SAPHIRA TUTORIAL

Compiling, Loading and Debugging C++ Files: Unix Systems
Software version 8.0 (Saphira/Aria)
September 2001

©Kurt Konolige
SRI International
konolige@ai.sri.com
http://www.ai.sri.com/~konolige

1 Introduction

This tutorial introduces loadable object files as a means of extending the functionality of Saphira. A simple example of such a file, compiled from C++ code, is given. The source and makefile for the sample object file can be found in the `Saphira/tutor/loadable` directory.

2 Shared Object Files under UNIX

Most UNIX systems allow object files to be dynamically loaded into a running program, using the `dl` functions (`dlopen`, `dlsym`, `dlclose`). Object files can be compiled from any source, but usually come from C or C++ programs. A special linking operation on these object files creates *shareable* object files, which contain information that allows them to be loaded into an executing program. Typically they will have the suffix `.so`.

Saphira takes advantage of the dynamic load facility under UNIX. Users can write C/C++ programs that are then compiled into shared object files.. These object files can be loaded into a running Saphira client, and the functions in them made available through the Colbert interface. In this way, the user can extend Saphira with new functions written in C/C++.

As of Saphira 8.x, the Saphira library is written as a set of classes in C++, and is built on top of the Aria system from ActivMedia Robotics. User programs access basic Saphira and Aria services through these classes. User programs can also be written a mixture of C and C++, using standard interfacing between the languages.

Because of limitations in the syntax and semantics of Colbert, it is not possible to express interfaces to C++ objects or their member functions directly. Instead, users must write C or C++ global functions accessing the C++ objects, and then make these functions available for calling from Colbert. The objects themselves can be passed around as NULL pointers within C programs and Colbert.

2.1 Compiling and Linking C/C++ Source Files

To compile a loadable shared object file, you must have installed the Saphira and Aria distributions according to the directions in the `readme` file. In particular, the environment variables `SAPHIRA and ARIA` must be set to the top levels of their respective distributions. In addition, Saphira and Aria distributions should be set up as subdirectories of a common directory.

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

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

In Saphira 8.x, all the Saphira library routines are contained in a shared library. In UNIX systems, it is the shared library `libsf.so`. Aria routines are also accessed through a shared library, `libAria.so`.

2.2 Writing C or C++ Client Programs

To load new C/C++ routines into Colbert, you write one or more C/C++ programs that contain your own functions, and make calls to the Saphira library routines. It may help to review Chapter 2 of the Saphira manual for an explanation of micro-tasks and asynchronous user routines. Note that there are very few remaining Saphira functions that are C functions, and all Aria functions are written in C++. So user programs will generally have at least on C++ file to access the Saphira/Aria libraries.

Figure 2-1 and Figure 2-2 show the C++ source code for the sample `tutor/loadable/testload.cpp` example. The header file `Saphira.h` must be included at the beginning of the source file. The rest of the file contains C++ and C function and variable definitions.

The `sfLoadInit` function is called after loading the file, and it typically makes the objects in the file available to the Colbert evaluator, through use of `sfAddEvalXXX` function calls. For information on the effect of these calls, see the Colbert User's Manual. The `sfLoadExit` function is called when the file is unloaded, and typically cleans up by getting rid of micro-tasks that are invoked by the file, etc.

These functions have the keyword `SFEXPORT` as part of their definition. For UNIX, this keyword is defined as the empty string, and doesn't affect the compilation or loading process. It's necessary under MS Windows, however, and it's good practice to include it so that your code will compile and run under either OS.

```
#include "Saphira.h"

// global integer variables
// on some systems, these will not be initialized on load...
int nopen = 0;
int globalvar = 1;

// string variable
char buf[80];
char *mystring;                  // we need to set aside space explicitly for a
string ptr


//
// C++ global functions
// Set up functions to create and manipulate a point artifact
// This point will draw on the screen
//

SfPoint *mypoint;

int                         // sets up a point artifact
myfn(int a)
{
  sfMessage("Argument to myfn is: %d",a);
  mypoint = new SfPoint(a,0,0);
  return a+2;
}

void                        // prints values of a point artifact
showpoint(void *p)
{
  SfPoint *pt = (SfPoint *)p;
  sfMessage("Point x: %d  y: %d  th: %d",
            (int)pt->p.getX(), (int)pt->p.getY(), (int)pt->p.getTh());
}

void                        // sets the X and Y values of a point
setpoint(void *p, int x, int y)
{
  SfPoint *pt = (SfPoint *)p;
  pt->p.setX((double)x);
  pt->p.setY((double)y);
  sfMessage("Point x: %d  y: %d  th: %d",
            (int)pt->p.getX(), (int)pt->p.getY(), (int)pt->p.getTh());
 }
```

