Chapter 3

Dynamic Shared Objects

A dynamic shared object (DSO) is an object file that's meant to be used simultaneously—or *shared*—by multiple applications (*a.out* files) while they're executing. DSOs can be used in place of archive libraries, and they replace static shared libraries provided with earlier releases of IRIX.

Since DSOs contain shared components, using them provides several substantial benefits. First, overall memory usage is minimized because code is shared. Two executables that use the same DSO and that run simultaneously will have only one copy of the shared components loaded into memory. For example, if executable A and executable B both link with the same DSO C, and if A and B are both running at the same time, the total memory used is what's required for A, B, and C, plus some small overhead. If C was an unshared library, the memory used would be what's required for A, B, and two copies of C. A related second benefit is that executables linked with DSOs are smaller than those linked with unshared libraries because the shared objects aren't part of the executable file image, so disk usage is minimized.

Yet another feature of DSOs is that they are much easier to use, build, and debug than static shared libraries. Most of the libraries supplied by Silicon Graphics are available as DSOs. In the past, only a few static shared libraries have been available; most libraries were unshared.

Additionally, executables that use a DSO don't have to be relinked if the DSO changes; when the new DSO is installed, the executable automatically starts using it. This feature makes it easier to update end users with new software versions. It also allows you to create hardware-independent software packages more easily. You can design the hardware-dependent routines required by your application so that they have the same interface across all platforms. Then, you create different DSOs for each of the platforms, each DSO containing the implementation of those routines for that particular platform. The shrink-wrapped software package can then contain all the DSOs and is able to run on all the platforms.

Finally, DSOs and the executables that use them are mapped into memory by a run-time loader, *rld*, which resolves external references between objects and relocates objects at run time. (DSOs contain only position-independent code (PIC), so they can be loaded at any virtual address at run time.) With *rld*, the binding of symbols can be changed at run time at the request of the executing program. You could use this feature to dynamically change the feature set presented to a user of your application, for example, while minimizing start-up time. The application could be started quickly, with a subset of the features available and then, if the user needs other features, those can be loaded in under programmatic control.

Naturally, some costs are involved with using DSOs, and these are explained in the next section, "Using DSOs." The sections after that explain how to build and optimize DSOs and how *rld* works. The *dso*(5) reference page also contains more information about DSOs.

- "Using DSOs" tells you how to obtain the most benefit from using DSOs when creating your executable.
- "Taking Advantage of QuickStart" discusses an optimization you can use to make sure that DSOs you build load as quickly as possible.
- "Building DSOs" describes how to build a DSO.
- "Runtime Linking" discusses the runtime linker, and how it locates DSOs at runtime.
- "Dynamic Loading Under Program Control" explains the use of the *libdl* library to control runtime linking.
- "Versioning of DSOs" discusses a versioning mechanism for DSOs that allows binaries linked against different, incompatible versions of the same DSO to run correctly.

Using DSOs

Using DSOs is easy—the syntax is the same as for an archive (*.a*) library. The following compile line creates the executable *yourApp* by linking with the DSOs *libyours.so* and with *libc.so.1*:

```
cc yourApp.c -o yourApp -lyours
```

If *libyours.so* isn't available, but the archive version *libyours.a* is available, that archive version is used along with *libc.so.1*.

You should note that there's a significant difference between DSOs and archive libraries in terms of what gets loaded into memory when an application is executing. With an archive library, only the text portion of the library that the application actually requires (and the data associated with that text) gets loaded, not the entire library. In contrast, the entire DSO that's linked gets loaded. Thus, to conserve memory, don't link with DSOs unless your application actually needs them.

Also, you should avoid listing any archive libraries on the compile line after you list shared libraries; instead, list the archive libraries first and then the DSOs.

