ActivMEDIA ROBOTICS

Pioneer 2

MOBILE ROBOTS

with Pioneer 2 Operating System Servers

SAPHIRA MANUAL

PRELIMINARY

Saphira Operations and Programming Manual Version 6.2, August 1999.

Contents

1 Saphira Software & Resources

Saphira is a mobile robotics-client applications and development environment. It is a product of SRI International's Artificial Intelligence Center. Saphira development and maintenance are under the direction of its original author, Dr. Kurt Konolige, who also designs Pioneer Mobile Robots. This *Saphira Software API Manual* provides the general and technical details you will need to program and operate your intelligent mobile robot with Saphira software.

1.1 Saphira Client/Server

Saphira software operates in a multitiered client/server environment. The foundation, of course, is a robot server like the included Pioneer simulator or a real Pioneer Mobile Robot from *Activ*Media ROBOTICS. The robot server carries the basic components of real-world sensing and navigation for intelligent mobile robot activities, including drive motors and wheels with position encoders, range-finding sensors, and so on, as well as the electronics and embedded controllers to manage those resources—a Pioneer 2 DX with its Siemens C166-based microcontroller running the Pioneer 2 Operating System, for example.

The robot server handles the low-level details of sensor and drive management, such as collecting range-finding information from onboard sonars, maintaining individual wheel speeds, positioning, heading, and so on. However, without a client to guide it, the mobile robot server is taskless; it is the machine in machine intelligence.

The Saphira multitiered client provides the intelligence for intelligent mobile robotics; it performs the work for taskfull operation of the robot server.

1.1.1 Client Components

In brief (the remainder of this manual is devoted to the details, of course), Saphira's lowest level—its interface with the robot—provides a coherent method and protocols for communication with and control of a robot server, by retrieving real-time, real-world operating data from the robot and sending back commands to control the robots activities.

Saphira's intermediate layers support higher-level functions for navigation control and sensor interpretation, and for the integration of robot accessories (plugins).

At its upper levels, Saphira provides state-of-the-art fuzzy-logic-based control behaviors and reactive planning systems, features-recognition systems, and a map-based navigation and registration system (future development to include localization).

Saphira also provides a full-featured Graphical-User Interface (GUI) and command-level interface (Colbert Executive; see below) for interactive monitoring and manual control of both the Saphira client and its robot server and accessories.

1.1.2 Behavior Compiler and Executive

Saphira uses *fuzzy control rules* for implementing and integrating rudimentary robot control programs, known as *behaviors.* Saphira comes with several pre-defined behaviors, including obstacle avoidance. And Saphira provides the tools for you to define and develop your own behaviors, including a *behavior compiler* that translates a simple fuzzy-control-rule syntax into C-language-based code that you include in your Saphira client.

1.1.3 Colbert Executive

Saphira version 6 added support for a simple, C-like language for creating robot-control programs. With Colbert, users 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 Saphira's Colbert executive, which supports concurrent processing of activities.

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

Please consult the *Colbert Programming Manual* for complete Colbert programming details.

1.1.4 Saphira Plugins

The Colbert Executive also provides a way to integrate and dynamically manage Saphira extensions, particularly device managers such as for the Pioneer 2 Gripper or Fast-Track Vision System, through shared-object libraries. Once composed and compiled, Saphira plugins may be shared among many clients, loaded, operated, and unloaded programmatically by the client or Colbert activity, or manually by the user.

Look into the `${SAPHIRA}/devices` directory for Saphira plugin `.dll` (Win32) or `.so` (UNIX/Linux) dynamically loadable, shared-object examples.

1.2 Saphira Development

Saphira comes as complete set of ANSI C-language-based software libraries and utilities which you write to and link with your C- or C++-based programs to create your own Saphira clients and shared-object libraries. Saphira programs can be written in and run under many different computing environments, including Microsoft Windows® 32-bit operating systems (WindowsNT®, Windows95®, and Windows98®), and with most UNIX® and like systems (SunOS®, Solaris®, IRIX®, OSF/1, FreeBSD, and RedHat Linux®).

Of course, details of Saphira's Applications Programming Interface (API) are in the following chapters of this manual. And we provide some guidance for preparing Saphira clients on the individual applications development platforms; specifically Microsoft's Visual C++®, Linux's GNU tools, and for common UNIX®-based C-compilers and linkers. For operation of the C- development platform itself, however, please consult its and its accessory documentation.

1.3 Robot Simulator

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

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

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

1.4 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[1]
- Radio modems or Ethernet radio bridge (optional)
- Computer: Macintosh[2]; PC with Microsoft Windows 95/98 or NT, FreeBSD, or Linux operating system; or UNIX workstation
- Open communication port (TCP/IP or serial)
- Four to five megabytes of hard-disk storage
- PKUNZIP (PCs), GUNZIP (PCs and UNIX), StuffIt Lite, or compatible archive-decompression software

Optional:

- C-program source-file editor and compiler. *Note: The current Windows98/NT version of Saphira supports only Microsoft's Visual C/C++ development environment, not Borland's Turbo-C/C++ products*. Necessary for compiling new subroutines in standard C.
- Motif GUI and libraries for FreeBSD/Linux/UNIX. Necessary only to compile new clients; with Colbert, users may instead operate with a pre-compiled Saphira client.

1.5 Saphira Resources

Saphira is available in many ways, and comes with a plethora of development supports.

1.5.1 Where to Get Saphira

Saphira demonstration packages are available for electronic download and free use currently from two Internet-based sources. Use your favorite browser and contact:

http://www.ai.sri.com/~konolige/saphira

or

http://robots.activmedia.com

The unlicensed Saphira demonstration package is the complete Saphira intelligent mobile robotics client-development environment. You just won't be able to connect with a real robot.

Licensed versions of Saphira are distributed through *Activ*Media ROBOTICS. Pioneer Mobile Robot customers automatically get a full Saphira license, and the Win32 version on a 3.5-inch diskette accompanies each robot. Pioneer customers and others by special license also may download any and all of the variety of Saphira platform versions and accessories from *Activ*Media ROBOTICS' support website:

`http://robots.activmedia.com`

To gain access to the license-restricted Saphira areas, enter the username and password that are written on the ***`Registration & Account Sheet`*** accompanying your Saphira distribution and this manual.

The latest information for installing and running Saphira can be found in the `readme` file in each distribution. Please examine this file carefully before and during installation. An `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.

---

[1] This may be the Pioneer Simulator or Pioneer Multi-agent Simulator.

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

1.5.2 Saphira Newsgroup

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

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

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

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

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

1.5.3 Support

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

saphira-support@activmedia.com

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

1.5.4 SRI Saphira Web Pages

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

- tutorials and other documentation on various parts of Saphira
- class projects from Stanford CS327B, *Real-World Autonomous Systems*

information about SRI robots and projects that use Saphira, including the integration of Saphira with SRI's Open Agent Architecture

- links to other sites using Pioneer robots and Saphira

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

1.6 Acknowledgments

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

2 Installation and Quick Start

The typical Saphira client has a GUI through which you may connect with and interactively control a robot client, including the simulators. This chapter describes the installation and quick startup of the Saphira client. Subsequent chapters describe the many feautres of the GUI client and how to develop your own Saphira clients.

2.1 Installing the Software

The Saphira distribution software, including the `saphira` demonstration client, Colbert Executive, the Pioneer simulator, and accompanying C libraries, headers, and demonstration sources, come as a compressed archive of directories and files either stored on a 3.5-inch diskette, or as download from the *Activ*Media ROBOTICS and SRI International websites.

Each Saphira archive is configured and compiled for a particular operating system, such as for Windows95/98/NT (Win32), a particular UNIX, or Linux. Choose the version that matches your client computer system. See *Resources* earlier in the previous chapter for details.

2.1.1 Locating the Saphira Distribution

When extracted, each Saphira distribution creates a single top-level directory named for its particular version—in this case, for instance, `ver62`. Beneath that main directory are several subdirectories containing everything you need to program and operate Saphira clients.

