Saphira Software Manual

Saphira Version 8.0a
Saphira/Aria integration

SRI International, Menlo Park, California

Contents

List of Tables

List of Figures

1 Saphira Software & Resources

This manual is intended to be an introduction to the Saphira robot control system for application programmers. It serves as an overview of Saphira, as well as a guide to running the standard Saphira client. It also contains reference material for the Saphira API.

1.1 Documentation and Sample Programs

Robot programming is a complex task, and this manual is an overview guide and reference. Further documentation and examples can be found in the documentation directory of the Saphira distribution. In particular, there are tutorials on many aspects of Saphira programming.

1.1.1 ARIA Documentaion

Aria has a number of examples and test programs, as well as documentation of the class hierarchy generated from the header files and source code. Aria documentation will be very useful to those wishing to write Saphira programs.

1.1.2 Manuals and Tutorials

Various specific aspects of Saphira and Aria are discussed in detail in manuals and tutorials available in the Saphira `docs` directory. Tutorial programs, complete with source code, are in the subdirectories of the `tutorial` directory.

Document	Contents	Tutorial code
`actions.pdf`	Describes the different modes of robot control in Saphira/Aria, including behavioral action schemas.	`tutor/movit` `colbert/direct.act`
`loadable.pdf`	Tutorial on compiling and loading C++ program files under Saphira.	`tutor/loadable`
`colbert.pdf`	Technical paper on the Colbert robot programming language.	

`colbert-user.pdf`	User's Manual for the Colbert robot programming language.	`tutor/movit`
		`tutor/loadable`
		`colbert/direct.act`
		`colbert/bump.act`

`motion.pdf`	Technical description of probability-based robot movement and localization.	`tutor/sample-move`
		`tutor/sample-update`
		`tutor/sonardist`

	Tutorial programs for synchronous and asynchronous tasks in Saphira.	`tutor/task`
	Tutorial programs for obstacle avoidance and wall-crawling behaviors.	`tutor/crawl`

1.1.3 Online API

The Saphira and Aria class and function API's are now documented using the Doxygen documentation system. HTML versions are available in `Saphira/docs/html` and `Aria/docs/reference/html`.

1.1.4 Class Notes

Saphira/Aria is the system used in Stanford's Robot Programming Lab, CS225B. You can find more information about the class at the website cs225b.stanford.edu.

1.2 Saphira and Aria

Saphira 8.x is based on Aria, a low-level robotic control system written by ActivMedia Robotics. Aria contains much of the lower-level functionality previously incorporated into Saphira, as well as new capabilities such as multiple-robot control. A good understanding of Aria is necessary for working with Saphira, since Saphira will use many of Aria's classes. Documentation on Aria is available from ActivMedia Robotics.

While Aria is meant to provide flexible low-level access to robot capabilities, Saphira simplifies the developer's task by providing a convenient interface to many of these facilities.

1.3 Saphira Client/Server

Saphira is a robotics application development environment written, maintained, and constantly updated at SRI International's 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. The Saphira library integrates a number of useful Aria 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 the Colbert control executive, and a map-based localization and navigation 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 through the Aria robot interface.

The Saphira client library is available for Microsoft Windows 98/ME/2000, and for Linux systems. 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, as well as the tutorials and supporting documentation.

1.4 Colbert Robot Programming Language

Starting with Version 6.x, 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 C++ code. Users may program interactively in Colbert, which makes many of the Saphira API functions available in the runtime environment.

1.5 Behaviors

Rule-based reactive control programs, called *behaviors*, are implemented using Aria's `ArAction` class. Saphira maintains a list of behavior schemas that can be invoked from Colbert or C++ programs. Behavior output can be combined in flexible ways, using Aria's `ArResolver` facility.

1.6 Localization and Navigation

Saphira incorporates sophisticated algorithms for some difficult robot tasks.

Localization is the task of keeping track of robot position within an environment. Saphira has facilities for both sonar and laser rangefinder based localization. It uses efficient probabilistic techniques developed recently by Dieter Fox and his colleagues.

Navigation is the task of determining a good path for the robot to follow to a goal, and also keeping the robot out of trouble as it moves. Saphira uses the *gradient method* for this task. Developed by Kurt Konolige, it is an optimal realtime path-planner for the robot.

1.7 Maps

Saphira uses line-drawing maps of the environment for localization and navigation. These maps can be input by hand from textual coordinate files for simple environments. ActivMedia Robotics has two more advanced map input methods. A *graphical mapping interface* allows the user to interactively create maps using a GUI tool. For automatic map-building, ActivMedia Robotics has deployed Steffen Gutmann's *ScanStudio*, a sophisticated algorithm that builds maps automatically from laser range scans.

1.8 Robot Simulator

Saphira comes with a software simulator of a physical robot and its environment. This feature allows developers to debug applications conveniently on a computer without using a physical robot.

The simulator has realistic error models for the sonar sensors, laser range-finder, and wheel encoders. Even its communication interface is the same as for a physical robot, so developers 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.

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.

1.9 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: Pentium or 486-class PC with Microsoft Windows 98/ME/2000 or Linux operating system
- ✓ Open communication port (TCP/IP or serial)
- ✓ 20 megabytes of hard-disk storage
- ✓ PKUNZIP (PCs), GUNZIP (PCs and UNIX), StuffIt Lite, or compatible archive-decompression software

Necessary for program development:

- ✓ C++-program source-file editor and compiler. *Note: under MS Windows, Saphira supports only Microsoft's Visual C/C++ software*.

1.10 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(.exe)` demonstration program, Colbert, simulator, and accompanying C libraries, come stored as a compressed archive of directories and files either on a CD ROM, or at the ActivMedia Internet site. Each archive is configured and compiled for a particular operating system, such as Windows98/2000 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 MSW 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 unpack the tar file in a standard directory such as `/usr/local` or another publicly accessible directory, and set the appropriate permissions for access and use by your robotics groups. Copy the Saphira archive to that directory, then uncompress and untar the Saphira archive. For example, with Linux the command is:

```
tar -zxvf linux80a.tgz
```

The extraction process will create two directories, `Saphira` and `Aria`. These directories must be children of the same directory..

For MSW, uncompress the ZIP archive; the location of the files is up to you. On extraction, the files in the ZIP archive will create two top-level directories, `Saphira` and `Aria`. These directories must be children of the same directory.

Table 1-1 below shows the general structure of the Saphira directories.

IMPORTANT NOTICE!

All Saphira operations require that the environment variables `SAPHIRA` and `ARIA` be set to their respective top-level directories, e.g., `/usr/local/Saphira` and `/usr/local/Aria` on a Unix system (note that the directory name does not have a final slash), or `c:\Saphira` and `c:\Aria` 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/Aria, your environment variables will still be set correctly, since the new distribution will overlay the previous one.