You may want to take advantage of the QuickStart optimization that minimizes start-up times for executables. You can use QuickStart when using or building DSOs. At link time, when an executable or a DSO is being created, the link editor *ld* assigns initial addresses to the object and attempts to resolve all references. Since DSOs are relocatable, these initial address assignments are really only guesses about where the object will be really loaded. At run time, *rld* verifies that the DSO being used is the same one that was linked with and what the real addresses are. If the DSOs are the same and if the addresses match the initial assignments, *rld* doesn't have to perform any relocation work, and the application starts up quickly (or QuickStarts). When an application QuickStarts, memory use is smaller since *rld* doesn't have to read in the information necessary to perform relocations.

To determine whether your application (or DSO) is able to do a QuickStart, use the *–quickstart_info* flag when building the executable (or DSO). If the application or DSO can't do a QuickStart, you'll be given information about how what to do. The next section goes into more detail about why an executable may not be able to use QuickStart.

In summary, when you use DSOs to build an executable:

- Link with only the DSOs that you need.
- Make sure that unshared libraries precede DSOs on the compile line.
- Use the *–quickstart_info* flag.

Taking Advantage of QuickStart

QuickStart is an optimization designed to reduce start-up times for applications that link with DSOs. Each time *ld* builds a DSO, it updates a registry of shared objects. The registry contains the preassigned QuickStart addresses of a group of DSOs that typically cooperate by having nonoverlapping locations. (See “Using Registry Files” for more information about how to use the registry when you’re building a DSO.) If you compile your application by linking with registered DSOs, your application takes advantage of QuickStart: all the DSOs are mapped at their QuickStart addresses, and *rld* won’t need to move any of them to an unused address and perform a relocation pass to resolve all references.

Suppose you compile your application using the *–quickstart_info* flag, and it fails. QuickStart may fail because:

- Your application has directly or indirectly linked with two different versions of the same DSO, as shown in Figure 3-1. In this example, *yourApp* links with *libyours.so*, *libmotif.so*, and *libc.so.1* on the compile line. When the DSO *libyours.so* was built, however, it linked with *libmalloc.so*, which in turn linked with *libc.so.1* when it was created. If the two versions of *libc.so.1* aren’t identical, *yourApp* won’t be able to QuickStart.

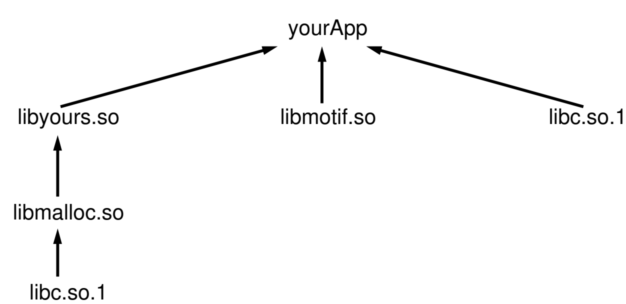

Figure 3-1 An Application Linked with DSOs

- You link with a DSO that can’t QuickStart. This may occur because the DSO wasn’t registered and therefore was assigned a location that overlaps with the location assigned another DSO.

- Your application pulls in incompatible shared objects (in a manner similar to the example shown in Figure 3-1).
- Your application contains an unresolved reference to a function (where it takes the address of the function).
- The DSO links with another DSO that can't QuickStart.

Even if QuickStart officially succeeds, your application may have name space collisions and therefore may not start up as fast as it should. This is because *rld* has to bring in more information to resolve the conflicts. In general, you should avoid having conflicts both because of the detrimental effect on start-up time and because conflicts make it difficult to ensure the correctness of an application over time.

In the example shown in Figure 3-1, you may have written your own functions to allocate memory in *libmalloc.so* for *libyours.so* to use. If you didn't use unique names for those functions (instead of **malloc()**, for example) the way this particular compile and link hierarchy is set up, the standard **malloc()** function defined in *libc.so.1* is used instead of the one defined in *libmalloc.so*. (Conflicts are resolved by proceeding through the hierarchy from left to right and then moving to the next level. See "Searching for DSOs at Runtime" for more information about how the runtime linker searches for DSOs.)