We recommend that you create a Saphira master directory to keep the various Saphira distributions in some publicly accessible partition, such as `C:\Saphira` on a Win32 drive or `/usr/local/Saphira` on a UNIX/Linux system. Be sure to give users the appropriate permissions for access.

2.1.2 Extracting the Saphira Distribution

Copy the Saphira distribution to the master directory, then extract it. The Win32 versions are self-extracting WinZip archives; the UNIX and Linux versions come gzip'd and tar'd. To decompress the Win32 software, simply double-click its icon or otherwise execute the self-extracting (.EXE) program. For UNIX/Linux systems, `gunzip` and then `tar -xf` or simply `tar -zxf` the Saphira distribution. (Consult the man pages for details on these and other UNIX/Linux commands.)

For all systems, a hierarchy of folders and files get put inside the version-related Saphira top-level directory, possibly replacing earlier file versions. For example, the distribution subdirectories for the Win32 Saphira version 6.2 look like the ones (among others) shown in the Figure 1-1.

```
ver62\
  readme                Explanation text file
  update                Comparison of versions
  clients\              Client application source files
     saphira.c          Saphira demonstration source file
     saphira.mak        MSVC++ makefile
  bin\
     saphira.exe        Saphira/Colbert runtime application
     pioneer.exe        Simulator
     bgram.exe          Behavior grammar compiler
     sf.dll             Saphira executable libraries
     msvcrt40.dll       Required MS Windows DLL
  colbert\              Colbert activities and samples
     init.act           Executive startup activity
  devices\              Saphira support for robot accessories
  handler\              Native libraries and resources
     basic\
        behavior.beh    Behavior examples
     include\           Development header files
```

```
      obj\   Library files
         sf.lib          Saphira link library
   \maps\    Saphira maps
   \params\  Robot description files
      P2DX.p Pioneer 2 DX robot, for example
   \worlds\  Simulator world files
```

Figure 1-1. Distribution directory for Win32 Saphira version 6.2.

2.1.3 Required Saphira Environment Variables

Saphira requires that you set at least one Win32 and two UNIX/Linux system environment variables. Other Saphira environment variables which may effect operations are optional; we describe them in context within later chapters.

IMPORTANT!

You must set a SAPHIRA environment variable before you can successfully operate any Saphira client.

For Windows95/98 systems, edit the `autoexec.bat` file found in the boot directory (usually `C:\`) with any simple text processor, such as `notepad` or `WordPad`. Assuming the top-level Saphira directory is `C:\Saphira\ver62`, add the following line to the file:

```
SET SAPHIRA=C:\Saphira\ver62
then reboot.
```

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

With UNIX/Linux systems, use one of the following methods to set the `SAPHIRA` environment variable, preferably in the user's `.cshrc` or even in the global `/etc/profile` script parameter file:

```
export SAPHIRA=/usr/local/Saphira/ver62  (bash shell)
setenv SAPHIRA /usr/local/Saphira/ver62  (csh shell)
```

2.1.4 Locating the Saphira Libraries

The Saphira library is dynamically loadable and sharable on all UNIX,Linux, and Win32 systems. This means that one or more Saphira applications each can link into the library at runtime, rather than each have a copy attached at compile time. Therefore, clients take up less space and are quicker to compile. They simply need to find the Saphira library at runtime.

With Win32 systems, we locate the `sf.dll` Saphira dynamically loadable library in the same directory where you find the Saphira client executables (`ver62\bin`). Consequently, the system automatically associates the Saphira client with the library. If you plan to relocate your Saphira clients, you might either copy `sf.dll` into the same directory as the client executable, or locate it in the system folder of your boot drive (normally `C:\`) for general access: `Windows/System` for Windows95/98 or `Winnt\System32` with Windows NT.

With UNIX/Linux systems, the Saphira shared library is in `ver62/handler/obj`. You can make the library accessible to applications in two ways. We recommend leaving the Saphira shared library in this directory and putting the directory name into the load-library list using the `shell` command:

```
export LD_LIBRARY_PATH=$SAPHIRA/handler/obj (bash)
     or
SETENV LD_LIBRARY_PATH=${SAPHIRA}/handler/obj  (csh)
```

Alternatively, copy the Saphira shared object (.so) library file from the `${SAPHIRA}/handler/obj` directory into the standard library directory, `/usr/lib`.

2.2 Saphira Quick Start

Have a real robot server or the Simulator readied for a Saphira connection. For example, execute the `${SAPHIRA}/bin/pioneer(.exe)` robot Simulator on the same computer, or simply connect (tether or radio modems) the "host" port on your Pioneer 2 Mobile Robot to a serial port on your basestation computer running the Saphira demonstration program. (See the *Pioneer 2 Operations Manual* for details.)

2.2.1 Execute the Client

Start the Saphira client demonstration program by navigating to the ${SAPHIRA}/`bin` directory and executing the program named `saphira(.exe)`. For instance, use the mouse to double-click the `saphira.exe` icon inside the `C:\Saphira\bin\` folder on your Windows98 desktop.

With UNIX/Linux, you must be running the X-Window system to execute the Saphira demonstration client software. The `${SAPHIRA}/bin/saphira` program is a shell script which automatically sets the Saphira environment variables for you and then launches the `saphira` executable. If the script fails, edit it for the correct environment variable assignments and the proper `saphira` executable:

The UNIX/Linux Saphira executable comes in two forms: statically linked or not with the Motif GUI library. If you do not have the Motif GUI shared library (`libXm.so`) installed, you need to use the `ssaphira` program. Otherwise, execute the smaller `xsaphira` program.

When successfully launched, the Saphira client window appears with a graphical display of the robot internals, a textual information window, and a command-line interaction window. Type `help` in the interaction window for a list of command classes that you can query for further information.

2.2.2 Connect with a Robot Server

Saphira establishes contact and control with a Pioneer robot server through a serial port, either `COM1` through `COM4` on a Win32 system, `/dev/ttys0` through `/dev/ttys3` on a UNIX system, or `/dev/cua0` through `/dev/cua3` under Linux. If you're accessing the Pioneer simulator on the same machine, `connect local`, which opens a local port to the simulator and starts things up.

Find and choose one of these connection options from the `Connect` menu in the Saphira main 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 becomes distinctively alive with robot readings.

We detail Saphira client operation in the next chapter. For now, we leave it to you to find the manual drive keys and take your robot for a joyride. (Hints: keyboard arrows move and the spacebar stops the drive motors; be sure to enable the motors on the real robot.)

Also, the Saphira demo automatically loads the demonstration Colbert program ${SAPHIRA}/`colbert/demo.act`; it and has more activities you can try out by starting them from the `Function/Activities` menu from the main Saphira window.

2.2.3 Disconnecting and Shutdown

The graceful way to shut down a Saphira client/robot server connection is to choose `Disconnect` from the main window pulldown `Connect` menu. Or you can also type the command `sfRobotDisconnect` in the Colbert interaction window. Either way, the client stays active and ready to establish another connection to same or another robot.

Close the Saphira main window or choose `Exit` from the `Connect` menu to shut down the Saphira client. A connected server automatically gets disconnected when you shut down the client.

3 Operating the Saphira GUI Client

Saphira comes in two flavors: one with a Graphical User Interface (GUI) and one without. The windowless client is for autonomous robot operation. On the other hand, the GUI Saphira client gives you visual and textual representation of both your Saphira client's and your robot server's operations and states, and gives you the ability to manually interact with each to effect changes in their activities.

This Chapter describes operation of the Saphira GUI client and its many features.

4 Guide to the Saphira API

This chapter details the current library of functions for development of a Saphira client. Additional information about prototypes, structures, and variables can be found in the various header files in the `handler/include/` directory of your Saphira distribution. Also study the sample source files in the `apps/` directory of working Saphira applications.

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

4.1 Saphira OS Functions

The Saphira OS functions perform initialization and setup of the Saphira client. One class of functions provides for automatic actions when Saphira connects to or disconnects from a robot. For example, you could place the robot at a certain global position within a map whenever Saphira connects.