Figure 2-1 Listing of the testload.cpp program (part one of two).

```
//
// C++ global functions
// Set up an interface to the sonar buffers
//

int
get sonar dist(int begang, int endang, int *retang)
{
  SfSonarDevice *sd = Sf::sonar(); // get the device
  if (!sd)
    return 100000;         // large value, no return
  double ret;
  ArPose rpose = SfROBOT->getPose();
  // Aria occupancy function on the sonar current reading buffer
  double fdist = sd->getCurrent()->getClosestPolar((double)begang,
(double)endang,
                                       rpose, 100000, &ret);
  if (retang)
    *retang = (int)ret;
  return (int)fdist;
}


//
// sfLoadInit and sfLoadExit must be declared as exported fns
// under MS Windows
//


SFEXPORT void
sfLoadInit(void)           // this function is evaluated when the object file is
loaded
{
  sfMessage("Opened: %d", nopen); // on open, we should get a message
  printf("Opened!\n");           // let the console know too...

  // define constants and variables
  //  sfAddEvalConst("sfLEFT",  sfINT, 0);
  sfAddEvalVar("gv", sfINT, (fvalue *)&globalvar); // note indirection in the
last argument!
  sfAddEvalVar("mypoint", sfPTR, (fvalue *)&mypoint); // note indirection in the
last argument!
  mystring = buf;          // set up string ptr
  mystring[0] = 'A';
  mystring[1] = 0;
  sfAddEvalVar("mystring", sfSTRING, (fvalue *)&mystring); // note the
indirection in the last argument!


  // define functions
  // any C++ global functions will give an error here if overloaded
  sfAddEvalFn("myfn", (void *)myfn, sfINT, 1, sfINT);
  sfAddEvalFn("showpoint", (void *)showpoint, sfVOID, 1, sfPTR);
  sfAddEvalFn("setpoint", (void *)setpoint, sfVOID, 3, sfPTR, sfINT, sfINT);
  sfAddEvalFn("get sonar dist", (void *)get sonar dist, sfINT, 3, sfINT,
            sfINT, sfTypeRef(sfINT));  // last arg is pointer to int
}


SFEXPORT void
sfLoadExit(void)           /* this should be evaluated on unload */
{
  sfMessage("Unloading: %d", nopen);
}
```

Figure 2-2 Listing of the testload.cpp program (part two of two).

3 Compiling and Linking Client Programs under UNIX

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

The compiler and linker are typically called using the `make` facility. The file `makefile` in the `Saphira/tutor/loadable` directory is used to make the `testload.cpp` program. Figure 3-1 shows this makefile. The first part of the `makefile` defines variables that are useful in compilation and linking. Note that the Saphira and Aria top-level directories have to share a common parent directory. The `ohandler/include` directory contains header files, and `ohandler/lib` has the Saphira libraries, and is where compiled shared objects typically are deposited.

Next, the file `handler/include/os.h` is read in. This file determines the operating system type and sets some system library variables appropriately. NOTE: `os.h` also sets the `CONFIG` variable to the particular OS of the machine, which is important for handling some of the system routines correctly. One peculiarity of `os.h` is that it relies on the conditional preprocessing facilities of gnu `make` (`gmake`). Not all native `make`s support this facility. If you get errors during the preprocessing phase of the compilation from `os.h`, switch to `gmake`.

The makefile produces the file `testload.so` in the Saphira library directory (`${SAPHIRA}/lib`). `testload.so` is a shared object file, which is loadable under Colbert. The C++ source is compiled as usual; make sure the `-g` flag is present so that debugging symbols are added to the object file. Under some UNIXes, the source must be compiled as position-independent (PIC) in order to create a shared object file. This detail is handled by the `PICFLAG` variable, set by `os.h`. Next, the `LD` command is invoked to create a shared object file with the extension `.so`. Saphira expects this extension for UNIX shareable objects, and cannot load objects with any other extensions. The most critical part of the linking operation is the inclusion of the `SHARED` flag. You must include this variable, defined in `os.h` for each particular OS, in order to create a shared object file.

```
##
## Makefile for the loadable tutorial
##

SRCD = ./
OBJD = ./
INCD = ../../ohandler/include/
BIND = ../../bin/
LIBD = ../../lib/
ARIAD = ../../../Aria/

# check which OS we have
include $(INCD)os.h

SHELL = /bin/sh

INCLUDE = -I$(INCD) -I$(ARIAD)include

############################################################
all: $(LIBD)testload.so
	touch all

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

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

Figure 3-1 Makefile for the testload.cpp program.

3.1 Loading Shared Objects into a Saphira Client

Shared object files, with extensions ending in `.so`, can be loaded into a Saphira client using the Colbert interaction window. The `loadlib` command will attempt to load a shared object file, using the current directory first, and then the default directories, which always include $(SAPHIRA)/lib. To load the example testload.so, use the following command