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    (bash shell)
setenv SAPHIRA /usr/local/Saphira    (csh shell)
```

In MS Windows 95/98/ME, assuming the top-level Saphira directory is `c:\Saphira`, add the following line to the file `C:\AUTOEXEC.BAT`:

```
SET SAPHIRA=C:\Saphira
```

In Windows NT/2000/XP, go to Start/Settings/System, and click on the Environment tab. Add the variable `SAPHIRA` and `ARIA` 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 `Saphira/lib/libsf.so` and `Aria/lib/libAria.so`. 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=${SAPHIRA}/lib:$(ARIA)/lib
  ```

  A second method is to copy the library file into the standard library directory, usually `/usr/lib`.
- Under MS Windows, the shared libraries are `Saphira\lib\sf.dll` and `Aria\lib\aria.dll`. You must copy or move this file to the standard MS Windows system directory. In Windows 95/98/ME this is `C:\Windows\System`; in Windows NT/2000/XP 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.

1.11 Saphira Quick Start

To start the Saphira client demonstration program, navigate to inside the `lib/` directory and execute the program named `saphira(.exe)`. For instance, use the mouse to double-click the `saphira.exe` icon inside the `Saphira/lib/` folder on your MS Windows 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` and `ARIA` environment variables.

The Saphira client window will appear, with a graphics display of the robot internals, a text information

```
Saphira/
  lib/
    saphira(.exe)         Saphira/Colbert runtime application
    pioneer(.exe)         simulator
    sf.dll, libsf.so      DLL or shared object library
  colbert/                Colbert language samples
  ohandler/
    include/              header files
    src/                  source files
  maps/                   Saphira example maps
  tutor/                  tutorial programs
    movit/                direct and behavioral examples
    …
  worlds/                 simulator world files
  readme                  explanation text file
  update                  comparison of versions
  license                 operation license

Aria/
  lib/                    Aria libraries
  docs/                   Aria documentation
  include/                Aria header files
  src/                    Aria source files
```

Table 1-1 Installed directories for Saphira/Aria Version 8.0a

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/lib/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`. Or, use the pulldown `Connect` menu at the top of the Saphira window.

If you're using the simulator, you can connect using `connect`, 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 `Saphira/lib/` directory. 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 `Saphira/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.

1.12 Additional Resources

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

1.12.1 FTP Software Archive

ActivMedia has a server connected full-time to the Internet, where customers may obtain Saphira software and support materials. Access is restricted to *Activ*Media customers, including Pioneer, AmigoBot, PeopleBot, and MonsterBot owners.

The ActivMedia server name is:

`ftp.activmedia.com`

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

1.12.2 Saphira Newsgroup

*Activ*Media also maintain several special e-mail-based newsgroups for robot owners to share ideas, software, and questions. To find out more about these special newsgroups, send an e-mail message with your reply e-mail address as shown in Figure 1-2, below:

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

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

Figure 1-2. Follow this format to find out more about newsgroups available through *Activ*Media.

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

1.12.3 SRI Saphira Web Pages

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

- tutorials and other documentation on various parts of Saphira
- class projects from Stanford CS225B, *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`**.

1.12.4 Support

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

`support@activmedia.com`

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

1.12.5 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.

The Aria system was created by the ActivMedia software team. The primary architect is Matt LaFary. Major contributions come from Chris Newton and Joe XXXX.

2 Saphira and Aria System Overview

Saphira is an architecture for mobile robot control. Originally, it was developed for the research robot Flakey[1] 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.[2]

With Saphira 8.x, the Saphira system has been split into two parts. Lower-level routines have been re-organized and re-implemented as a separate software system, Aria. Aria is developed and maintained by ActivMedia Robotics. It is a production-level system for robot control, based on an extensive set of C++ classes. The class structure of Aria makes it easy to expand and develop new programs: for example, to add new sensor drivers to the system.

The Saphira/Aria system can be thought of as two architectures, with one built on top of the other. The *system architecture*, implemented entirely in Aria, 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 and Aria share the control architecture duties, with Aria providing the basic elements of action and sensor interpretation. Saphira's contribution to the 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 behavioral control regime or a 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 and Aria. More in-depth information can be found in the documentation at the SRI Saphira web site (**http://www.ai.sri.com/~konolige/saphira**) and ActivMedia's Aria website.

2.1 System Architecture

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

---

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

[2] 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**).

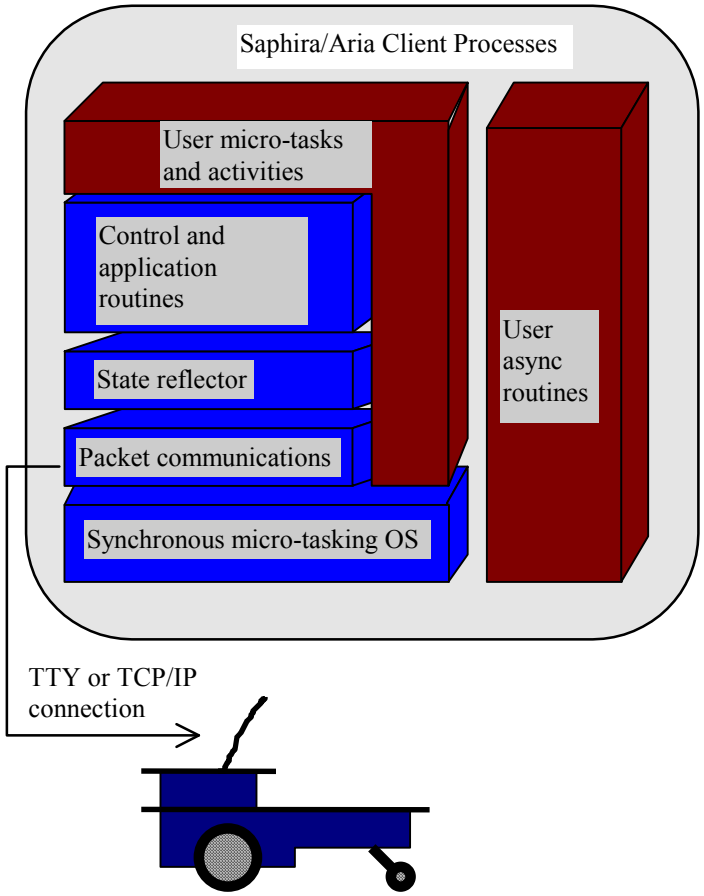

Figure 2-1 Saphira/Aria System Architecture.

Blue areas represent routines in the Saphira/Aria libraries, redroutines 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.

2.1.2 Micro-Tasking OS

The Saphira/Aria architecture is 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 500 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/Aria library routines, that runs synchronously every 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/Aria 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/Aria 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 synchronous 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/Aria system architecture, without affecting the real-time nature of robot control.

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

Packet Communications

Aria 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 **Error! Reference source not found.**). 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. Aria 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 Aria's performance will degrade.