Thus, it's not a good idea to allow more than one DSO to define the same function. Even if the DSOs are synchronized for their first release, one of them may change the definition of the function in a subsequent release. Of course, you can use conflicts to override function definitions intentionally, but you should be sure you have control over what is overriding what over time.

If you use the *-quickstart_info* option, *ld* tells you if conflicts arise. It also tells you to run *elfdump* with the *-Dc* option to find the conflicts. See the *elfdump*(5) man page for more information about how to read the output produced by *elfdump*.

Building DSOs

In most cases, you can build DSOs as easily as archive libraries. If your library is written in a high-level language, such as C or Fortran, you won't have to make any changes to the source code. If your code is in assembly language, you must modify it to produce PIC, as described in Appendix A, "Position-Independent Coding in Assembly Language."

To create a DSO from a set of object files, use *ld* with the *–shared* option:

```
ld -shared stuff.o nonsense.o -o libdada.so
```

The above example creates a DSO, *libdada.so*, from two object files, *stuff.o* and *nonsense.o*. Note that DSO names should begin with "*lib*" and end with "*.so*", for ease of use with the compiler driver's *–l*lib argument. If you're already building an archive library (*.a* file), you can create a DSO from the library by using the *–shared* and *–all* arguments to *ld*:

```
ld -shared -all libdada.a -o libdada.so
```

The *–all* argument specifies that all of the object files from the library, *libdada.a*, should be included in the DSO.

Making DSOs Self-Contained

When building a DSO, be sure to include any archives required by the DSO on the link line so that the DSO is self-contained (that is, it has no unresolved symbols). If the DSO depends on libraries not explicitly named on the link line, subsequent changes to any of those libraries may result in name space collisions or other incompatibilities that can prevent any applications that use the DSO from doing a QuickStart. Such incompatibilities can also lead to unpredictable results over time as the libraries change asynchronously. To make the archive *libmine.a* into a DSO, for example, and *libmine.a* depends on routines in another archive, *libutil.a*, include *libutil.a* on the link line:

```
ld -shared -all -no_unresolved libmine.a -o libmine.so -none libutil.a
```

This causes the modules in *libutil.a* that are referenced in *libmine.a* to be included in the DSO, but these modules won't be exported. See "Controlling Symbols to be Exported or Loaded" for more information about exported symbols. The *–no_unresolved* option causes a list of unresolved symbols to be created; generally, this list should be empty to enable QuickStarting.

Similarly, if a DSO relies on another DSO, be sure to include that DSO on the link line. For example:

```
ld –shared –all –no_unresolved libbtree.a –o libtree.so –lyours
```

This example places *libyours.so* in the *liblist* of the new DSO, *libtree.so*. This ensures that *libyours.so* is loaded whenever an executable that uses *libtree.so* is launched. Again, symbols from *libyours.so* won't be exported for use by other libraries. (You can use the *–exports* flag to reverse this exporting behavior; the *–hides* flag specifies the default exporting behavior.)

Controlling Symbols to be Exported or Loaded

By default, to help avoid conflicts, symbols defined in an archive or a DSO that's used to build another DSO aren't externally visible. You can explicitly export or hide symbols with the *–exported_symbol* and *–hidden_symbol* options:

```
–exported_symbol name1, name2, name3
```

```
–hidden_symbol name4, name5
```

By default, if you explicitly export any symbols, all other symbols are hidden. If you both explicitly export and explicitly hide the same symbol on the link line, the first occurrence determines the behavior. You can also create a file of symbol names (delimited by white space) that you want explicitly exported or hidden, and then refer to the file on the link line with either the *-exports_fil*e or *-hiddens_file* option:

```
–exports_file yourFile
```

```
–hiddens_file anotherFile
```

These files can be used in addition to explicitly naming symbols on the link line.

