sfFinished` is 1 if the task has completed successfully, failed, or timed out; it is 2 if the micro-task is not on the scheduler's list; and it is 0 if the micro-task is still active.

8.5.3 Micro-Task Manipulation

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

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

The `sfInitProcess` function starts up a micro-task with the name `name` and function `fn`, and returns the micro-task instance pointer, which can be used in micro-task-manipulation functions. No corresponding function for deleting micro-tasks exists—suspend it if it is no longer needed.

```
sfprocess *sfFindProcess (char *name)
```

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

```
void sfSetProcessState (sfprocess *p, int state)
 void sfSuspendProcess (sfprocess *p, int n)
 void sfSuspendTask (char *iname, int n)
 void sfSuspendSelf (int n)
 void sfInterruptProcess (sfprocess *p)
 void sfInterruptTask (char *iname)
 void sfInterruptSelf (void)
 void sfResumeProcess (sfprocess *p)
 void sfResumeTask (char *iname)
 void sfRemoveProcess (sfprocess *p)
 void sfRemoveSelf (void)
 void sfRemoveTask (char *iname)
```

The `sfSetProcessState` function sets the state of micro-task instance `p` to `state`. The argument `p` must be a valid micro-task instance pointer, returned from `sfFindProcess` or `sfInitProcess`. The other functions are particular calls to `sfSetProcessState`. The other functions are convenience functions for signaling micro-tasks to set certain states.

8.5.4 Invoking Behaviors

Behavior activities can be invoked from Colbert with the `start` command, or from C code with the following function.

```
sfprocess sfStartBehavior(behavior *b, char *in, int tout,
                 int pri, int suspend, ...)
```

The `sfStartBehavior` function instantiates a behavior activity, using behavior schema `b`. The instantiation name is `in`, and the priority of the behavior is `pri`. A timeout (`tout`) must be specified; a timeout of 0 means the behavior will execute indefinitely. The `suspend` argument is 0 if the behavior is to be active immediately, and 1 if it is to be started in a suspended state, to be activated by a `resume` signal.

The remainder of the arguments to `sfStartBehavior` are the arguments to the behavior. There must be exactly the same number and types of arguments as are specified by the behavior parameters.

This function is equivalent to the following:

```
start b(...) iname in timeout tout priority pri [suspend]
```

where `b` is the name of the behavior schema.

8.5.5 Activity Schema Instantiation

An activity schema can be instantiated from another Colbert activity or the user interaction area, with the `start` command (see Section 4.8.3). Alternatively, activities can be started from C code with the `sfStartActivity` function.

```
int sfStartActivity(char *schema, char *in, int tout,
                     int suspend, ...)
```

The `sfStartActivity` function instantiates an activity whose library name is `schema`. The instantiation name is `in`. A timeout (`tout`) must be specified; a timeout of 0 means the activity executes indefinitely. The `suspend` argument is 0 if the behavior is to be active immediately, and 1 if it is to be started in a suspended state, to be activated by a `resume` signal.

The remainder of the arguments to `sfStartActivity` are the arguments to the activity. The number and types of arguments must equal the number specified by the behavior parameters.

This function is equivalent to this one:

```
start schema(...) iname in timeout tout [suspend]
```

where `schema` is the name of the activity schema.

The function returns 0 if it instantiated the activity successfully, and -1 if it did not.

8.6 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 in Chapter 4 (a system that represents the dead-reckoned position of the robot server), the LPS is an egocentric coordinate space that remains clamped to the robot center (see Figure 8-1).

Units in the LPS are millimeters and degrees. For example, the position of a point artifact in the LPS is represented by an *x* and *y* coordinate in mm, and as an angle relative to the *x* axis, in degrees. *Note: Starting with version 6.1, all internal and user angles are specified in degrees, rather than radians.*

8.6.1 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.

The 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 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 it's almost impossible to distinguish the shape of the obstacle. A wall in front of the robot, for example, will look only a little bit like a straight line (see the excellent book by Leonard and Durant-Whyte).