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 Control Architecture

The control architecture is built on top of the state reflector (Figure 2-1). It consists of a set of micro-tasks and asynchronous tasks 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.

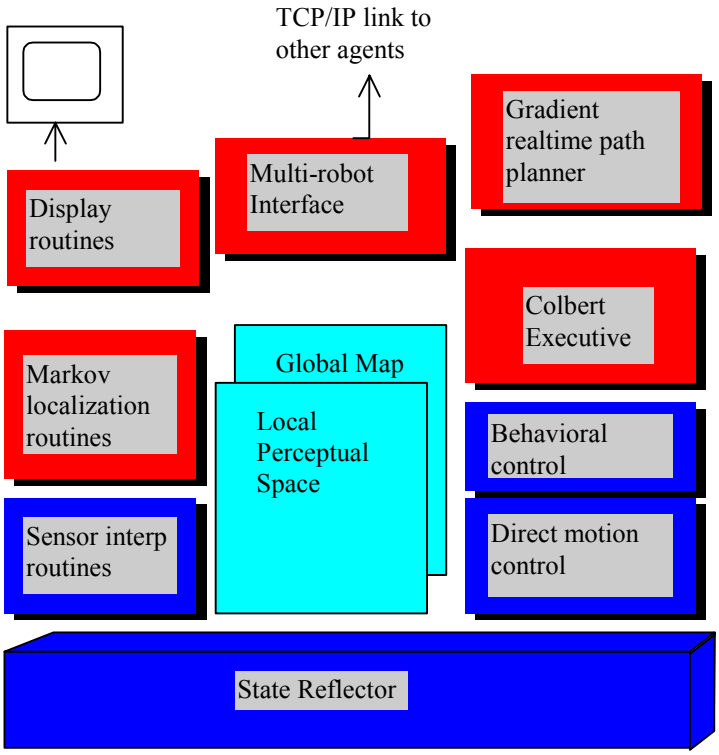

Figure 2-1. Saphira/Aria Control Architecture

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. Markov localization routines link the robot's local sensor readings to its map of the world, and the Colbert Executive sequences actions to achieve specific goals. The multi-robot interface links the robot to other robots using TCP/IP connections. Aria system is in blue, Saphira in red.

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.

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 **Error! Reference source not found.**).

Behavioral Control

For more complicated motion control, Aria provides a facility for implementing *behaviors* as sets of 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 8.x includes several major changes in behavior management. Aria implements a general behavior architecture in which behaviors are C++ objects. The interaction among behaviors is implemented by a *resolver* class. Aria provides several types of resolvers, and the user can define his own additional resolvers for particular applications. Behaviors are now integrated with Colbert activities, so that they appear as the leaves of an executing activity tree.

Behaviors can be turned on and off by sending them signals, either from the interaction window, or from the Activities window.

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.

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 and surface reconstruction are some of the routines that currently exist; all work with data reflected from the sonars, laser range-finders, and motion sensing.

Localization 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.

Localization is the process of keeping the robot's global location in an internal map consistent with sensor readings from the local environment. Saphira implements an efficient Markov Localization algorithm for taking information from sonars or laser range-finders, matching it to map structures in the GMS, then updating the robot's position.

Realtime, Optimal Path Planning

Saphira 8.x incorporates a new, efficient method for planning optimal paths in real time. The *Gradient Method*, developed at SRI International, operates with both map artifacts and current sensor information to generate optimal paths that move the robot safely through the environment.

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.

Multi-Robot Interface

Aria is a multi-robot control system, with a class structure set up to handle multiple instances of robot controllers. Currently, Saphira is oriented towards controlling a single robot. In the immediate future, we plan on providing access to Aria's multi-robot facilities through Saphira.

Additionally, we are working on providing a TCP/IP interface between robot controllers running on different physical robots. This interface will tie together Saphira/Aria clients, enabling them to form a distributed robot control system.

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 and Aria (`sf.dll/aria.dll` or `libsf.so/libAria.so`). You should have installed these as directed in Section 1.10.

The Saphira client will initialize an interface window showing the LPS (see Figure 2-2). The robot is near the center of the display, which shows the sonar information returned to the client program. 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(.exe)` has only a bare set of micro-tasks loaded. 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 `startup.act` in the current load directory, which by default is `$(SAPHIRA)/colbert`. The initialization file loads the windowing system, and then 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 `startup.act` file in the `colbert/` directory. Be careful, though, to