Another class of Saphira OS functions let you initialized and run the Saphira client in parallel with threads from other routines, thereby creating more complex Saphira clients than the default one and useful for linking the Saphira libraries with other applications. These functions provide for the Saphira client thread to be, for example, vision processing or planning routines written by the user.

A final class of Saphira OS functions invokes standard microtasks for communication, perceptual processing, and robot action.

4.1.1 Startup

```
void sfStartup (int async)                                    [UNIX]
void sfStartup (HANDLE hInst, int cmdShow, int async) [MS Windows]
void sfPause(int ms)
char *sfVersion
```

Use the `sfStartup` function exactly once to initialize and execute a Saphira client thread. Use `sfPause` to pause the client. The `sfVersion` string reports the current Saphira version number.

Description

The `sfStartup` function may be called at any time by your program, but it should be called only once (there is no explicit Saphira OS shutdown). The first form of `sfStartup` is for UNIX/Linux systems. The form is for Win32 environments, and include the Windows OS required application instance handle (`hInst`) and the visibility parameters (`cmdShow`), which you typically just pass through from the WinMain parameters.

When invoked, `sfStartup` initializes and executes the Saphira OS. With UNIX/Linux versions, if the client has been linked with the window libraries, Saphira opens its main GUI interface window (see also Chapter 3, “Operating the Saphira GUI”). With Win32 systems, the Saphira OS always includes a GUI window.

Set the `async` argument to 0 to give Saphira sole control of the client. In this mode, associated functions must be integrated with the Saphira multitasking OS.

Set `async` to 1 to have Saphira run as one of other threads in your client program. In this mode, sfStartUp initializes the Saphira OS, but control returns immediately to the calling program.

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

The Saphira OS startup functions are not available in Colbert, since Colbert is a feature of, and thereby requires a running Saphira OS. Moreover, there is a native method (“wait”) for pausing in Colbert activities.

The Saphira variable `sfVersion` is a short string containing the current version number and revision letter of Saphira; “6.2a”, for example.

The provided example is the simplest Saphira client. It starts the Saphira OS, complete with GUI, and shuts down when you quit the Saphira main window.

Example

```
#ifdef IS_UNIX
void main(int argc, char **argv)
#endif
#ifdef MS_WINDOWS
  int PASCAL
  WinMain (HANDLE hInst, HANDLE hPrevInstance, LPSTR lpszCmdLine, int
nCmdShow)
#endif
{
  /* … Initialize and prepare client here… */
#ifdef IS_UNIX
  sfStartup(0);           /* Give Saphira full control */
#endif
#ifdef MS_WINDOWS
  sfStartup(hInst, nCmdShow, 0);
  return 0;               /* Returns here after Saphira OS shutdown */
#endif
}
```

4.1.2 Handlers and States

```
void sfAddStartupHandler(void (*fn)(),int which)
void sfAddConnectHandler(void (*fn)(),int which)
void sfAddDisconnectHandler(void (*fn)(),int which)
void sfAddExitHandler(void (*fn)(),int which)
void sfOnStartupFn (void (*fn)())
void sfOnConnectFn (void (*fn)())
void sfOnDisconnectFn (void (*fn)())
void sfOnExitFn (void (*fn)())
int  sfIsConnected
int  sfIsExited
```

#define sfFirst 1

#define sfLast 0

#define sfRemove ?????

Several Saphira functions register system callbacks to code in your client and Colbert/plugin programs that get executed when key OS events occur: When the Saphira OS first starts up, when it connects with a robot server, when it disconnects from a robot server, and when it exits. Associated variables keep track of these various OS states.

Handler Installation and Invocation

Handlers are functions that Saphira invokes when a particular event takes place. Currently, there are four Saphira OS events that invoke handlers: `Startup`, `Connect` (to robot server), `Disconnect` (from robot server), and `Exit`. Saphira maintains a list of up to 10 separate handlers for each of these OS event. Each gets invoked in order from `sfFirst` to `sfLast` in the list.

Register your event handlers with the related `sfAddHandler` Saphira function. The `which` parameter value adds the referenced handler to the top (`sfFirst`) or bottom (`sfLast`) of the list, or removes (`sfRemove`) a previously added handler from the list so it is no longer invoked by an event.

The functions `sfOnStartupFn`, `sfOnConnectFn`, `sfOnDisconnectFn`, and `sfOnExitFn` are convenience functions provided for compatibility with previous releases of Saphira. If you include a function-pointer (non-`NULL`) argument, it gets added to the top of the event handler list, as if you had used `sfAddHandler` with `sfFirst` as the `which` argument value. With a `NULL` argument, each of the convenience functions removes the last handler from their list. (Yes, this is a bit confusing since the NULL argument *adds* a handler to the end of the list with the `sfAddHandler` command.)

None of the OS event-handlers are required. Your `StartUp` handlers should include any relevant initialization code, such as menu or directory settings. The `Connect` handlers should start micro-tasks, behaviors, and other Saphira control routines. Your `Disconnect` handler can be used to clean up after the Saphira client disconnects from a robot server. And use the `Exit` callback to do some system housekeeping when the Saphira OS exits.

All of the Saphira OS handlers may be included in your Saphira client code, but only `Connect` and `Disconnect` handlers may appear in Colbert and related plugins. This is because Colbert requires a running Saphira OS.

4.1.3 Saphira OS State Variables

The variables `sfIsExited` and `sfIsConnected` reflect the states of the Saphira OS and its connection with a robot server, respectively. The user should not change their values.

The `sfIsExited` variable is particularly useful with an asynchronous Saphira client, which retains execution control after starting the Saphira OS (`sfStartup(1);`). This way, code outside of the Saphira OS can check the `sfIsExited` flag and act if the user has requested to exit Saphira, such as with the Saphira main window `Connect/Exit` menu option.

Examples

The Saphira client consists of what happens before and after invoking the Saphira OS, adjusted for the different native operating environments. There are several sample clients in the {SAPHIRA}`/apps` directory, including the GUI-based, synchronous demonstration Saphira client, `saphira.c,` described in the previous chapter, *Quick Start*, and an asynchronous GUI Saphira client, `async.c`.

```
void main(int argc, char **argv)
{  /* set up user button and key processing /
  sfAddButtonHandler(myButtonFn, sfFIRST);
  sfAddKeyHandler(myKeyFn, sfFIRST);
  sfAddConnectHandler(myConnectFn,sfFIRST);
  sfAddStartupHandler(myStartupFn,sfFIRST);
  / start up, don't return */
  printf("starting...\n");
  sfStartup(0);
}
```

Here a few handler callbacks are posted for various Saphira client actions, and then the Saphira OS initialization is invoked. In this case, since async is 0, the Saphira OS thread is started, and the main program waits until it finishes execution before going on from the sfStartup function.

A more complicated invocation of the Saphira libraries is in handler/src/apps/async.c. Here, the Saphira OS is invoked, and the user program continues to execute a sequence of commands to connect to and move the robot. These commands are executed asynchronously with the Saphira OS, which is handling all the basic communication with the robot necessary to make the user commands work. Here is the code from async.c:

```
void main(int argc, char **argv)
{
  int i = 0;
  sfStartup(1);     /* start up Saphira window, and keep going /
  sfMessage("Connect to robot to start this demo");
  while (!sfIsConnected) sfPause(100); / wait until connected /
  sfSetDisplayState(sfGLOBAL, TRUE);   / use the global view /
  sfMessage("Rotate left");
  sfSetRVelocity(100);           / in mm/sec on each wheel... */
  sfPause(4000);
  sfSetRVelocity(0);
  sfPause(4000);

  for (i=0; i<280; i+=60)
    {
      sfSMessage("Turn %d degrees", i);
      sfSetDHeading(i);          /* turn i degrees cc /
      while (!sfDoneHeading(10)) sfPause(100);
                                 / wait till we're within 10 degrees /
      sfSetDHeading(-i);  / turn i degrees c /
      while (!sfDoneHeading(10)) sfPause(100);
                                 / wait till we're within 10 degrees */
    }

  sfMessage("Move forward and turn");
  sfSetVelocity(300);            /* move forward at 300 mm/sec /
  for (i=0; i<10; i++)
    {
      sfSMessage("X: %d  Y: %d", (int)(sfRobot.ax), (int)(sfRobot.ay));
      sfPause(1000);      / DON'T USE SLEEP!!!! /
      sfSetDHeading(10);
    }
  sfSetVelocity(0);       / stop /
  sfPause(4000);
  sfDisconnectFromRobot();/ we're gone... */
}
```

SfStartThread, sfSuspendThread, sfResumeThread, sfDeleteThread

sfSuspendMT, sfResumeMT, sfSetPriority

```
int  sfStartThread(void *fn, void *arg)
int  sfSuspendThread(int id)
int  sfResumeThread(int id)
```

```
void sfDeleteThread(int id)
void sfSuspendMT(void)
void sfResumeMT(void)
void sfSetPriority(int pri)
```

These functions provide an interface to threads that run in parallel with the Saphira OS. Threads are a handy way to implement functions that take a long time to complete, and so cannot be written as Saphira microtasks. For example, a planner might be invoked using a thread, allowing the Saphira OS to continue while it computes its result.