```
loadlib testload
```

Note that the extension `.so` isn't necessary in specifying the file for the `loadlib` command.

The `loadlib` command, if successful, prints out the name of the file loaded, and then declares whether it found an `sfLoadInit` function to evaluate. In this case, when `sfLoadInit` is evaluated it prints the message `Opened: 1` using the `sfMessage` call.

Unloading the shared object file can be accomplished with the `unload` command. Generally, under UNIX systems there's no need to explicitly unload a file, since re-loading the same file will cause an automatic unload first. The `unload` command is mainly a convenience for MS Windows systems, where a shared object file cannot be recompiled if it is in use.

If there is an error in your shared object code, you can make changes, recompile and relink the code, and then reload it using the `load` command again. The new function definitions will replace the old ones. One possible problem with this process is that the Saphira client may be invoking some functions of the shared object, usually in a microtask. Reloading does not update function or other pointers into the original loaded object that may exist. The microtask must be exited before the reload operation is performed, since otherwise it will try to execute nonexistent function code, as the original loaded object code is unloaded automatically.

One consequence of reloading is that the previous values of any global variables in the shared object file are lost, and the variable is reset to its initialization value. For example, the variable `nopen` is always initialized to 0 when `testload.so` is loaded. Even though it is incremented by the call to `sfLoadInit`, it gets reset to its initialization value whenever `testload.so` is reloaded, and so never increments above 1.

Within the Colbert interaction window, several functions and variables from the shared object file are made available. Evaluating `myfn`, for example, will set up a new point artifact, and then return its argument plus 2.

3.2 Initializations on Loading

There is one peculiarity of shared object files that are dynamically loaded. Unlike other object files, variables which are declared globally initialized *may not be initialized* when the file is loaded. For example, the following line in the `testload.cpp` example initialized a global variable:

```
int globalvar = 1;
```

The variable `globalvar` may not be set to 1 upon loading the resultant shared object file. In fact, under the Linux system it typically is; under MS Windows it isn't. The only way to initialize variables securely is with the `sfLoadInit` function.

4 Debugging C Code under UNIX

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

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

Figure 4-1 shows typical output from the GDB debugger. GDB is started by giving it the Saphira executable `bin/saphira`. Next, Saphira itself is started with the `run` command. At this point, the Saphira client doesn't even have a window system available, so it prints out messages on the console. First, it finds its home directory (using the `SAPHIRA` environment variable) and sets up the Colbert directory path from that. Then, it reads the activity file `startup.act` from the Colbert directory. You can change `startup.act` to have any initial behavior you want, but the default file loads the windowing system (`fltk.so` and `fltkwin.so`), after which messages will print on the Colbert window. The rest of `startup.act` loads some basic activities for the robot, and then Saphira continues with its main loop, and several threads are spawned. Saphira itself uses one main thread to control its 100 ms main cycle.

At this point, in the Saphira client window, `testload.so` is loaded. The `printf` statement in `sfInitLoad` prints the `Opened!` line on the GDB console.

4.1 Setting a Breakpoint

Since our user program is now loaded, we can set a breakpoint. First, the Saphira client is interrupted by pressing Ctrl-C in the GDB window (actually, you need two consecutive Ctrl-C's in the Emacs GDB window). A breakpoint is inserted at the `myfn` function. It's also possible to scout around and check the values of variables, for example, the value of `globalvar`.

A subtle aspect of GDB is how it handles shared libraries and their symbols. In this version of GDB under Linux, shared object file symbols are automatically made available in the debugger. The `sharedlibrary` command prints info about all shared libraries known by the debugger; here, all of the symbols in these files are loaded.

Under some OSes, such as Solaris, the symbols of shared object files are not automatically loaded into GDB when the shared object is dynamically loaded. In this case, you must interrupt the debugger with Ctrl-C, and issue the command `sharedlibrary testload.so` to make its symbols visible to the debugger.

To continue from the break, use the "c" command; sometimes this has to be issued twice to continue, apparently because of GDB's handling of threads.

4.2 Debugging from a Breakpoint

Continuing from the break, the user then evaluates `myfn(1000)` in the Colbert interaction window. This interrupts the debugger, and now the user can examine the stack, change data values, and so on, debugging this function (Figure 4-2).

To find out where the breakpoint function is being called from, issue the command `bt` (backtrace). This prints out the function call stack, from the most recently called function. [Anything below the call to `eval_fexpr` is internal to Saphira and the Colbert interpreter, and not particularly useful.]

One nice feature of GDB is the ability to step through lines of your code. Under Emacs, a second buffer will open, showing the position of the program counter in your source code. Here, we step through, and GDB prints out each line as it is executed. After the variable `mypoint` is set to an object pointer, we