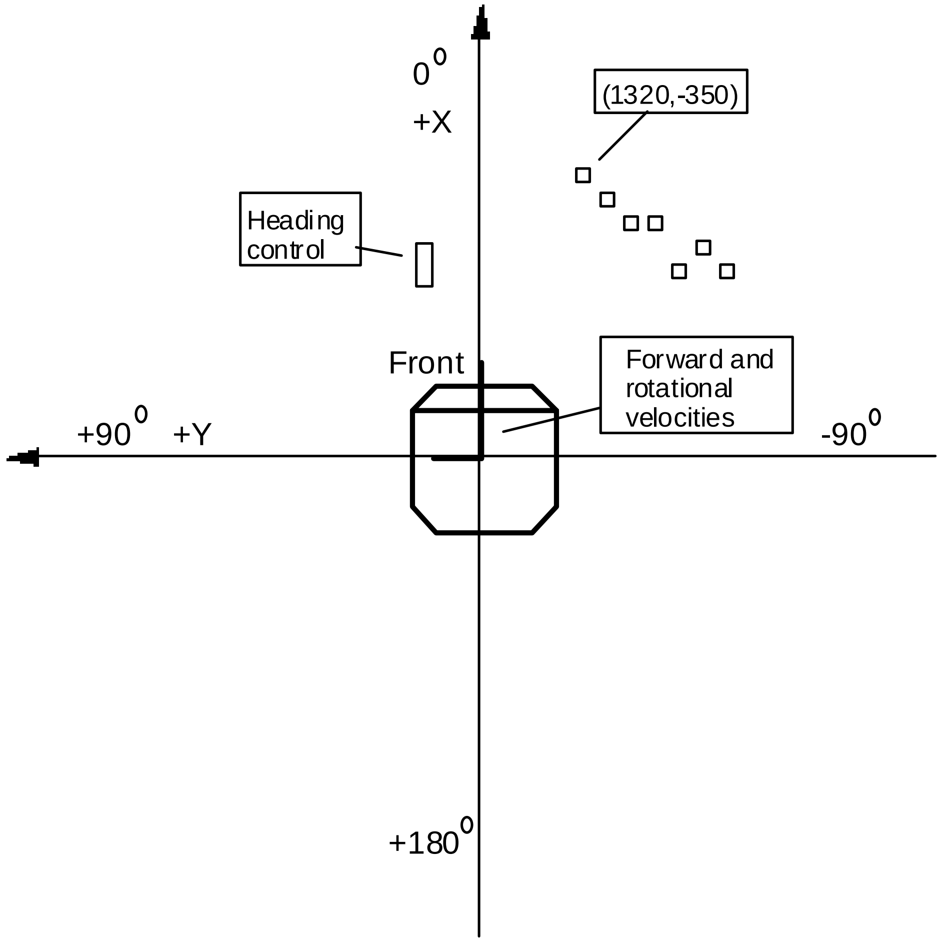


Figure 8-1. Saphira's LPS coordinate system.

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 often is moving parallel to wall surfaces. As side sonar readings are accumulated, it's possible to pick out a nice straight feature.

The buffers differ slightly in how they accumulate sonar readings and therefore serve different purposes. 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 nothing has been in the robot's way for a short time. 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 nothing is 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

```
#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;
```

sonar readings stay *registered* with respect to the robot's movements.

`Listing 8-3.`

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

These buffers are not currently available in Colbert. 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 0, they clear their buffers, that is, set the `valid` flags to 0. These buffer limits can also be set from the parameter file; they 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.

8.6.2 Occupancy functions

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

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 a 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,
                 float *x, float *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 no sonar reading is made within the rectangle, it returns 5,000 (5 meters).

For example, in the case of an LPS shown in Figure 8-2, `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.