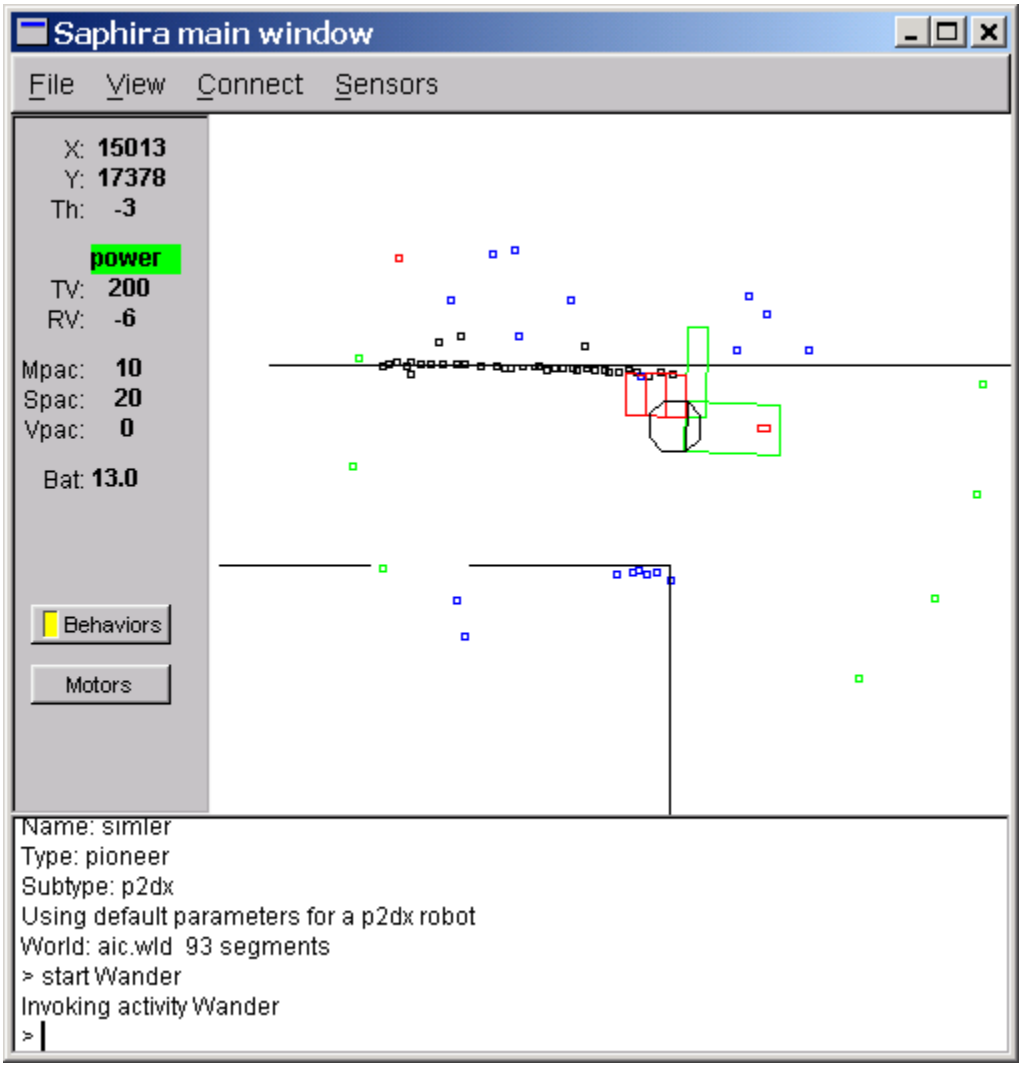

Figure 2-2 Saphira client Local Perceptual Space.

Small squares are sonar readings; green ones are the current sonar values, blue and black are history buffers. The small rectangle immediately in front of the robot is the angular setpoint. Larger green rectangles indicate areas of sensitivity for the behaviors controlling the robot. The lines are wall artifacts, from a loaded map. Information about the robot's position, velocity, and internal state are shown on the left.

have your own startup file load the windowing system as in `colbert/startup.act`, or you won't get a Saphira GUI.

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

2.12.1 Connecting to a Robot

As we mentioned earlier, connecting Saphira with either the simulator or the actual robot is similar. First, if you are using the simulator, make sure that the correct robot parameters are loaded (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 the robot manuals 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 XXX for details.
- Simulator connection. If you've started the simulator, it's listening on a local TCP/IP port. Type `connect`, which opens the local port to the simulator and starts things up. Or, choose the `Connect/Local` 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:

- 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.

2.12.2 Local Perceptual Space Display

The Saphira client's display contains most of the items likely to be found in the robot's LPS. 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:

Robot icon

The robot icon near 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 (the default), 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 6)

Sonar readings

Accumulated sonar readings appear on screen as small open rectangles, in blue and black. Current sonar readings are green rectangles. The number of accumulated sonar readings can be set by the user.

Control point

The small 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 controller adjusts its heading accordingly, trying to keep heading towards the control point.

Obstacle sensitivity areas

Several obstacle-avoidance behaviors draw large rectangles in the LPS, indicating areas of sensitivity for the behaviors. These rectangles can change color when an obstacle is detected.

2.12.3 Artifacts

Artifacts are internal representations of external objects or imaginary constructions, such as goal positions or map elements. In the LPS display of Figure 2-2 map walls are drawn as lines.

2.12.4 Information Area

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

Status (St)

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

Velocity (Tr, Rot)

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

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.

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.

Battery

The battery (`Bat`) voltage level on the server indicates when the robot needs to be recharged.

Behaviors

The `Behaviors` button is lit if behavioral actions are enabled for Saphira. Behavioral actions are overridden by direct actions, e.g., joysticking the robot using movement keys from the LPS window. Behavioral actions can be turned back on by pressing the `Behaviors` button.

Motors

The `Motors` button is used to enable the motors on a physical robot (the simulator's motors are always "enabled").

2.3.2 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.

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 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.

2.3.3 Menus

The main client window contains several pull-down menus. 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:

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 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.

The Disconnect option closes an open connection to the robot.

Files Menu

The `Files` menu has items for loading different kinds of files, and for exiting from the Saphira system.

`Load World File` brings up a dialog to load a world file map into Saphira. You can also use the Colbert `loadworld` command.

`Load Activity File` brings up a dialog to load a Colbert activity file into Saphira. You can also use the Colbert `load` command.

`Load Library File` brings up a dialog to load a compiled library file (DLL or shared object file) into Saphira. You can also use the Colbert `loadlib` command.

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

View Menu

The View menu control various aspects of the display windows.

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

The Rate item is a 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.

Robot Onscreen, if checked, keeps the viewport of the display window so that the robot is always visible.

Robocentric, if checked, changes the display so that everything is viewed from the perspective of the robot, which stays centered and pointing up in the middle of the window. Global mode (unchecked) shows the robot wandering in the global coordinate system.

Robot Visible and Artifacts Visible turn on or off the drawing of the robot and other artifacts to the window.

Activity Window, when selected, will bring up the Activity Window for viewing the state of Colbert activities and behavioral actions.

Sensors Menu

The Sensors menu has entries for the current sensor interpreters loaded for the Saphira client. Usually these are the sonar and laser range finder programs. Selecting either one will bring up a Sensor Window for controlling the action of the sensor buffers.

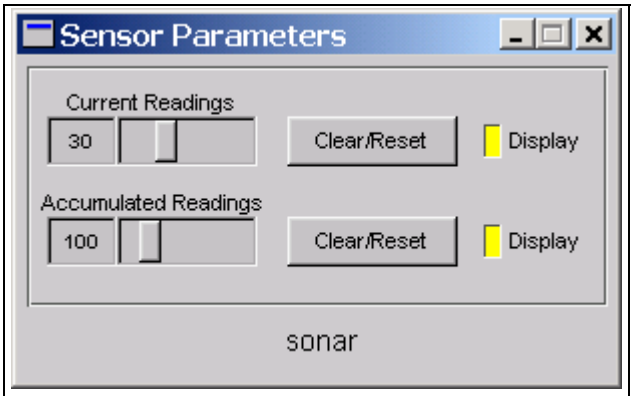

Figure 2-3 Sensor buffer dialog. Controls the sensor accumulation buffers.

The sensor buffer dialog windows give control over the size and display of the sensor history buffers. Generally, range sensors such as sonars accumulate readings into various history buffers, so that other routines can get a larger view of the world than that provided by single readings.

There are two types of history buffers. A *current buffer* holds the last N readings from the sensor, and the readings are replaced as new ones come in, on a first-in, last-out basis. For example, in a typical case the sonar current buffer will hold the last 30 sonar readings. If the readings are coming in at 20 per second, the buffer will hold the last 1.5 seconds of sonar readings. Since the readings are dynamically replaced, objects that come into view and then go away will not leave a lasting impression on the buffer. Current buffer readings appear in blue (for sonars) on the LPS window.

The *accumulation buffer* holds longer-term readings, and usually ones that are more certain of being correct. When new readings are added, old readings near them are erased; typically items stay in the buffer for longer amounts of time. Accumulation buffer readings appear in black (for sonars) in the LPS window.