Threads as Asynchronous Tasks

Multiple user threads can run in parallel with the Saphira OS, sharing its address space and having access to all of the standard Saphira functions and variables. The Saphira OS itself runs as a thread, usually at a higher priority than the asynchronous user threads, so that microtasks can execute in their standard 100 ms cycle time.

Thread services are provided by the underlying system OS, either UNIX or MS Windows. There are differences in thread implementations, but for the most part the Saphira functions abstract away from them to provide a simple, common interface.

Description

A thread function is started with `sfStartThread`. The thread function should be a function of one argument, a pointer. The `arg` parameter of `sfStartThread` is passed to the thread function as its argument when it starts up.

Threads are identified by a unique nonnegative integer, returned by `sfStartThread`. If for some reason the thread can't be started, e.g., if there aren't enough system resources, then `sfStartThread` returns –1.

A thread terminates when the thread function exits, either by calling `return` or by falling through the last statement. Alternatively, a thread can be terminated by another thread or the Saphira OS by calling `sfDeleteThread` using its thread id. If the thread has already terminated or doesn't exist, then no action is taken.

A thread can be paused from within the thread function by using the `sfPause` function. This function causes the thread to yield all processor cycles for a specified period of time.

Threads can be paused and resumed from outside the thread by calling the `sfSuspendThread` and `sfResumeThread` functions with the thread id. These two functions are only available under MS Windows; the POSIX specification of threads doesn't include this feature.

Because thread functions execute asynchronously with the Saphira OS, there can be a problem in simultaneous invocation of Saphira functions. For example, a thread executing the `sfMessage` function may be interrupted by the Saphira OS, which then executes its own `sfMessage` functions. The result is a scrambling of the output in the Colbert text window. To prevent simultaneous access, the functions sfSuspendMT and sfResumeMT provide a locking function. When `sfSuspendMT` is called by a user thread, it waits until the OS cycle is complete before continuing. Until the `sfResumeMT` function is called, the Saphira OS is prevented from executing. Obviously, user threads should execute very quickly between `sfSuspendMT` and `sfResumeMT` calls, so as not to lock out the Saphira OS excessively. Generally, calls to functions that access common Saphira data structures are placed between the locking functions.

The locking functions can also be used for synchronization between user threads. At any given time, only one thread can be executing between `sfSuspendMT` and `sfResumeMT` calls. The locking functions are implemented as mutex locks.

Examples

A simple example of a user thread and the locking functions is in `demos/tlock`.

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

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

If you call this function before connecting to the robot (in the start-up callback), it will set the default state for the display function. Thereafter, the preset display values are *sticky*—Saphira automatically resets them to the preset values, perhaps different from the defaults given in Table 4-1), whenever a new connection is made with the robot.

Table 4-1. Optional states for various Saphira display functions.

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

Default state values are in bold typeface.

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

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

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

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

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

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

It is called by Saphira whenever the user presses a key when the main Saphira window is active. The argument `ch` is the character representing the key that was pressed and is operating-system-dependent. Return 0 if you don't handle the keypress; return 1 if you do, particularly to override any of Saphira's built-in key processing routines (see Table 4-1).

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

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

It is called by Saphira whenever the user clicks the mouse when the main Saphira window is active. The *x* and *y* arguments are the screen position of the cursor; *b* is the mouse button, with the values `sfButtonLeft`, `sfButtonRight`, and `sfButtonMiddle`. The shift mask argument `m` is an integer that has bits set indicating which modifier keys were pressed. Return 0 if you don't handle the mouse click; return 1 if you do, to override any of Saphira's built-in mouse processing routines.

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

Not available in Colbert.

4.2 Predefined Saphira Micro-Tasks

We've provided a variety of predefined Saphira micro-tasks for control of the robot. You may initiate these micro-task sets using the API functions described here, or invoke them individually using the `sfInitProcess` API call (see Section 4.5)

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

`void sfInitBasicProcs(void)`

.Starts up a set of basic communication, display, motor, and sensor control processes. Among other activities, these processes implement the client state reflector. The processes invoked are shown in Table 8-2.

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

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

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

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

`void sfInitControlProcs(void)`

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

Table 8-3.

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

`void sfInitInterpretationProcs (void)`

Starts up processes for interpretation of sonar results.

Table 8-4.

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

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

`void sfInitRegistrationProcs (void)`

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

Table 8-5.

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

`void sfRunEvaluator (void)`

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

4.3 State Reflection

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

The state reflector has three important data structures.

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

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

The control structures command robot motions.

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

struct robot sfRobot

The variable `sfRobot` holds basic information reflected from the robot server. Table 8-6, below, shows the values of the various fields in this structure; the definition is in `handler/include/struct.h`.

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

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

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

Table 4-6. Definition of the `sfRobot` structure.

`help robot` for a list of fields.

`sfRobot` field	Units	Description
`x, y, th`	mm, mm, degrees	Robot's location in robot coordinates; always (0, 0, 0)
`ax, ay, ath`	mm, mm, degrees	Robot's global location
`tv, mtv`	mm/sec	Current and max velocity
`rv, mrv`	deg/sec	Current and max rotational velocity
`leftv, rightv`	mm/sec	Left and right wheel velocities
`status`	int STATUS_STOPPED STATUS_MOVING STATUS_NOT_CONNECTED STATUS_NO_HIGH_POWER	Robot status: Robot stopped Robot moving Client not connected Robot motors stalled
`battery`	1/10 volt	Battery power
`bumpers`	int	Bumper state
`ptu`	usecs	Pan/tilt unit (servo) heading
`diginput`	int	Digital input state
`digoutput`	int	Digital output state
`analog`	0-255 [0V-5V]	Analog input voltage

`motor_packet_count`
`sonar_packet_count`
`vision_packet_count`
	counts per second	Packet communication information

4.3.1 Motor Stall Function

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

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

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

4.3.2 Sonar buckets

The current range reading of sonar sensors is held in an `sdata` structure, defined below. The structures for all the sonars are in an array called `sbucket`, e.g., `sbucket[2]` is the `sdata` structure for sonar number 2. Sonars start at number 0. This variable is not defined in Colbert, which doesn't have arrays; instead use the convenience function `sfSonarBucket`.

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

A value of 5000 for the sonar range indicates that no echo was received after the sonar fired and waited for a return. Several convenience functions for accessing current sonar readings are described below.

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

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

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

Listing 8-1.

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

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

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

4.4 Direct Motion Control

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

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

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

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

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

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

The direct motion functions require the state reflector to be operational; that is, the function `sfInitBasicProcs` must be called. This is done automatically by `sfStartup`, so the user need not call it explicitly.