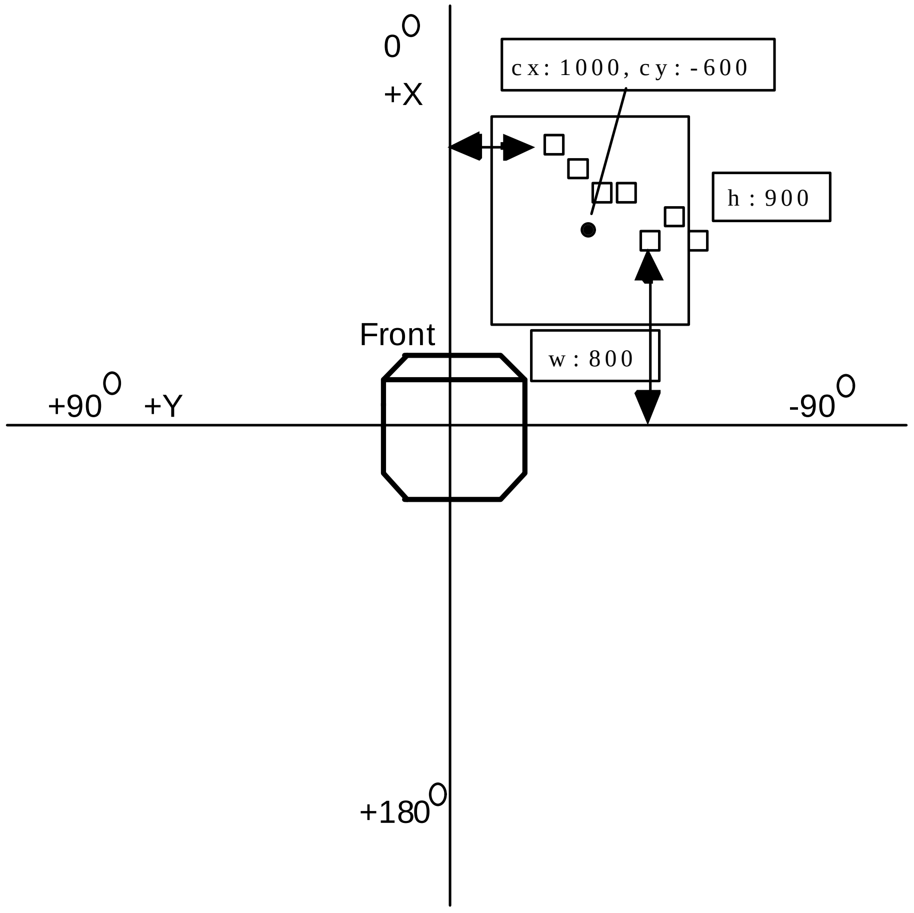


Figure 8-2. Sensitivity rectangle for the `sfOccBox` functions.

```
int sfOccPlane (int xy, int source, int d, int s1, int s2)
 int sfOccPlaneRet (int xy, int source, int d, int s1, int s2,
                    float *x, float *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 8-3 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.

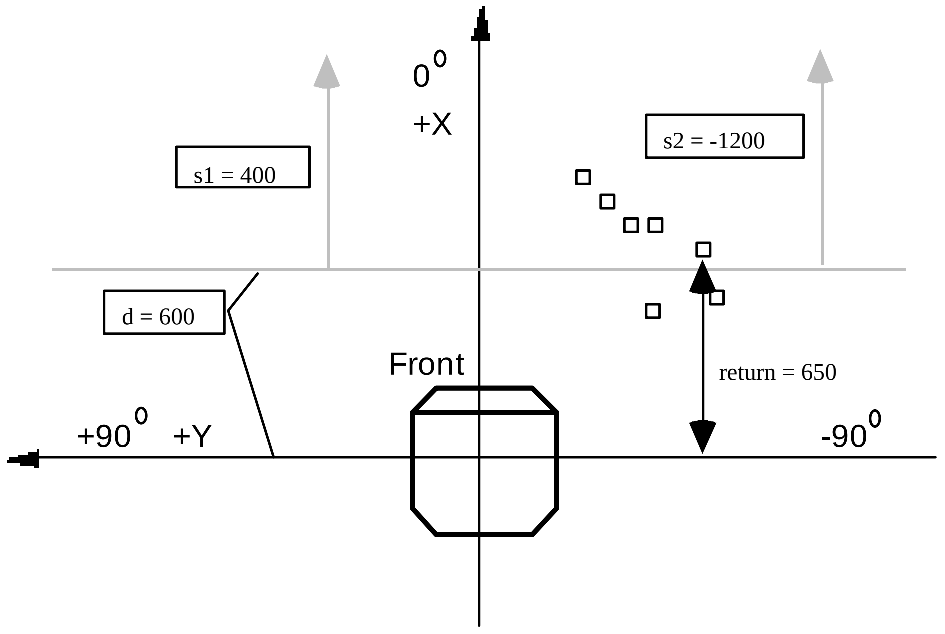


Figure 8-3 Sensitivity rectangle for `sfOccPlane` functions.

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

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

8.7 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 don'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. User may add two types of artifacts. *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, a behavior to reach a goal position exists, and the goal is represented as a point artifact in the LPS. Usually, these artifacts are deleted when the behavior is completed.

The system also maintains artifacts of different types: An artifact represents the origin of the global coordinate system, for instance, and various *hypothesis artifacts* represent hypothesized objects extracted by the perceptual routines and used by the registration routines.

8.7.1 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;
```

Listing 8-4.

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. All *x*,*y* coordinates are in millimeters, and direction is in degrees from -180 to 180. The type `POS` is used for goal positions in behaviors. Other types may add additional fields to the basic `point` type– for example, 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;                  /* unique numeric id */
  float ax, ay, ath;       /* global coords */
  unsigned int matched;    /* last time we matched */
  unsigned int announced; /* last time we announced */
} point;
```

Listing 8-5.

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 use the supplied coordinates to create new `ARTIFACT` points of type `POS`, which is very useful for behavir goal positions. For example, `sfCreateLocalPoint(1000.0, 0.0, 0.0)`creates a point 1 meter in front of the robot.

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.

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.

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

These functions add and delete members of the pointlist. Ordinarily, to add a point to the pointlist, you 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. `sfFindArtifact` returns the artifact on the pointlist with identifier `id`, if it exists; otherwise, it returns `NULL`. Finally, `sfRemArtifact` removes an artifact from the list, given its `id`.

The pointlist is available as the value of the variable `sfPointList`. The definition of a list is given in `handler/include/struct.h`. If it is necessary to check current artifacts, a function can iterate through this list.

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

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

8.7.2 Other Artifact Creation Functions

Walls, corridors, doors, junctions, and lanes can all be created with the following help functions. These artifacts are important in defining maps for the robot.

```
point *sfCreateLocalArtifact(int type, int id, float x, float y,
      float th, float width, float length)
 point *sfCreateGlobalArtifact(int type, int id, float x, float y,
      float th, float width, float length)
```

Type	Return Value

Table 8-7. Artifact creation types.

sfCORRIDOR	corridor *
sfLANE	lane *
sfDOOR	door *
sfJUNCTION	junction *
sfWALL	wall *
sfPOINT	point *

These two functions create and return artifacts of the specified type, using either local or global coordinates. Table 8.7 shows the allowed types:

Although these functions are declared as returning type `point *`, in fact they return a pointer to the appropriate structure, and the result should be cast as such. All these structures are similar in their first several arguments (i.e., local and global coordinates), so all can be used in the geometry manipulation functions.

Unlike the `sfCreateXPoint` functions, these functions automatically add the artifact to the pointlist. So, if you want to create a point and add it to the pointlist, use the `sfPOINT` type here, instead of the `sfCreateXPoint` functions.

Not all types use all of the parameters: length and width are ignored for `sfPOINT`, length is ignored for `sfDOOR` and `sfJUNCTION`., and width is ignored for `sfWALL`. In general, the `x, y ,th` coordinates are for a point in the middle of the artifact. Figure 8-4 hows the geometry of the constructed artifacts.

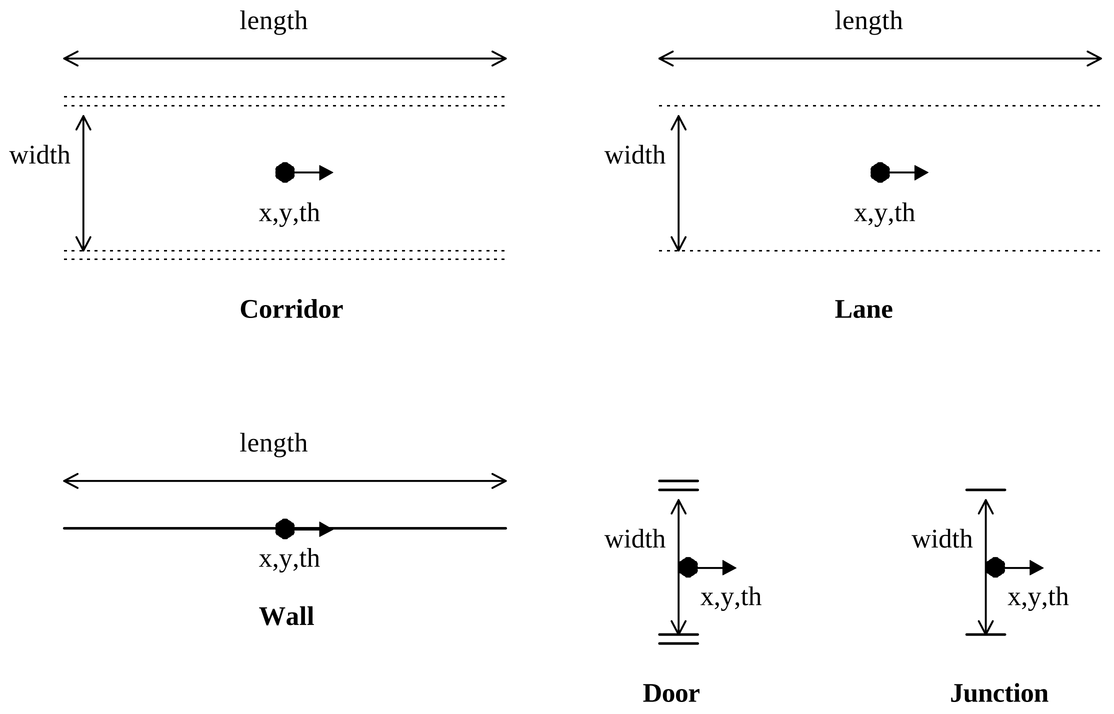


Figure 8-4 Geometry of artifact types. The defining point for the artifact is shown as a vector with a circle at the origin.

Artifacts are most often used in constructing maps for the robot and registering it based on sensor readings (see Section 8.10).

8.7.3 Geometry Functions

Saphira provides a set of functions to manipulate the geometric parameters of artifacts. These functions typically work on the local coordinates of the artifact. To update an artifact properly after changing its local coordinates, you should call the `sfSetGlobalCoords` function.