The sliders in the sensor buffer dialog will change the number of slots in the buffers. The Clear/Reset item clears all of the sonar readings from the buffer. The display of the buffer can be toggled with the Display button.

2.3.4 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. The keys shown in Table 2-1 show motion control of the robot (*keyboard joystick*). These keys work *only* when the main Saphira window is active, that is, you have to left-click in the graphics window first.

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, there are in the Saphira API for intercepting keyboard actions and mouse clicks.

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.

2.3.5 Activities Window

Key	Action
i, ↑	Increment forward velocity
m, ↓	Decrement forward velocity
j, ←	Incremental left turn
l, →	Incremental right turn
k, space	All stop

Table 2-1 Keyboard joystick commands for the Saphira client.

Saphira's Activities window shows the state and relationship of all current Colbert activities and behavioral actions (Figure 2-4). Open it from the Activities menu item in the View menu of the main window.

The Activities window contains a scrolled list where each line has 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

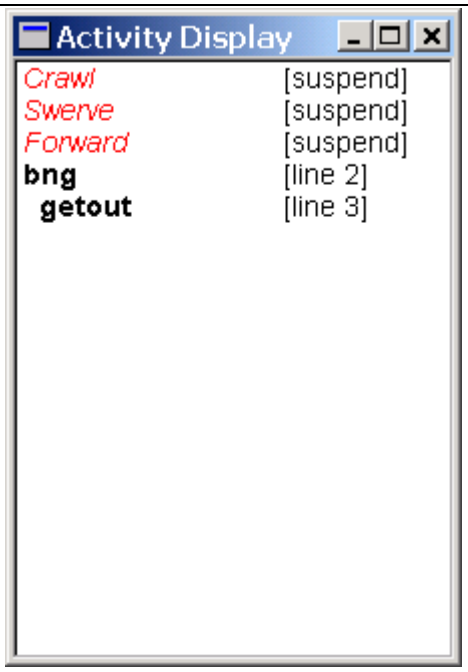

Figure 2-4 Activity Display Window. This window shows the state of currently executing activities (black) and behavioral actions (red).

2-4, the second activity `getout` is indented to show that it is a child of the first activity, `bng` (bump and go). These activities can be found in the file `Colbert/bump.act`; just type "`load bump`" to load it.

`Bng` monitors the state of the robot; when the robot bumps into something and the wheels stall, it starts up the `getout` activity to back up the robot and turn it away from the obstacle. That's the point at which this snapshot of the Activities Window was taken. There are also three behavioral actions that are started by `demo.act`, in a suspended state.

You may manually interrupt an activity by shift-clicking it with the mouse. If the activity is running, this will force it into the `suspend` state. Use the same action to reactivate an interrupted/suspended activity. This will invoke the `resume` state. Normally, an activity will respond to this state by reinitializing and starting its characteristic behaviors.

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/2000/XP). In UNIX, they are set from a shell using `setenv` or `export`.

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.
SAPHIRA_LOAD	Initial load directory for the Colbert evaluator. This directory is searched for the file startup.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/ttyS0 under Linux, and so on.
SAPHIRA_SERIALBAUD	Baud rate for serial connection. Defaults to 9600.
SAPHIRA_COMPORT	Local TCP/IP 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; use numbers greater than 8101. 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.

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

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 `lib/` directory. (By default, the simulator acts like the Pioneer II 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 a TCP/IP port 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 disconnect.

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

3.1.1 Listening on Other Ports

The simulator listens on a TCP/IP port for connections from a server. By default, this is port 8101. Only one simulator may be connected at a time to that port. 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_COMPORT` 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.

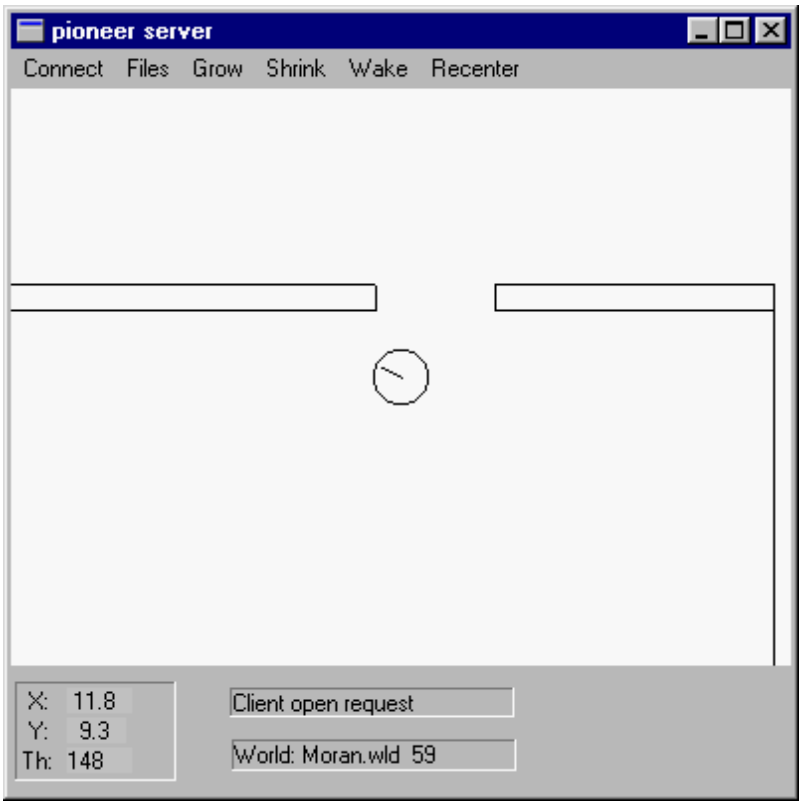

Figure 3-1. A sample window of the simulator.

3.2 Parameter File

The default operating parameters for the simulator are for the Pioneer II. 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 6 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

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 0, 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.

3.3 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.)

3.3.1 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.

3.3.2 Connect Menu

The Connect menu disconnects the simulator from an aborted client, or exits the simulator.

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.

The Exit item terminates the simulator. A connected simulator should be disconnected first from the client side, or it will cause the client to abort.

3.3.3 Display Menu (Grow, Shrink)

The Grow and Shrink menus or change the size of the display.

3.3.4 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.

3.3.5 Original Position

Pressing this button will return the simulated robot to the position given by the loaded map. This button is very useful for resetting while debugging.

3.3.6 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.

3.4 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.

3.5 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 Creating Loadable Files

This chapter describes how to create Saphira clients. As of version 8.0, we recommend using the standard Saphira client as the robot control client, and customizing its behavior by loading in shared object files containing more system and user code. The base client is `lib/saphira(.exe)`. The loaded files may be Colbert language interpreted files, or compiled C++ code in shared object files.

Also with version 8.0, Saphira is a completely object-oriented system (as is Aria). The language of Saphira is C++, and loadable files should also be written in this language (it is possible to use a mixture of C++ and C).