Another useful option, *–delay_load*, prevents a library from being loaded until it's actually referenced. Suppose, for example, that your DSO contains several functions that are likely to be used in only a few instances. Furthermore, those functions rely on another library (archive or DSO). If you specify *–delay_load* for this other library when you build your DSO, the runtime linker loads that library only when those few functions that require it are used. Note that if you explicitly export any symbols defined in a library that the runtime linker is supposed to delay loading, the export behavior takes precedence and the library is automatically loaded at runtime.

Note: You can build DSOs using *cc*. However, if you want to export symbols/files or use *–delay_load*, use *ld* to build DSOs. ♦

Using DSOs With C++

To make a DSO, build the C++ objects as you would normally:

```
CC -c
```

Then type:

```
CC -shared -o libmylib.so <list your objects here>
```

For example:

```
CC -shared -o libmylib.so a.o b.o c.o
```

In this instance, the *–l* and *–L* options to *ld* will work. However, most *ld* options won't work. If you want to specify other options, first determine the options that you must pass to *ld*. These options include:

```
-init _main
-fini _fini
-hidden_symbol _main
-hidden_symbol _fini
-hidden_symbol __head
-hidden_symbol __endlink
```

Finally, link in */usr/lib/c++init.o*.

Using Registry Files

You can make sure that your DSOs don't conflict with each other by using a QuickStart registry file. The registry files contain location information for shared objects. When creating a shared object, you can specify a registry file to *ld*, and *ld* ensures that your shared object doesn't conflict with any of the shared objects listed in the registry. A registry file containing the locations of all the shared objects provided with the system is supplied in */usr/lib/so_locations*.

You can use two options to *ld* to specify a registry file: *–check_registry* and *–update_registry*. When you invoke *ld* to build a shared object, with the argument *–check_registry file*, *ld* makes sure that the new shared object doesn't conflict with any of the shared objects listed in *file*. When invoked with *–update_registry file*, *ld* checks the registry in the same way, but when it's done, it writes an entry in *file* for the DSO being built. If *file* isn't writable, *–update_registry* acts like *–check_registry*. If *file* isn't readable, both *–update_registry* and *–check_registry* are ignored.

By exchanging registry files, providers of DSOs can avoid collisions between their shared objects. You should probably start out with a copy of */usr/lib/so_locations*, so that your shared objects won't conflict with any of the standard DSOs. However, you should remember that when collisions occur between shared objects, the only effect is slowing program startup.

Registry File Format

Three types of lines in the registry file include:

- comment lines, which begin with a pound sign (#)
- directive lines, which begin with a dollar sign ($)
- shared object specification lines, which begin with the name of a shared object

Comment lines are ignored by *ld*. Directive lines and shared object specification lines are described below.

Directive Lines

Directive lines specify global parameters that apply to all the DSOs listed in the registry.

```
$text_align_size=align padding=pad-size
$data_align_size=align padding=pad-size
```

These two directives specify the alignment and padding requirements for text and data segments, respectively. The current default segment alignment is 64K, which is the minimum permissible. The size value of a segment of a DSO appearing in the registry file is calculated based on the actual section size plus padding, and is aligned to the section align size (either the default or the one specified by the above directive). The align values for text and data as well as the padding values must be aligned to the minimum alignment size (64K). If not, *ld* generates a warning message and aligns these values to the minimum alignment.

`$start_address=`*addr*

This directive specifies where to start looking for addresses to put shared objects. The default start_address is 0x6000000.

`$data_after_text={ 1 | 0 }`

In this directive, a value of one instructs the linker to place data immediately after the text at specified text and data alignment requirements. A value of zero (the default) allows the linker to place these segments in different portions of the address space.