```
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 degrees, using floating-point numbers. Artifact angles are always normalized to the interval $[0,360]$. `sfNormAngle` will put its argument into this range. The corresponding functions `sfAddAngle` and `sfSubAngle` also normalize their results in 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 $[-180,180]$, 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 $[-90,90]$. 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 function `sfPointPhi` returns the angle of the vector between the robot and point `p`, in degrees from -180 to 180. `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; it will be positive if the normal segment is to the left of the robot's *x* axis, and negative if to the right.

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`. The distance will be positive if the normal segment is to the left of `q`'s *x* axis, and negative if to the right. `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. Because each point artifact represents a coordinate system, often it is 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 along the direction it is currently pointing.

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.

8.8 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. For wall hypotheses to be found, the wall-finding routines must be invoked with `sfInitInterpretationProcs`.

8.9 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 global LPS coordinate system.

```
void sfDrawVector (float x1, float y1, float x2, float y2)
 void sfDrawRect (float x, float y, float dx, float dy)
 void sfDrawCenteredRect (float x, float y, float w, float h)
```

`sfDrawVector` draws a line from `x1, y1` to `x2, y2`. This line is in global coordinates.

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 non-centered version takes the lower-left corner position, a width, and a height.

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 because you cannot depend on the state of the graphics context when you make a graphics call, you should set it appropriately.

```
void sfSetLineWidth (int w)
 void sfSetLineType (int w)
 void sfSetLineColor (int color)
 void sfSetPatchColor (int color)
 int sfRobotColor
 int sfSonarColor
 int sfWakeColor
 int sfArtifactColor
 int sfStatusColor
 int sfSegmentColor
```

For lines, set the width `w` to the desired pixel width. This width affects all lines drawn in rectangles and vectors. You may select one of two line types: Set the `w` function parameter to sfLINESOLID for a solid line, and sfLINEDASHED for a dashed line. The patch and line colors accept a color value as shown in Table 8.8.

Table 8.8. 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

Saphira drawing colors for the robot icon and various artifacts can be set using the variables shown above.

8.10 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 using 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. You can also save the map created by the robot to a file, for later recall.

8.10.1 Map File Format

A map file contains optional comments, designated with a semicolon (;) prefix, and lines specifying artifacts in the map. All coordinates for artifacts are global coordinates. For example, Listing 8-6 shows a portion of the map file for SRI's Artificial Intelligence Center.

```
;;
;; Map of a small portion of the SRI Artificial Intelligence Center
;;
;;               X     Y    Th  Length Width
CORRIDOR (1)    2000, 3000, 0,  3500,  800
CORRIDOR (2)    1000, 2000, 90, 6000, 1000
DOOR (3)        3000, 2600, 90,       1000
DOOR (4)        1500, 1000, 180,      1000
JUNCTION (5)    1500, 3000, 0,         800
WALL (6)        1000, 4000, 0,  1000
WALL (8)         800, 3500, 90,  400
WALL             800, 4500, 90,  400
```

Listing 8-6.

The `CORRIDOR` lines define a series of corridor artifacts. The number in parentheses is the (optional) 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.

`JUNCTION` entries are like doors, but delimit where corridors meet. T-junctions should have three junction artifacts, and X-junctions four. It's not necessary to put in any junctions, but they can be useful in keeping the robot registered (see below).

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.

The map file, when loaded into a Saphira client using the Files/Load Map menu (or the function `sfLoadMapFile`), creates the artifact structure shown in Figure 8-5-5. For illustration, the defining point of the artifact is also shown as a small circle with a vector. These points will not appear in the Saphira window.