C++ programs can be compiled into object files using standard compilers, such as `gcc` or MS Visual C++. The header files in `ohandler/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 standard client `lib/saphira(.exe)` 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**).

Saphira uses the Aria libraries, and all Aria functions and classes are available in Saphira. Users should be familiar with the Aria documentation when writing Saphira programs, because often there will be a mixture of Saphira and Aria classes and functions.

There is a tutorial, *Compiling, Loading, and Debugging C++ Files*, which gives details about the compilation and debugging process for Saphira load files. There is also a tutorial example program, `tutor/loadable`, with a sample load file.

4.1 Host System Requirements

Saphira and Aria libraries are available for Linux and MS Windows systems. 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.

Saphira now uses the FLTK cross-platform windowing system for GUI objects (www.fltk.org). It has classes for drawing within the Saphira graphics window. Users may also write their own FLTK GUI interfaces.

For MS Windows, the libraries have been compiled with MS Visual C 6 tools. A `DLL` file and an associated `LIB` file are available.

4.2 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` in a UNIX system, for example.

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

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

As of Saphira 8.0, 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`.

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.

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 when creating loadable shared object files.

Figure 4-1 is a graphical view of the standard Saphira client execution process. The main client thread starts up, and invokes the Saphira OS to run the synchronous task loop. After start-up, the OS wakes up every 100 ms and runs every micro-task.

For most robot programming, all operations can be handled in micro-tasks. If a more compute-intensive task must be done concurrently, the user can now run asynchronous 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 and class variables.

4.2.1 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`

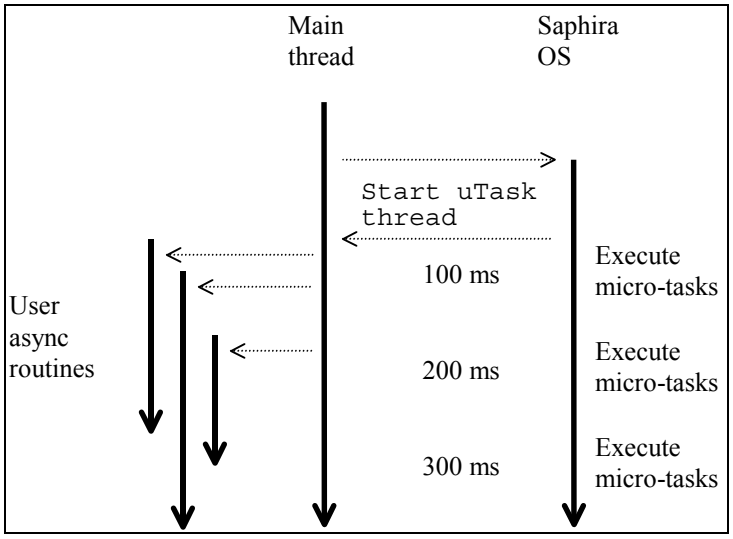

Figure 4-1. Concurrent execution of Saphira OS and user asynchronous tasks.

`testload.so` will make all the symbols in this file available to the debugger, assuming it was compiled with the `-g` option.

4.2.2 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.

5 Saphira API Documentation

The Saphira and Aria class and function API's are now documented using the Doxygen documentation system. HTML versions are available in `Saphira/docs/html` and `Aria/docs/reference/html`.

6 Markov Localization Module

Saphira incorporates an efficient version of Monte-Carlo Markov Localization [Fox et al. 1999] that can keep the robot localized in a global map, based on sonar or laser sensors.

The ML module is packaged as both an application for demonstrating the capabilities of the algorithm, and an API for invoking the ML functionality from a Saphira client.

6.1 Markov Localization Overview

Markov Localization is a process for estimating the state of the robot, in particular, its pose. The basic idea is to divide the motion of the robot into a set of discrete steps (e.g., 1 meter of distance or 20 degrees of turn between steps). At each new step, the position of the robot is estimated from dead reckoning information, and sensor returns compared against an *a priori* map.

The estimation is a two-step process: *prediction* from the dead reckoning information, and *update* from the sensor information. Generally, uncertainty in the robot's pose grows with the prediction step, and is reduced with the update step, where the pose is registered with the map. The process is *Markovian* because it is assumed to be history-less: it doesn't matter how the robot got to a particular place --- all information is given by the robot's current pose and uncertainty.

There are several ways of representing uncertainty in the robot's pose; a good overview is in the document, **Robot Notes: Robot Motion** in the `docs/` directory. For this implementation, we chose the efficient *Monte-Carlo* method, where the uncertain robot pose is represented as a set of sample poses. The sample poses are concentrated around the most likely area for the robot to be.

There are a number of parameters that are important for the ML process. Chief among these are the number of sample points, the *gain* applied when updating with sensor information, and the distance between updates. All of these parameters are set to reasonable values on startup, and can be manipulated by the user program, or via the GUI interface.

6.2 Loading the ML Module

Here are the relevant library files for the ML algorithms:

Name	Description
`lib/loc.so`, `lib\loc.dll`	Core library routines and API interface for Markov Localization with sonars or laser rangefinder.
`lib/flloc.so`, `lib\flloc.dll`	GUI library; adds menu items and dialogs for changing localization parameters.
`colbert/flloc.act`, `lrfloc.act`, `scan.act`	Application initialization files. Use `flloc.act` to initialize ML with sonar sensors. Use `lrfloc.act` to initialize ML with laser rangefinder and the standard `aic.wld` file. Use `scan.act` to initialize ML with laser rangefinder and a grid map built from ScanStudio, and to run the Gradient module in addition.

The simplest way to start is to load `flloc.act`, using the Colbert command "`load flloc`". This will load a sample world file (`aic.wld`), the core ML library and the GUI, and start up the ML processes. Start the simulator with the same world file, connect to it, and the red sample cloud will follow the correct real-world position of the robot. Select `Update Position` from the `Localize` menu, and the Saphira