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

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

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

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

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

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

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

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

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

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

float sfTargetVel(void)
float sfTargetHead(void)

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

4.5 Saphira Multitasking

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

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

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

4.5.1 Micro-task Definition

Simple micro-tasks are functions with no arguments together with state information. Micro-tasks access their state through a global integer variable, `process_state`. Processes are initiated by an API call, `sfInitProcess`, which places the function onto the process stack. After they are initialized, Saphira will call them with an initial state of `sfINIT`. The micro-task can change its state by setting the value of `process_state`. User-defined state values are integers greater than 10; values less than 10 are reserved for special states (see Table 8-7).

Table 4.7. Saphira multiprocessing reserved process state values.

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

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

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

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

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

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

Listing 8-2 provides an example of a typical interpretation micro-task function. It starts by setting up housekeeping variables, then proceeds to alternate door recognition with display of its results every second or so.

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

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

4.5.2 State Inquiries

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

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

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

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

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

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

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

4.5.3 Micro-Task Manipulation

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

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

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

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

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

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

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

4.5.4 Invoking Behaviors

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

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

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

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

This function is equivalent to the following:

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

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

4.5.5 Activity Schema Instantiation

An activity schema can be instantiated from another Colbert activity or the user interaction area, with the `start` command (see Section **Error! Reference source not found.**). Alternatively, activities can be started from C code with the `sfStartActivity` function.

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

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

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

This function is equivalent to this one:

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

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

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

4.6 Local Perceptual Space

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

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

4.6.1 Sonar buffers

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

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

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

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

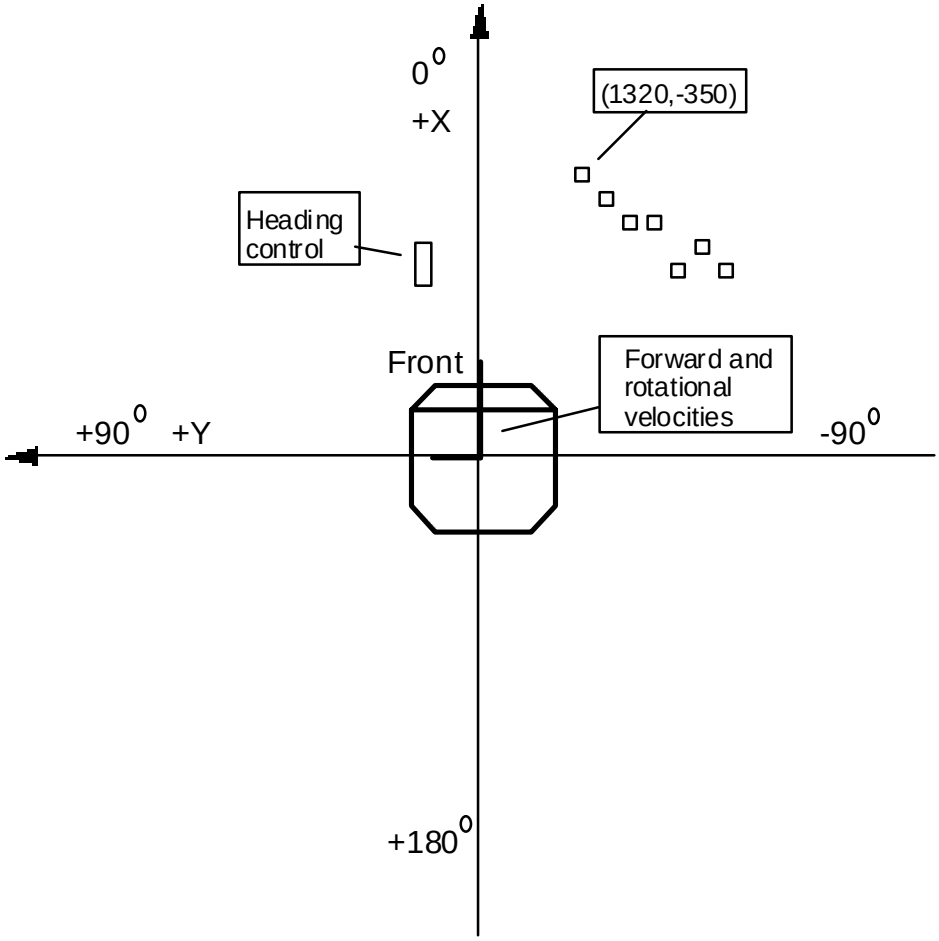

Figure 4-1. Saphira's LPS coordinate system.

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

The buffers differ slightly in how they accumulate sonar readings and therefore serve different purposes. They are all circular buffers; that is, a new reading replaces the oldest one. The front buffer, `sraw_buf`,

accumulates one reading each time a sonar is fired, regardless of whether it sees anything. If nothing is found, the `valid` flag at that buffer position is set to 0; otherwise, it is set to 1, and the `xbuf` and `ybuf` slots are set to the position of the sonar reading, in the robot's local coordinate system. This strategy guarantees that the front buffer can be cleared out after nothing has been in the robot's way for a short time. For example, if the robot is getting 20 front sonar readings a second, and the front buffer is 30 elements long, it will be completely clear in 1.5 seconds if nothing is in front of the robot.

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

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

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

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

Listing 8-3.

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

These buffers are not currently available in Colbert. The first two functions, when given an argument greater than zero, set the front and side buffer limits to that argument, respectively. If given an argument of 0, they clear their buffers, that is, set the `valid` flags to 0. These buffer limits can also be set from the parameter file; they are initialized for a particular robot on connection.

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

4.6.2 Occupancy functions

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

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

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

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

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

For example, in the case of an LPS shown in Figure 4-2, `sfOccBox(sfSIDES,1000,600,900,800,1)` returns 300; `sfOccBox(sfFRONT, 1000,-600,900,600,0)` returns 600.

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

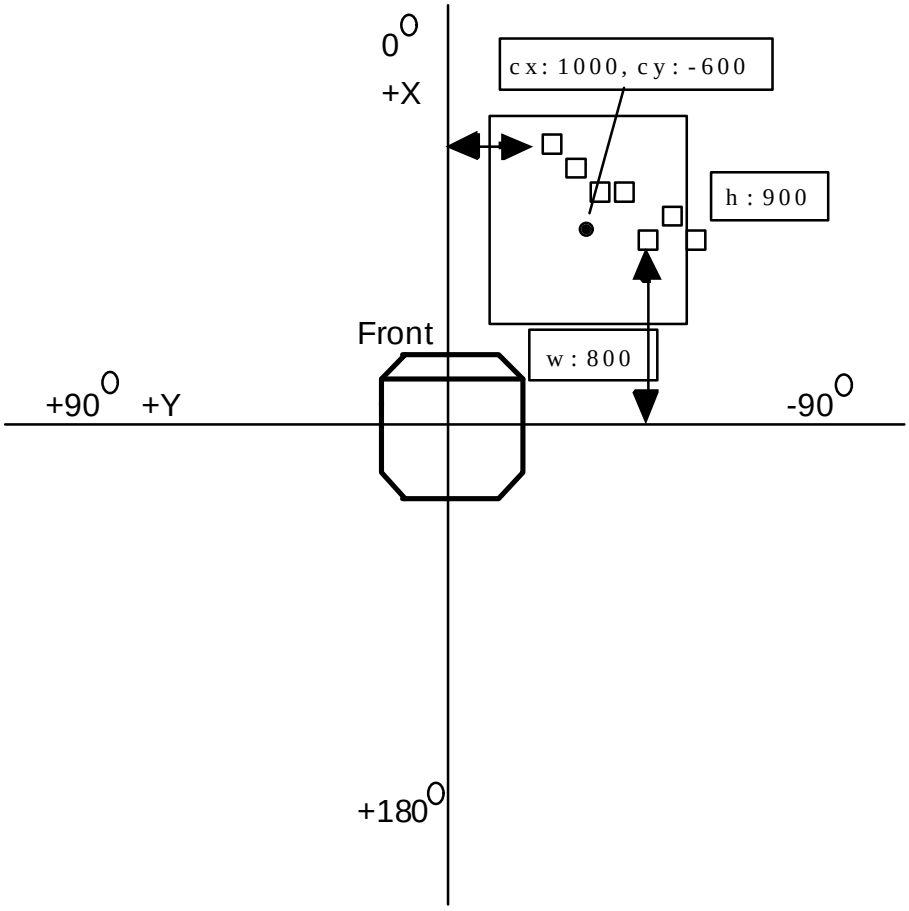

Figure 4-2. Sensitivity rectangle for the `sfOccBox` functions.