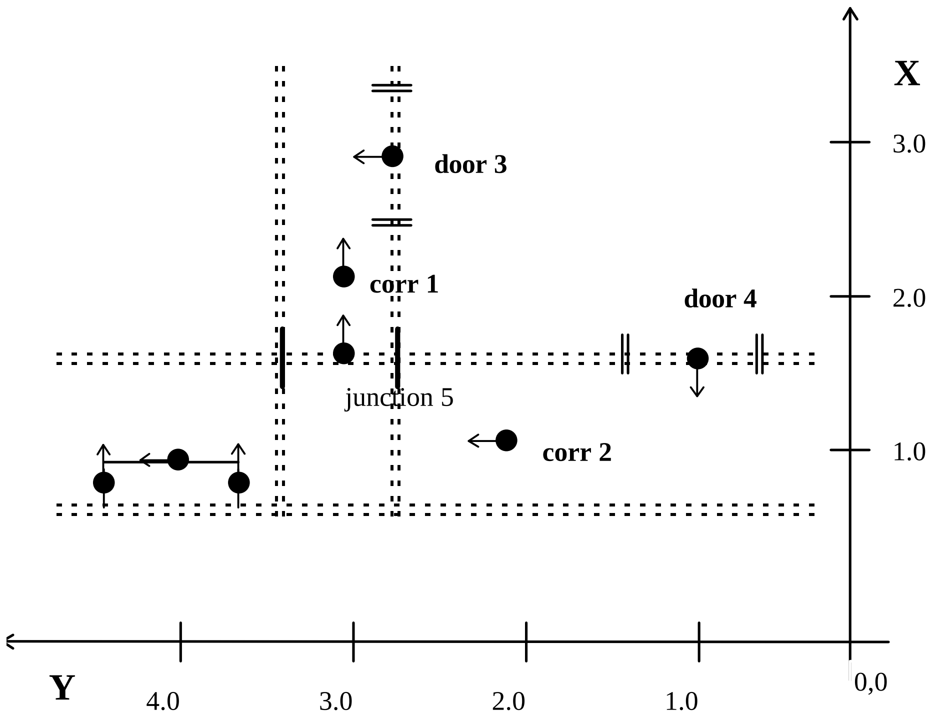


Figure 8-5. Sample map created from the map file above, as shown in a Saphira client. Corridor artifacts display with double dotted lines; doors display with double solid lines; walls display as single solid lines; junctions as pairs of solid lines. Numbers are the artifact ID's. For illustration, the defining vector for each artifact is shown.

Note that a map represents artificial structures in the Saphira client, in the same way that latitude and longitude lines are artifacts in global maps and are not found on the earth's surface. The robot or simulator will not pay attention to these lines, because they are internal to the client. This can be a useful feature. For example, a corridor is conceptually a straight path through an office environment; even where it has door openings or junctions with other corridors, you can imagine the corridor walls as extended through these areas. The robot can still go "through" the artifact corridor sides at these points. The registration micro-tasks (described below) use the map artifacts as registration markers, matching sensor data from the sonars against this internal model to keep the robot registered on the map.

Obstacles within corridors, such as water coolers or boxes, can be represented using wall structures, such as the one in corridor 2.

```
int sfLoadMapFile (char *name)
 int sfSaveMapFile(char *name)
 char *sfMapDir
 int sfDeleteMapArtifacts(void)
 int sfLoadWorldFile(char *name)
```

The `sfLoadMapFile` function loads a map file `name` into Saphira. It returns 0 if successful; -1 if the file cannot be found. Any map file errors are reported in the message window, but note that only the last one is displayed long enough to be read.

If the argument to the map file functions is a relative directory path (e.g., `maps/mymap`), then Saphira will use the map directory `sfMapDir` as a prefix for this path. By default, `sfMapDir` is set to the directory `maps` in the top level of the Saphira distribution.

Loaded artifacts are added to any map artifacts already in the system. To delete all map artifacts, use the `sfDeleteMapArtifacts` function. An individual artifact can be deleted using its ID number (see Section 8.7).

The current client map can be saved to a file using `sfSaveMapFile`. The saved file is in map file format, so it can be read in using `sfLoadMapFile`.

When using the simulator with Saphira clients that have maps, it is useful to have the simulated world correspond to the map. Unfortunately, the format of simulator world files is different from map files, and currently no utility exists to convert map files into simulator world files. They must be created by hand.

A simulator world file can be loaded into the simulator either by the menu commands in the simulator, or by the `sfLoadWorldFile` command issued from a client connected to the simulator.

8.10.2 Map Registration

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 micro-task that performs registration is called `test matching`. In the sample Saphira client, this micro-task is invoked by the function `sfInitRegistrationProcs`. To disable registration, either do not start the `test matching` micro-task, or set its state to `sfSUSPEND`, using `sfTaskSuspend`.

The registration micro-tasks will preferentially match a complete doorway or corridor, if it has constructed the corresponding hypothesis from sonar readings and a suitable map artifact is close by. Otherwise, it will attempt to match single walls or sides of doorways. Matching corridors and walls helps keep the robot's angle aligned, and also its sideways distance. Finding doors helps it to align in a forward/back direction. Both of these are important to keeping the robot registered, but the angle registration is critical, because the robot's dead-reckoned position quickly deteriorates if its heading is off.

Corridor junctions can also be important landmarks for registration. Ideally, junctions should be automatically generated from intersections of corridors. However, this capability does not currently exist, and you have to put them in by hand. In Figure 8-5, Junction 5 is only one of three possible junction artifacts for the corridor intersection. It will be used to register the robot as it moves down Corridor 2, just as it would be to move through a doorway. To register the robot as it moves in Corridor 1, you would have to put in the other two junctions at right angles to Junction 5.

8.10.3 Map Element Creation

A by-product of the registration micro-task 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`.