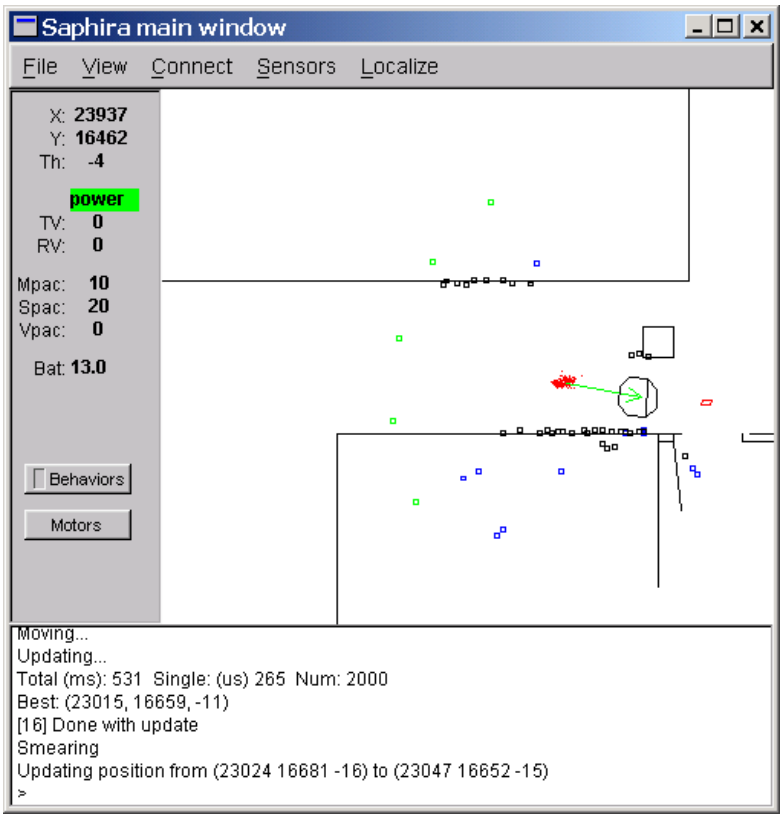

Figure 6-1 Localization module snapshot. The red cloud is the set of sample points for Markov Localization. The green arrow shows the best estimate of robot position at the last update. The sample cloud is updated as the robot moves, typically every 1.0 meters or 20 degrees.

client will stay tracked to this position. Figure 6-1 shows a snapshot of the ML module algorithm during a typical robot run.

Loading the libraries does not actually create the localization objects. To do this, the function `mcSonarInit()` or `mcLrfInit()` must be called (Section 6.5). `mcSonarInit()` is called by `flloc.act`, and `mcLrfInit()` is called by `lrfloc.act`.

6.3 Localization Parameters

ML parameters can be manipulated through a pop-up dialog. Pull down the Localize menu, and select the Parameters item. The window shown in Figure 6-2 will appear. Using the tabs, the number of samples, gain, and frequency of update can be changed. These changes take effect immediately.

Changing the number of samples results in the sample poses all being reset to zero, so the sample cloud will vanish. It can be restored to the robot position using the `Localize->Sample Set->Center on Robot` iterm.

The following table gives the Colbert/C++ functions for setting parameters of the ML process.

Function	Parameter
`mcSetNumSamples(int n)`	Sets the number of samples to *n*. All sample poses are reset to zero, and `mcSetGauss` may be called to re-center the sample set on the robot.
`mcSetMove(int ds, int dth, int dt)`	Sets the distance the robot must move (in mm and degrees) before an update takes place. Also, when the robot stops moving, and update will take place after dt sync cycles to move the cloud to the robot. If dt is 0, no such update takes place.
`mcSetGain(int pct)`	Sets the gain of the sensor information in the update step to the percent `pct`. If pct is 0, no sensor information is used. Reasonable values range from 10 to 50 percent, depending on the environment, the application, and the sensors.
`mcSetGauss(float ds, float dth)`	Centers the sample cloud in a gaussian around the robot, with a deviation of `ds` in distance (mm) and `dth` in angle (degrees).

6.4 Localization Menu

Using the Localize menu, several aspects of the display and performance of the ML module can be manipulated.

- The display of the sample cloud and the best estimate can be turned off and on.
- The display of the underlying map grid can be toggled. The map grid feathers out the map walls, so that the update step has a more continuous nature.
- Update Samples checkbox. Use this checkbox to toggle the state of the ML process. If the box is checked, ML will proceed as the robot moves. Unchecked, ML is halted.
- Update Robot Pos checkbox. Use this checkbox to jump the robot to the position of the best estimate from ML. If unchecked, the sample point estimate and the robot position will diverge.
- The Sample Set item changes the position of the sample set. Center on Robot causes it to assume a gaussian shape around the robot's center. Uniform causes it to spread uniformly over

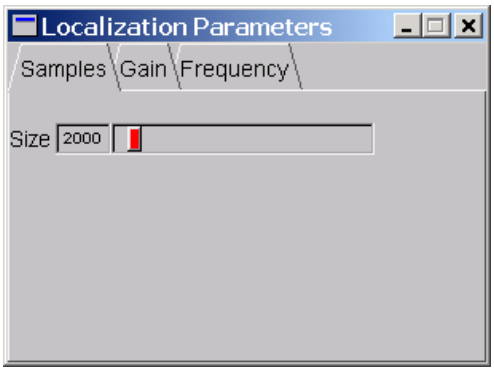

Figure 6-2 Localization parameter dialog window.

the space occupied by the map. This is useful for experiments with *global* localization, where the pose of the robot is initially unknown.

- Update Map. If a new world map is loaded into Saphira, the data structures used by the ML module must be updated; choosing this menu item will do so.

6.5 Initialization Functions

Several functions are available for initializing the ML module. These functions can be called from Colbert or from C++ code. Initialization sets up the ML process object, and performs other necessary processing such as setting up the map data structures.

Function	**Parameter**
`mcSonarInit()`	Initializes the ML module, using sonar readings for the update step. A world file map should already have been loaded, since this function also sets up the ML map structures.
`mcLrfInit()`	Initializes the ML module, using laser range finder readings for the update step. A world file map should already have been loaded, since this function also sets up the ML map structures.
`mcLrfScanInit()`	Initializes the ML module, using laser range finder readings for the update step. Instead of a world file map, this form of the ML algorithm uses a scan map generated by ScanStudio. The scan map must be loaded after this call.
`mcLoadScanMap(char *mfile)`	Loads the scan map file `mfile` into the system. The scan map file must be generated with ScanStudio. `mcLrfScanInit()` should have been previously called.

7 Gradient Path Planning

For efficient movement based on local obstacles and world maps, Saphira has a realtime path planner based on the gradient method [Konolige 2000]. For planning paths and moving in a world map, the gradient method is typically used with Markov Localization to keep the robot registered with a map as it moves.

The Gradient module is packaged as both an application for demonstrating the capabilities of the algorithm, and an API for invoking the Gradient functionality from a Saphira client.

7.1 Gradient Overview

Gradient Path Planning is a process for determining optimal paths for the robot, in real time. These paths can take into account both local obstacles, sensed by sonars and/or laser range-finder devices; and global map information such as the location of walls and other structural obstacles.

At each sync cycle (100 ms), the Gradient module calculates the lowest-cost path from a goal point or set of goal points to the robot. The algorithm starts by considering a local neighborhood connecting the robot and the goal or goals, and then expands its search if no path is found. There is a user-settable limit on the size of the neighborhood considered.