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

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

Figure 4-3 shows the relevant areas for `sfOccPlane(sfFRONT,sfFRONT,600,400,1200)`. The first parameter indicates positive *x* direction for the placement of the rectangle. The second parameter indicates the source of the sonar information: the front sonar buffer (`sfFRONT`), the side sonar buffer (`sfSIDES`), or both (`sfALL`).

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

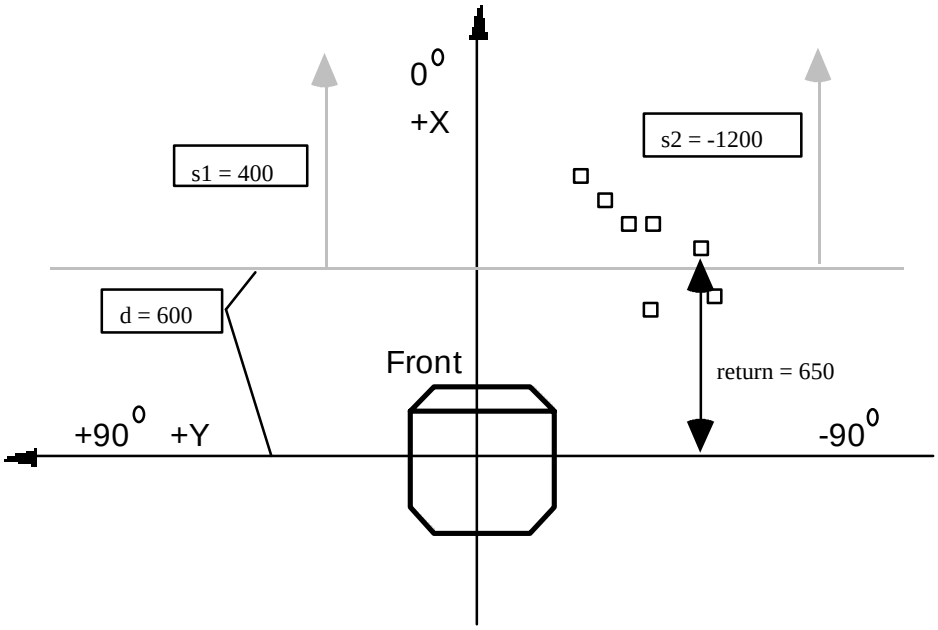

Figure 4-3 Sensitivity rectangle for `sfOccPlane` functions.

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

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

4.7 Artifacts

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

Artifacts, like the robot, exist in both the LPS and the global map space. Their robot-relative coordinates in the LPS (`x, y, th`) can be used to guide the robot locally; e.g.., to face towards a goal point. Their

global coordinates (`ax, ay, ath`) represent position and orientation in the global space. As the robot moves, Saphira continuously updates the LPS coordinates of all artifacts, to keep them in their relative positions with respect to the robot. The global positions of artifacts don't change, of course. But the dead-reckoning used to update the robot's global position as it moves contains errors, and the robot's global position gradually decays in accuracy. To bring it back into alignment with stationary artifacts, *registration routines* use sensor information to align the robot with recognized objects. These functions are described in a subsequent section.

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

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

4.7.1 Points and Lines

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

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

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

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

Listing 8-4.

The `point` type consists of a directed point (position and direction), with an identifier, a type, a category, and other parameters used by the system. All *x*,*y* coordinates are in millimeters, and direction is in degrees from -180 to 180. The type `POS` is used for goal positions in behaviors. Other types may add additional fields to the basic `point` type– for example, length and width for corridors.

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

Listing 8-5.

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

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

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

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

To keep a point's local coordinates updated within the LPS, it must be added to the *pointlist* after it is created. The pointlist is a list of artifacts that Saphira updates when the robot moves.

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

These functions add and delete members of the pointlist. Ordinarily, to add a point to the pointlist, you use `sfAddPointCheck`, which first checks to make sure point `p` is not in the list already before adding it. It is not a good idea to have two copies of a pointer to a point in the pointlist, because its position will get updated twice. The `sfRemPoint` function removes a point from the list, of course. `sfFindArtifact` returns the artifact on the pointlist with identifier `id`, if it exists; otherwise, it returns `NULL`. Finally, `sfRemArtifact` removes an artifact from the list, given its `id`.

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

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

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

4.7.2 Other Artifact Creation Functions

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

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

Type	Return Value

Table 4-7. Artifact creation types.

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

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

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

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

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

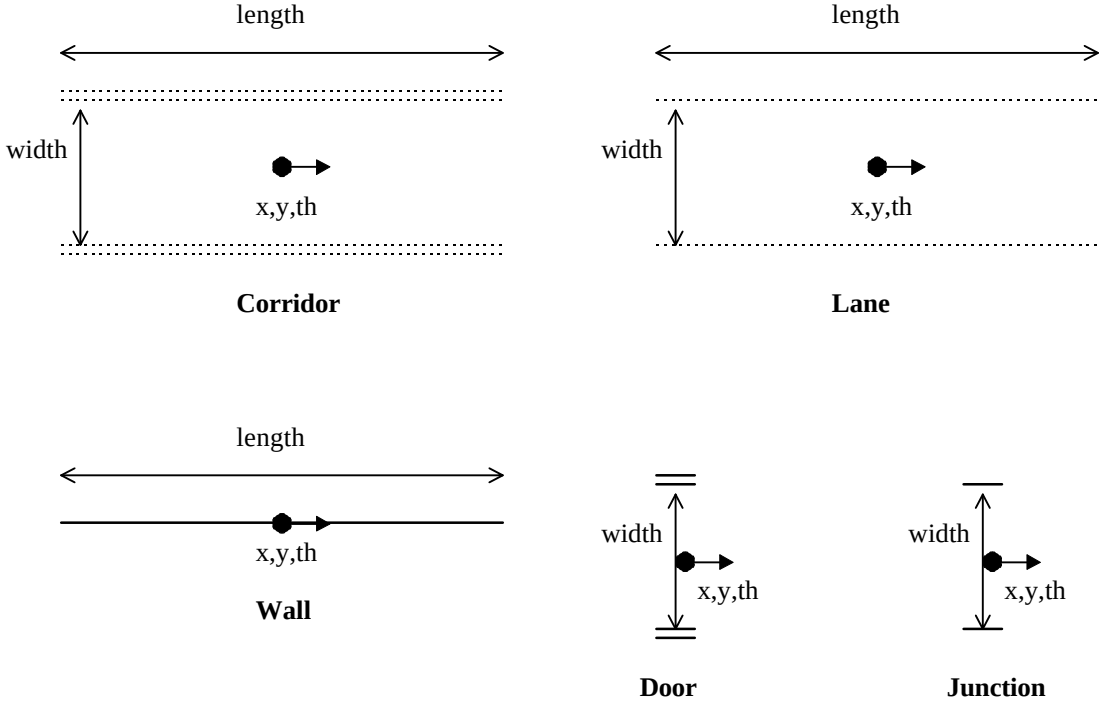

Figure 4-4 Geometry of artifact types. The defining point for the artifact is shown as a vector with a circle at the origin.

Artifacts are most often used in constructing maps for the robot and registering it based on sensor readings (see Section 4.10).

4.7.3 Geometry Functions

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

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

These functions compute angles in the LPS. Normally, angles in the LPS are represented in degrees, using floating-point numbers. Artifact angles are always normalized to the interval $[0,360]$. `sfNormAngle` will put its argument into this range. The corresponding functions `sfAddAngle` and `sfSubAngle` also normalize their results in this way.

It is often convenient to give headings in terms of positive (counterclockwise) and negative (clockwise) angles. The second normalization function, `sfNorm2Angle`, converts its argument to the range $[-180,180]$, so that the discontinuity in angle is directly behind the robot. The corresponding functions `sfAdd2Angle` and `sfSub2Angle` also normalize their results this way.