Map elements can also be created by hand, using the artifact creation functions of Section 8.7.

8.11 File Loading Functions

This section describes functions for loading Colbert files, shared object files, parameter files, and simulator world files. Map file loading functions can be found in the previous section.

```
int sfLoadEvalFile(char *name)
 char *sfLoadDirectory
 int sfLoadParamFile(char *name)
 char *sfParamDir
 int sfLoadWorldFile(char *name)
```

`sfLoadEvalFile` loads a Colbert language file or loadable shared object file into Saphira. The load directory, `sfLoadDirectory`, is set by default to the value of the environment variable `SAPHIRA_LOAD` if it exists, or to the working directory if it doesn't. The load directory is used as a prefix on relative path names; absolute path names are always loaded with no modification. All load functions return 0 if successful, and -1 if not.

Parameter files for different robot servers can be loaded with the `sfLoadParamFile` function. Bewcause Saphira clients autoload the correct parameter file when they connect to a robot server, the user should call this function only in special circumstances. The load directory is in `sfParamDir`, which is set by default to the directory `params` at the top level of the Saphira distribution.

A Saphira client, if it is connected to the simulator, can cause the simulator to load a world file through the `sfLoadWorldFile` command.

8.12 Colbert Evaluator Functions

Several library functions add functionality to the Colbert evaluator, by linking the evaluator to native C functions, variables, and structures. For examples, see Section 1 on the Colbert language.

```
int sfAddEvalFn (char *name, void *fn, int rtype, int nargs, ...)
 int sfAddEvalVar (char *name, int type, void *v)
 int sfAddEvalConst (char *name, int type, ...)
 int sfAddEvalStruct (char *name, int size, char *ex, int numslots, ...)
```

These functions all return the Colbert index of the defined Colbert object. Generally this index is not useful in user programs, and can be ignored. The exception is the `sfAddEvalStruct` function, which returns the type index of the Colbert structure.

`sfAddEvalFn` makes the native C function `fn` available to Colbert as `name`. The return type of the function is `rtype`, and the number of parameters is `nargs`. The additional arguments are the types of each of the parameters. A Colbert function may have a maximum of seven parameters. Functions with a variable number of parameters should set `nargs` to the negative of the number of fixed parameters and give the types of the fixed parameters.

`sfAddEvalVar` makes a native C variable of type `type` available to Colbert as `name`. A pointer to the variable should be passed in `v` as type `(fvalue *)`. For example, if the variable is `myVar`, use `(fvalue *)&myVar`. The value of the C variable can be modified from Colbert.

`sfAddEvalConst` defines a constant in Colbert with name `name` and type `type`. The function should have one additional argument, which is the constant value, either an integer, floating-point number, or pointer.

`sfAddEvalStruct` makes a native C structure available to Colbert with name `name`. The size of the structure, in bytes, should be given in `size`. A pointer to an example structure should be passed in `ex`. The number of structure elements is given by `numslots`. The additional arguments are triplets describing the elements, in any order. A sample element description follows:

```
"x", &ex.x, sfFLOAT,
```

Here `x` is the Colbert name of the element, `&ex.x` is a pointer to the example element, and `sfFLOAT` is an integer describing the type of the element.

This function returns the Colbert index of the structure type, which should be saved for future reference by the program.

```
int sfINT, sfFLOAT, sfSTRING, sfVOID, sfPTR
 int sfSrobot, sfSpoint
 int sfTypeRef (int type)
 int sfTypeDeref (int type)
```

These constants and functions refer to Colbert type indices, which are integers. The first set of constants are the basic type indices for Colbert; the second set are predefined structures. `sfTypeRef` returns the index of a pointer to its argument, while `sfTypeDeref` returns the index to the type referenced by its argument, or 0 if its argument is not a pointer type index.

```
void sfAddHelp(char *name, char *str)
 char *sfGetHelp(char *name)
```

These functions are the C interface to Colbert's help facility. `SfAddHelp` adds the string `str` as a help string for the Colbert object named `name`. It puts it in alphabetical order, so that searching for help entries is easier. The help string may have embedded formatting commands such as "`\t`" and "`\n`".

`sfGetHelp` returns the help string associated with `name`, or `NULL` if there is none.

```
void sfLoadInit(void)
 void sfLoadExit(void)
```

When a shared object file is loaded, the special function `sfLoadInit`, if it is defined in the file, is evaluated at the end of the load. Colbert variables, functions, and structures are typically defined here.

When a shared object file is unloaded or reloaded, the special function sfLoadExit, if it is defined in the file, is executed. This function should disable activities that reference C functions and variables defined in the file.

Note that these functions can be defined in each loaded file. In MS Windows, they must be declared EXPORT.

8.13 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. If you start up the Saphira OS with `sfStartup,` do not use these functions to parse information packets or send motor control commands.

```
int sfConnectToRobot(int port, char *name)
 char *sfRobotName
 char *sfRobotClass
 char *sfRobotSubclass
```

(This Saphira function tries to open a communications channel to the robot server on port type `port` with name `name`. It returns 1 if it is successful; 0 if not. This function also is available as the `connect` command in Colbert.

Table 8-9. Port types and names for server connections.

Classification	**Name**	**Description**
Port types	`sfLOCALPORT`	Connects to simulator on the host machine

	`sfTTYPORT`	Connects 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 Table 8-10, 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 8-10. Robot names and classes.

Structure	Explanation
`(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 either `pion1` (Pioneer I) or `pionat` (Pioneer AT).