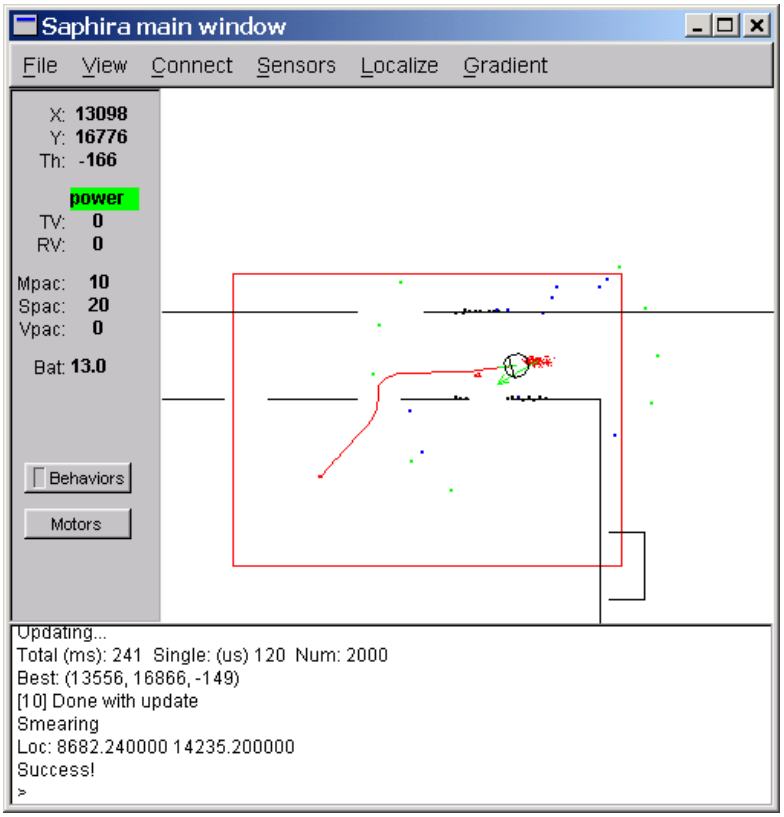

Figure 7-1 Gradient module snapshot. The red cloud is the set of sample points for Markov Localization, which is also loaded. The red line connecting the robot and the goal position is the optimal path, calculated by the Gradient algorithm during the current time step. The red rectangular box is the neighborhood considered by the algorithm. Note that the path goes around the map walls, even where the robot cannot see them.

7.2 Loading the Gradient Module

Here are the relevant library files for the Gradient algorithms:

Name	Description
`lib/grad.so`, `lib\grad.dll`	Core library routines and API interface for the Gradient algortihms with sonars and/or laser rangefinder.
`lib/flgrad.so`, `lib\flgrad.dll`	GUI library; adds menu items and dialogs for changing gradient performance characteristics.
`colbert/flgrad.act`, `scan.act`	Application initialization files. Use `flgrad.act` to initialize Gradient for either sonars or laser range finders (or both). Use `scan.act` to initialize ML with laser rangefinder and a grid map built from ScanStudio, and to run the Gradient module in addition.

Normally Gradient is run in conjunction with localization and a map, although it can be run by itself for local obstacle avoidance.

The simplest way to start is to load `flloc.act`, using the Colbert command "`load flloc`". This will load a sample world file (`aic.wld`), the core ML library and the GUI, and start up the ML processes. Then, load the Gradient library and GUI with the `flgrad.act` file, using the Colbert command "`load flgrad`". Start the simulator with the same world file, connect to it, and the red sample cloud will follow the correct real-world position of the robot. Select `Update Position` from the `Localize` menu, and the Saphira client will stay tracked to this position. Finally, pull down the `Gradient` menu, and check the `Use Artifacts` item. This will make sure Gradient pays attention to the map walls.

At this point you're ready to roll. Goal positions are set by *shift-left-click* in the Saphira graphics window (you may have to just click first in the graphics window to select it). You should see a path such as the one in Figure 7-1, which shows a snapshot of the Gradient algorithm during a typical robot run.

If the robot is connected, and Behaviors are turned on (the Behavior button is lit on the left side of the Saphira main window), then the robot will start moving towards the goal. The Gradient algorithms include programs that calculate an optimal speed along the path, so that the robot slows down near tight curves and speeds up on straightways.

The goal can be changed at any time, by shift-left-click in the graphics window. The path will change immediately.

7.3 Gradient Menu

Using the `Gradient` menu, several aspects of the display and performance of the Gradient module can be manipulated.

- The display of the path and goal can be toggled on and off.
- The display of the cost field (`accel`) and gradient field can be toggled on and off.
- The sensors used in local obstacle avoidance can be toggled, using the check boxes.
- Artifact obstacles (map walls) can be toggled on or off.

There is no parameter dialog window for Gradient yet, although the menu item for it is present.

7.4 Gradient Functions

Several functions are available for initializing and controlling the Gradient module. These functions can be called from Colbert or from C++ code. Initialization sets up the Gradient process object, and performs other necessary processing such as setting up the Gradient behavioral action.

Function	**Parameter**
`gradInit()`	Initializes the Gradient module. Should be called right after loading the Gradient library.
`gradSetMax(int width, int height)`	Sets the maximum size (in mm) of the neighborhood considered by the Gradient module. The neighborhood will expand until it reaches this size, in searching for a valid path.
`gradSetSpeed(int high, int mid)`	Sets the maximum speed for free running (high) and more congested travel (mid) for the Gradient path-following behavior. Speeds are in mm/sec.
`gradSetGoal(float x, float y)`	Sets the Gradient goal to this global point (in mm). The goal can be changed at any time.
`gradSetMap(void *mobj)`	Sets the internal map to a ScanStudio map already loaded into the system. Typically this will be a map object returned by `mcGetObject()`.

8 Parameter Files

This section describes the parameter files used by the Pioneer simulator and Saphira client to describe the physical robot and its characteristics.

8.1 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`.

8.2 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
`RotVelConvFactor`	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.

9 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).

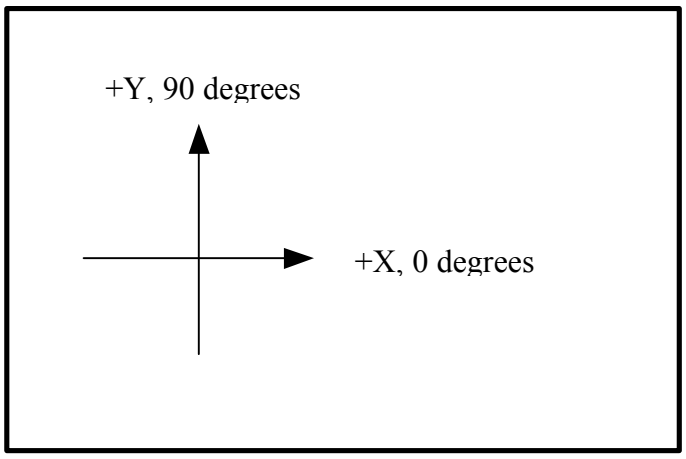

Figure 11-1. Coordinate system for world definition.

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 `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.