Finally, it is sometimes useful to reflect all angles into the upper half-plane $[-90,90]$. The function `sfNorm3Angle` will do this to its argument, by reflecting any angles in the lower half-plane around the X-axis; e.g., +100 degrees is reflected to +80 degrees.

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

The first three functions compute properties of points relative to the robot. The function `sfPointPhi` returns the angle of the vector between the robot and point `p`, in degrees from -180 to 180. `sfPointDist` returns the distance from the point to the robot. `sfPointNormalDist` returns the distance from the robot to the line represented by the artifact point; it will be positive if the normal segment is to the left of the robot's *x* axis, and negative if to the right.

The second three functions compute properties of points. `sfPointDistPoint` returns the distance between its arguments. `sfPointNormalDistPoint` returns the distance from point `q` to the line represented by artifact point `p`. The distance will be positive if the normal segment is to the left of `q`'s *x* axis, and negative if to the right. `sfPointBaricenter` sets point `p3` to be the point midway between point `p1` and `p2`.

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

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

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

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

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

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

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

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

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

4.8 Sensor Interpretation

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

```
wall sfLeftWallHyp
 wall sfRightWallHyp
```

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

4.9 Drawing and Color Functions

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

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

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

To draw a rectangle, use the function `sfDrawCenteredRect` or `sfDrawRect`. The centered version takes a center point of the rectangle, and a width and height. The non-centered version takes the lower-left corner position, a width, and a height.

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

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

For lines, set the width `w` to the desired pixel width. This width affects all lines drawn in rectangles and vectors. You may select one of two line types: Set the `w` function parameter to sfLINESOLID for a solid line, and sfLINEDASHED for a dashed line. The patch and line colors accept a color value as shown in Table 4.8.

Table 4.8. Saphira colors.

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

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

4.10 Maps and Registration

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

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

A map can either be created by the robot as it wanders around the environment, or you may create one as a file. You can also save the map created by the robot to a file, for later recall.

4.10.1 Map File Format

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

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

Listing 8-6.

The `CORRIDOR` lines define a series of corridor artifacts. The number in parentheses is the (optional) artifact ID, and it must be a positive integer. The first three coordinates are the *x, y*, and θ position of the center of the corridor in millimeters and degrees. The fourth coordinate is the length of the corridor, and the fifth is the width.

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

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

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

The map file, when loaded into a Saphira client using the Files/Load Map menu (or the function `sfLoadMapFile`), creates the artifact structure shown in Figure 4-5-5. For illustration, the defining point of the artifact is also shown as a small circle with a vector. These points will not appear in the Saphira window.

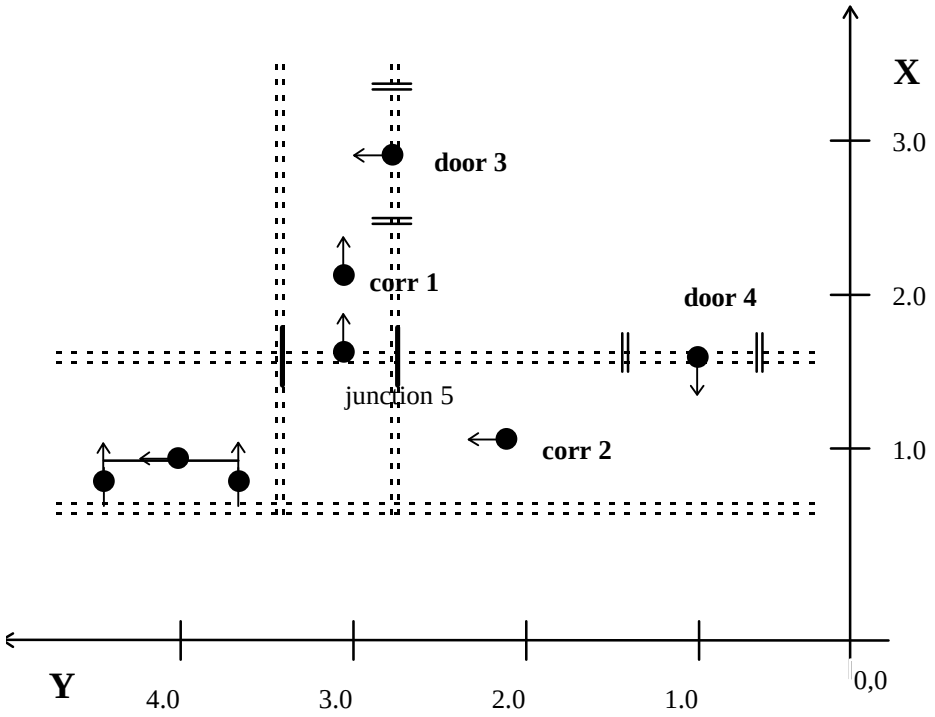

Figure 4-5. Sample map created from the map file above, as shown in a Saphira client. Corridor artifacts display with double dotted lines; doors display with double solid lines; walls display as single solid lines; junctions as pairs of solid lines. Numbers are the artifact ID's. For illustration, the defining vector for each artifact is shown.

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

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

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

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

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

Loaded artifacts are added to any map artifacts already in the system. To delete all map artifacts, use the `sfDeleteMapArtifacts` function. An individual artifact can be deleted using its ID number (see Section 4.7).

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

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

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

4.10.2 Map Registration

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

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

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

4.10.3 Map Element Creation

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

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

Map elements can also be created by hand, using the artifact creation functions of Section 4.7.

4.11 File Loading Functions

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

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

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

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

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

4.12 Colbert Evaluator Functions

Several library functions add functionality to the Colbert evaluator, by linking the evaluator to native C functions, variables, and structures. For examples, see Section **Error! Reference source not found.** on the Colbert language.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4.13 Packet Communication Functions

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

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

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

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

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

This function also sets the global variables sfRobotName, sfRobotClass, and sfRobotSubclass according to the information returned from the robot; see Table 8-10, below. Assuming the environment variable SAPHIRA is set correctly, it will autoload the correct parameter file from the params directory, using first the subclass if it exists, and then the class.

Table 8-10. Robot names and classes.

Structure	Explanation
(char *)sfRobotName	See robot descriptions for information on how to set the name. The simulator returns the name of the machine it is running on.
(char *)sfRobotClass	Robot classes are B14, B21, and Pioneer.
(char *)sfRobotSubclass	Subclasses are subtypes, e.g., in Pioneer-class robots the subclass is either pion1 (Pioneer I) or pionat (Pioneer AT).

void sfDisconnectFromRobot (void)

This structure sends the server a close command, then shuts down the communications channel to the server.

void sfResetRobotVars (void)

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

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

These Saphira functions packetize and send a client command to the robot server. Use the command type appropriate for the type of argument. See Section **Error! Reference source not found.** for a list and description of currently supported PSOS commands.

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

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

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

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

```
void sfProcessClientPacket (int type)
```

`sfProcessClientPacket` parses a client packet into the `sfRobot` structure and sonar buffers. Typically, a client will call `sfWaitClientPacket` or `sfHaveClientPacket` to be sure a packet is waiting to be parsed. The argument to `sfProcessClientPacket` is a byte, the type of the packet. This byte can be read using `sfReadClientByte`. By examining this byte, the client can determine if it wishes to parse the packet itself, or send it on to `sfProcessClientPacket`.

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

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

5 Saphira Vision

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

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

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

5.1 Channel modes

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

BLOB_MODE 0

BLOB_BB_MODE 2

LINE_MODE 1

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

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

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

where the mode `N` is 0, 1, or 2, and the channel `X` is `a`, `b`, or `c` (small letters). On start-up, the vision system channels are set to BLOB_MODE. (The processing performed in BLOB_MODE, BLOB_BB_MODE, and LINE_MODE is explained in the FTVS manual.)

As Table 9-1 shows, several FTVS parameters affect the processing in line mode.

Table 9-1. FTVS parameters used to determine a line segment.