`void sfDisconnectFromRobot (void)`

This structure 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 7.2 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 an integer packed from two bytes, an 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 (int type)
```

`sfProcessClientPacket` parses a client packet into the `sfRobot` structure and sonar buffers. Typically, a client will call `sfWaitClientPacket` or `sfHaveClientPacket` to be sure a packet is waiting to be parsed. The argument to `sfProcessClientPacket` is a byte, the type of the packet. This byte can be read using `sfReadClientByte`. By examining this byte, the client can determine if it wishes to parse the packet itself, or send it on to `sfProcessClientPacket`.

```
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 developed by Newton Labs, Inc. and 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 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:

BLOB_MODE	0
BLOB_BB_MODE	2
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 this 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 start-up, the vision system channels are set to BLOB_MODE. (The processing performed in BLOB_MODE, BLOB_BB_MODE, and LINE_MODE is explained in the FTVS manual.)

As Table 9-1 shows, several FTVS parameters affect the processing in line mode.

Table 9-1. FTVS parameters used to determine a line segment.

Parameter	**Description**
`line_bottom_row`	First row for line processing
`line_num_slices`	How many rows are processed
`line_slice_size`	How many pixels thick each row is
`line_min_mass`	Number of pixels needed to

These parameters can be set using a command such as the following:

```
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 (see Listing 9-1).

```
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 */
};
```

Listing 9-1. Saphira vision information structure.

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 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 0, 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.

The following global variables hold information for each channel: `extern struct vinfo sfVaInfo, sfVbInfo, sfVcInfo.`

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

```
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.

This sequence sets up parameters of the vision system, putting all channels into BLOB_BB_MODE and initializing line parameters:

`void setup_vision_system(void)`

This one 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:

`int found_blob(int channel, int delta)`

If no blob is found with these parameters, it returns -1000.

`void draw_blobs(void)`

This is the 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 at which 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)`

This command defines the activity for turning left until a blob is found in the center of the image on channel A, or until 20 seconds elapses.

`void search_and_go_blob(void)`

This command defines the activity for finding a blob (using `find_blob`) on channel A, then approaching it. It uses sonars to detect when it is close to the blob.

10 Parameter Files

This section describes the parameter files used by the Pioneer simulator and Saphira client to describe the physical robot and its characteristics.

10.1 Parameter File Types

Pioneer robots have four parameter files:

```
pioneer.p
psos41x.p
psos41m.p
psosat.p
```

The sequence `41` refers to PSOS versions equal to or greater than PSOS version 4.1. Early versions of the Pioneer that have not been upgraded to at least version 4.1 should use the `pioneer.p` parameter file. These Pioneers do not send an autoconfiguration packet; therefore, Saphira clients by default are configured for pre-PSOS 4.1 robots and will correctly control these robots without explicitly loading a parameter file.

Pioneer robots with PSOS 4.1 or later send an autoconfiguration packet on connection that tells the Saphira client which parameter file to load. Pioneers made before August 1996 use old-style motors, and these load `psos41x.p`. Those made after this date use new-style motors, and load `psos41m.p`. The only difference is in some of the conversion factors for distance and velocity.

The Pioneer AT has its own parameter file, `pionat.p`. The only change from `psos41m.p` is that the robot is larger than the other Pioneers.

The B14 and B21 robots from RWI also have parameter files, `b14.p` and `b21.p`.

10.2 Sample Parameter File

The sample parameter file in Listing 10-1 illustrates most of the parameters that can be set. This is the file `psos41m.p`. An explanation of the parameters is given in Table 10-1, below.

```
;;
;; Parameters for the Pioneer robot
;; New motors
;;
AngleConvFactor   0.0061359 ; radians per encoder count diff (2PI/1024)
DistConvFactor 0.05066   ; 5in*PI / 7875 counts (mm/count)
VelConvFactor     2.5332    ; mm/sec / count  (DistConvFactor * 50)
RobotRadius       220.0     ; radius in mm
RobotDiagonal      90.0     ; half-height to diagonal of octagon
Holonomic         1         ; turns in own radius
MaxRVelocity      2.0       ; radians per meter
MaxVelocity       400.0     ; mm per second

;;
;; Robot class, subclass
;;
Class          Pioneer
Subclass       PSOS41m
Name           Erratic

;; 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
;;
```

Listing 10-1. The example parameter file, `psos41m.p,` shows how to set most Saphira parameters.

```
RangeConvFactor           0.1734 ; sonar range mm per 2 usec tick
;;
SonarNum 7
;;         #   x   y   th
;;------------------------
SonarUnit  0  100 100  90
SonarUnit  1  120  80  30
SonarUnit  2  130  40  15
SonarUnit  3  130   0   0
SonarUnit  4  130 -40 -15
SonarUnit  5  120 -80 -30
SonarUnit  6  100 -100 -90
SonarUnit  7    0   0   0

;; Number of readings to keep in circular buffers
FrontBuffer 20
SideBuffer  40
```

Listing 10-2.

Floating-point parameters can be in any standard format and do not require a decimal point. Integer parameters may not have a decimal point. Strings are any sequence of non-space characters.

Table 10-1. Functions of Saphira parameters.