```
[konolige@dill saphira]$ gdb bin/saphira
GNU gdb 19991004
Copyright 1998 Free Software Foundation, Inc.
…
(gdb) run
Starting program: /home/konolige/saphira/bin/saphira
starting...
SAPHIRA directory is /home/konolige/saphira
Started basic processes
Load directory is /home/konolige/saphira/colbert
Loading file /home/konolige/saphira/colbert/startup.act
Loading shared object file /home/konolige/saphira/colbert/../lib/fltk.so
No initializer
 Eval to (void)
Loading shared object file /home/konolige/saphira/colbert/../lib/fltkwin.so
Evaluating sfLoadInit
Starting main frame...
[New Thread 31076 (manager thread)]
[New Thread 31075 (initial thread)]
[New Thread 31077]
[Switching to Thread 31077]
Opened!

Program received signal SIGINT, Interrupt.
0x4017cc61 in __libc_nanosleep () from /lib/libc.so.6
(gdb) òQuit
(gdb) break myfn
Breakpoint 1 at 0x400711cb: file ./testload.cpp, line 52.
(gdb) p globalvar
$1 = 1
(gdb) p mypoint
$2 = (sfPoint *) 0x0
(gdb) c
Continuing.
[Switching to Thread 31075 (initial thread)]

Program received signal SIGINT, Interrupt.
0x4017cc61 in __libc_nanosleep () from /lib/libc.so.6
(gdb) c
Continuing.
```

Figure 4-1 Listing from a running GDB debugging session.

can look at its value, and check that the object has the correct position (1000,0,0). Variables and objects can also have their values set by GDB commands.

```
[Switching to Thread 31077]

Breakpoint 1, myfn (a=1000) at ./testload.cpp:52
warning: Source file is more recent than executable.

52      {
Current language:  auto; currently c++
(gdb) bt
#0  myfn (a=1000) at ./testload.cpp:52
#1  0x4003c59a in eval_fexpr (e=0x805bd68) at src/control/expr.c:1450
#2  0x40037ea7 in do_eval (e=0x805bd68) at egi.y:829
#3  0x40039d58 in yyparse () at egi.y:2144
#4  0x4003776b in parse_command () at egi.y:510
#5  0x4003437d in run_process (p=0x40064848) at src/basic/process.c:603
#6  0x40034404 in run_all_processes () at src/basic/process.c:629
#7  0x4004846c in update_and_comm () at src/basic/toplevel.c:779
#8  0x400482a6 in sfDoToplevel () at src/basic/toplevel.c:652
#9  0x40047beb in doact (signum=0) at src/basic/toplevel.c:213
#10 0x40047c15 in dotop (a=0x0) at src/basic/toplevel.c:233
#11 0x4007db85 in pthread_start_thread (arg=0xbf7ffe40) at manager.c:241
(gdb) n
53        sfSMessage("Argument to myfn is: %d",a);
(gdb) n
81          { p.x = x; p.y = y; p.th = th;  type = sfPOINT; }
(gdb) n
54        mypoint = new sfPoint(a,0,0);
(gdb) n
55        return a+2;
(gdb) p mypoint
$3 = (sfPoint *) 0x805bde0
(gdb) p *mypoint
$4 = {<sfArtifact> = {<sfDrawable> = {visible = 1, color = 0, _vptr. =
      0x40058280}, p = {<sfPosition> = {x = 1000, y = 0}, th = 0},
      type = 1, id = 0, newflags = -1, static   newid = 1}, <No data fields>}
(gdb)
```

Figure 4-2 Breakpoint execution from GDB.