Parameter	**Description**
`line_bottom_row`	First row for line processing
`line_num_slices`	How many rows are processed
`line_slice_size`	How many pixels thick each row is
`line_min_mass`	Number of pixels needed to

These parameters can be set using a command such as the following:

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

5.2 Vision Packets

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

Saphira parses these packets into a vision information structure (see Listing 9-1).

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

Listing 9-1. Saphira vision information structure.

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

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

This constant lets a client convert from image pixel coordinates to angles. The area is the approximate size of the blob in pixels. If the area is 0, no blob was found.

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

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

The following global variables hold information for each channel: `extern struct vinfo sfVaInfo, sfVbInfo, sfVcInfo.`

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

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

5.3 Sample Vision Application

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

This sequence sets up parameters of the vision system, putting all channels into BLOB_BB_MODE and initializing line parameters:

`void setup_vision_system(void)`

This one returns the X-image-coordinate of a blob on channel (`0=A, 1=B, 2=C`), if the blob's center is within `delta` pixels of the center of the image:

`int found_blob(int channel, int delta)`

If no blob is found with these parameters, it returns -1000.

`void draw_blobs(void)`

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

`void find_blob(void)`

This command defines the activity for turning left until a blob is found in the center of the image on channel A, or until 20 seconds elapses.

`void search_and_go_blob(void)`

This command defines the activity for finding a blob (using `find_blob`) on channel A, then approaching it. It uses sonars to detect when it is close to the blob.

6 Parameter Files

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

6.1 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`.

6.2 Sample Parameter File

The sample parameter file in Listing 10-1 illustrates most of the parameters that can be set. This is the file `psos41m.p`. An explanation of the parameters is given in Table 10-1, below.

```
;;
;; Parameters for the Pioneer robot
;; New motors
;;
AngleConvFactor   0.0061359 ; radians per encoder count diff (2PI/1024)
DistConvFactor    0.05066   ; 5in*PI / 7875 counts (mm/count)
VelConvFactor     2.5332     ; mm/sec / count  (DistConvFactor * 50)
RobotRadius       220.0      ; radius in mm
RobotDiagonal              90.0   ; half-height to diagonal of octagon
Holonomic         1          ; turns in own radius
MaxRVelocity      2.0        ; radians per meter
MaxVelocity               400.0         ; mm per second

;;
;; Robot class, subclass
;;
Class  Pioneer
Subclass  PSOS41m
Name   Erratic

;; These are for seven sonars: five front, two sides
;;
;; Sonar parameters
;; SonarNum N is number of sonars
;; SonarUnit I X Y TH is unit I (0 to N-1) description
;;        X, Y are position of sonar in mm, TH is bearing in degrees
```

```
;;
```

Listing 10-1. The example parameter file, `psos41m.p`, shows how to set most Saphira parameters.

```
RangeConvFactor            0.1734      ; sonar range mm per 2 usec tick
;;
SonarNum 7
;;          #   x   y   th
;;-------------------------
SonarUnit  0  100 100  90
SonarUnit  1  120  80  30
SonarUnit  2  130  40  15
SonarUnit  3  130   0   0
SonarUnit  4  130 -40 -15
SonarUnit  5  120 -80 -30
SonarUnit  6  100 -100 -90
SonarUnit  7    0   0   0

;; Number of readings to keep in circular buffers
FrontBuffer 20
SideBuffer  40
```

Listing 10-2.

Floating-point parameters can be in any standard format and do not require a decimal point. Integer parameters may not have a decimal point. Strings are any sequence of non-space characters.

Table 10-1. Functions of Saphira parameters.

Parameter	Type	Description
`AngleConvFactor`	float	Converts from robot angle units (4096 per revolution) to radians.
`VelConvFactor`	float	Converts from robot velocity units to mm/sec
`DistConvFactor`	float	Converts from robot distance units to mm
`DiffConvFactor`	float	Converts from robot angular velocity to rads/sec
`RangeConvFactor`	float	Converts from robot sonar range units to mm
`Holonomic`	integer	Value of 1 says the robot is holonomic (can turn in place); value of 0 says it is nonholonomic (front-wheel steering). Holonomic robot icon is octagonal; nonholonomic is rectangular.
`RobotRadius`	float	Radius of holonomic robot in mm.
`RobotDiagonal`	float	Placement of the horizontal bar indicating the robot's front, in mm from the front end. (Sorry about the name.)
`RobotWidth`	float	Width of nonholonomic robot, in mm.
`RobotLength`	float	Length of nonholonomic robot, in mm.
`MaxVelocity`	float	Maximum velocity of the robot, in mm/sec.
`MaxRVelocity`	float	Maximum rotational velocity of the robot in degrees/sec.
`MaxAcceleration`	float	Maximum acceleration of the robot in mm/sec/sec
`Class`	string	Robot class: `pioneer`, `b14`, `b21`. Not case-sensitive. Useful only for the simulator, which will assume this robot personality. The client gets this info from the autoconfiguration packet.
`Subclass`	string	Robot subclass. For the Pioneer, indicates the type of controller and

		body combination. Values are `psos41m`, `psos41x`, or `pionat`. Not case-sensitive. Useful only for the simulator, as for the `Class` parameter.
`Name`	string	Robot name. Useful only for the simulator, as for the `Class` parameter.
`SonarNum`	integer	Number of active sonars.
`SonarUnit`	`n,x,y,th`	Description sonar unit `n`. The `x,y,th` arguments describe the pose of the sonar on the robot body, relative to the robot center. Provide one such entry for each active sonar unit. Used by both the simulator and client.
`FrontBuffer`	integer	Number of front sonar readings to keep. Higher values mean the robot will be more sensitive to obstacles but slower to get rid of moving obstacle readings.
`SideBuffer`	integer	Number of side sonar readings to keep. Higher values mean the interpretation routines can find longer side segments.

7 Sample World Description File

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

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

Any entry in the world file that starts with a number is interpreted as creating a single line segment. The first two numbers are the *x*,*y* coordinates of the beginning and the second two are the coordinates of the end of the line segment. The coordinate system for the world starts in the lower left, with `+Y` pointing up and `+X` to the right (Figure 11-1).

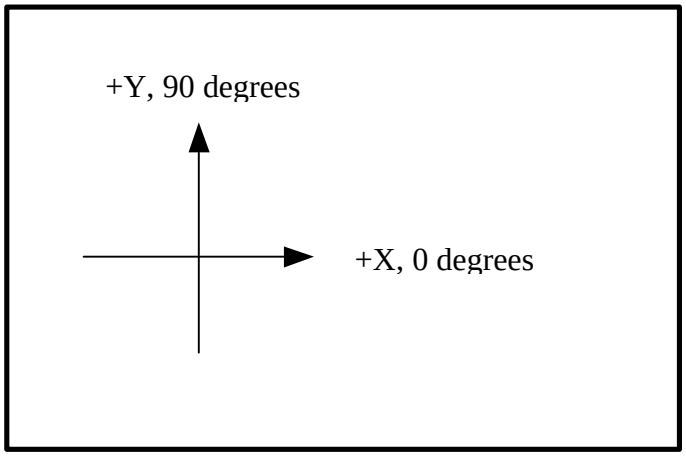

Figure 11-1. Coordinate system for world definition.

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

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

Listing 11-1 is a fragment of the `simple.wld` world description file found in Saphira's `worlds` directory.

```
;;; Fragment of a simple world

width 38000
height 30000

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

```
push 10000 14000 0

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


;; Starting position

position 17500 14000 -90
```

Listing 11-1. Fragment of the `simple.wld` world description file found in Saphira's `worlds` directory.

8 Saphira API Reference

Artifacts

Behaviors

Behaviors; Predefined Saphira

Direct Motion Control

Drawing and Color

Fuzzy Variables

Activities

Map File

Occupancy

OS and Window Functions

Packet Functions

Processes

Processes; Predefined

Sensor Interpretation

Sonars

State Reflection

Vision

9 Index

10 Warranty & Liabilities

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

Saphira Software Manual Version 6.2, August 1999