Parameter	**Type**	**Description**
`AngleConvFactor`	float	Converts from robot angle units (4096 per revolution) to radians.
`VelConvFactor`	float	Converts from robot velocity units to mm/sec
`DistConvFactor`	float	Converts from robot distance units to mm
`DiffConvFactor`	float	Converts from robot angular velocity to rads/sec
`RangeConvFactor`	float	Converts from robot sonar range units to mm
`Holonomic`	integer	Value of 1 says the robot is holonomic (can turn in place); value of 0 says it is nonholonomic (front-wheel steering). Holonomic robot icon is octagonal; nonholonomic is rectangular.
`RobotRadius`	float	Radius of holonomic robot in mm.
`RobotDiagonal`	float	Placement of the horizontal bar indicating the robot’s front, in mm from the front end. (Sorry about the name.)
`RobotWidth`	float	Width of nonholonomic robot, in mm.
`RobotLength`	float	Length of nonholonomic robot, in mm.
`MaxVelocity`	float	Maximum velocity of the robot, in mm/sec.
`MaxRVelocity`	float	Maximum rotational velocity of the robot in degrees/sec.
`MaxAcceleration`	float	Maximum acceleration of the robot in mm/sec/sec
`Class`	string	Robot class: `pioneer`, `b14`, `b21`. Not case-sensitive. Useful only for the simulator, which will assume this robot personality. The client gets this info from the autoconfiguration packet.
`Subclass`	string	Robot subclass. For the Pioneer, indicates the type of controller and

		body combination. Values are psos41m, psos41x, or pionat. Not case-sensitive. Useful only for the simulator, as for the Class parameter.
Name	string	Robot name. Useful only for the simulator, as for the Class parameter.
SonarNum	integer	Number of active sonars.
SonarUnit	n,x,y,th	Description sonar unit n. The x,y,th arguments describe the pose of the sonar on the robot body, relative to the robot center. Provide one such entry for each active sonar unit. Used by both the simulator and client.
FrontBuffer	integer	Number of front sonar readings to keep. Higher values mean the robot will be more sensitive to obstacles but slower to get rid of moving obstacle readings.
SideBuffer	integer	Number of side sonar readings to keep. Higher values mean the interpretation routines can find longer side segments.

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 non-blank 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 11-1).

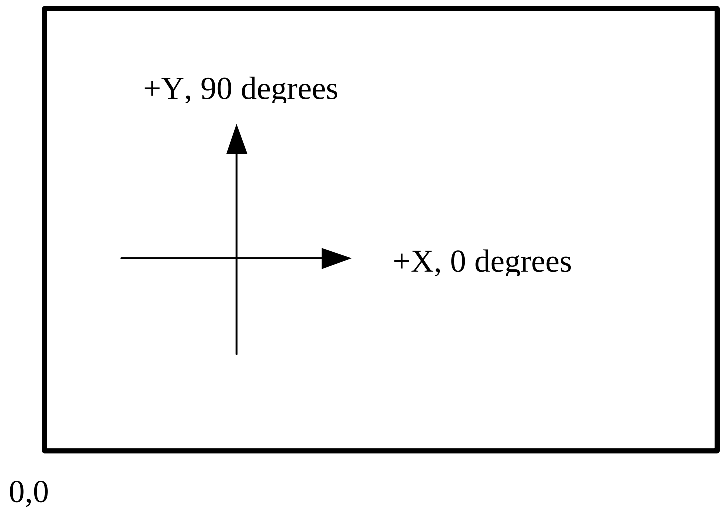


Figure 11-1. Coordinate system for world definition.

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.

Listing 11-1 is a fragment of the `simple.wld` world description file found in Saphira's `worlds` directory.

```
;;; 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
```

Listing 11-1. Fragment of the `simple.wld` world description file found in Saphira's `worlds` directory.

12 Saphira API Reference

Artifacts	*Page*
void sfAddAngle	112
void sfAdd2Angle	112
void sfAddPoint(point *p)	110
void sfAddPointCheck(point *p)	110
void sfChangeVP(point *p1, point *p2, point *p3)	112
point *sfCreateGlobalPoint(float x, float y, float th)	110
point *sfCreateLocalPoint(float x, float y, float th)	110
point *sfFindArtifact(int id)	110
point *sfGlobalOrigin	110
void sfMoveRobot(float dx, float dy, float dth)	113
void sfNormAngle	112
void sfNorm2Angle	112
void sfNorm3Angle	112
void sfPointBaricenter(point *p1, point *p2, point *p3)	112
float sfPointDist(point *p)	112
float sfPointDistPoint(point *p1, point *p2)	112
void sfPointMove(point *p1, float dx, float dy, point *p2)	113
float sfPointNormalDist(point *p)	112
float sfPointNormalDistPoint(point *p, point *q)	112
float sfPointPhi(point *p)	110
float sfPointXo(point *p)	112
float sfPointXoPoint(point *p, point *q)	112
float sfPointYo(point *p)	112
float sfPointYoPoint(point *p, point *q)	113
void sfRemPoint(point *p)	110
point *sfRobotOrigin	110
void sfSetGlobalCoords(point *p)	110
void sfSetLocalCoords(point *p)	110
void sfSubAngle	112
void sfSub2Angle	112
void sfUnchangeVP(point *p1, point *p2, point *p3)	112

Behaviors

BEHCLOSURE sfFindBehavior(char *name)	**Error! Bookmark not defined.**
BEHCLOSURE sfInitBehavior(behavior *b, int priority, int running, ...)	**Error! Bookmark not defined.**
BEHCLOSURE sfInitBehaviorDup(behavior *b, int priority, int running, ...)	**Error! Bookmark not defined.**
int sfBehaviorControl	57
void sfBehaviorOff(BEHCLOSURE b)	**Error! Bookmark not defined.**
void sfBehaviorOn(BEHCLOSURE b)	**Error! Bookmark not defined.**
void sfKillBehavior(BEHCLOSURE b)	**Error! Bookmark not defined.**
void sfSetBehaviorState(BEHCLOSURE b, int state)	**Error! Bookmark not defined.**

Behaviors; Predefined Saphira

behavior *sfAttendAtPos	**Error! Bookmark not defined.**
behavior *sfAvoidCollision	**Error! Bookmark not defined.**
behavior *sfConstantVelocity	60

13 Index

14 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 6.1f, August 1998