Shared Object Specification Lines

Shared object specification lines have the format:

so_name `[ :st = {.text | .data | $range}` *base_addr*`,`*padded_size* `: ] *`

where:

so_name	full path name (or trailing component) of a shared object
:st =	literal string indicating the beginning of the segment description
.text, .data	segment types: text or data
$range	range of addresses that can be used
base_addr	address where the segment starts
padded_size	padded size of the segment
:	literal string indicating the end of the segment description

A shared object specification can span several lines by “escaping” the newline character (using “\” as the last character on the line that is being continued). The following is an example of a shared object specification line:

```
libc.so.1 \
           :st = $range 0x5fc00000, 0x00400000:\
           :st = .text 0x5fe40000, 0x000a0000:\
           :st = .data 0x5fee0000, 0x00030000:
```

This specification instructs *ld* to relocate all segments of *libc.so.1* in the range 0x5fc00000 to 0x5fc00000+0x0040000, and, if possible, to place the text segment at 0x5fe40000 and the data segment at 0x5fee0000. The text segment should be padded to 0xa0000 bytes and the data segment to 0x3000 bytes. See */usr/lib/so_locations* for examples of shared object specifications.

When building a DSO with the *–check_registry* or *–update_registry* flag, if an entry corresponding to this DSO exists in the registry file, the linker tries to assign the indicated addresses for text and data. However, if the size of the DSO changes and no longer fits in the specified location, the linker searches for another location that fits. If the *$range* option is specified, the linker places the DSO only in the specified range of addresses. If there isn’t enough room, an error is returned.

Runtime Linking

This section explains the search path followed by the run-time linker and how you can cause symbols to be resolved at run time rather than link time.

Searching for DSOs at Runtime

When you run a dynamically linked executable, the runtime linker, *rld*, identifies the DSOs required by the executable, loads the required DSOs, and if necessary relocates DSOs within the process’s virtual address space, so that no two DSOs occupy the same location.The program header of a dynamically linked executable contains a field, the *liblist*, which lists the DSOs required by the executable.

When looking for a DSO, *rld* searches directories in the following sequence:

1. the path of the DSO in the *liblist* (if an explicit path is given)
2. RPATH if it's defined in the main executable
3. LD_LIBRARY_PATH if defined
4. the default path (*/usr/lib:/lib*)

RPATH is a colon-separated list of directories stored in the main executable. You can set RPATH by using the *–rpath* argument to *ld*:

```
ld -o myprog myprog.c -rpath /d/src/mylib libmylib.so -lc
```

This example links the program against *libmylib.so* in the current directory, and configures the executable such that *rld* searches the directory */d/src/mylib* when searching for DSOs.

The LD_LIBRARY_PATH environment variable is a colon-separated list of directories to search for DSOs. This can be very useful for testing new versions of DSOs before installing them in their final location. You can set the environment variable _RLD_ROOT to a colon-separated list of directories. The runtime linker prepends these to the paths in RPATH and the paths in the default search path.

In all of the colon-separated directory lists, an empty field is interpreted as the current directory. A leading or trailing colon counts as an empty field. Thus, if you set LD_LIBRARY_PATH to:

```
/d/src/lib1:/d/src/lib2:
```

the runtime linker searches the directory */d/src/lib1*, then the directory */d/src/lib2*, and then the current directory.

Note: For security reasons, if an executable has its set-user-ID or set-group-ID bits set, the runtime linker ignores the environment variables LD_LIBRARY_PATH and _RLD_ROOT. However, it still searches the directories in RPATH and the default path. ♦

Runtime Symbol Resolution

Dynamically linked executables can contain symbol references that aren't resolved before run time. Any symbol references in your main program or in an archive must be resolved at link time, unless you specify the *–ignore_unresolved* argument to *cc*. DSOs may contain references that aren't resolved at link time. All data symbols must be resolved at run time. If *rld* finds an unresolvable data symbol at run time, it will cause the executable to exit with an error. Text symbols are resolved only when they're used; so a program can run with unresolved text symbols, as long as the unresolved symbols aren't used.

You can force *rld* to resolve text symbols at run time by setting the environment variable LD_BIND_NOW. If unresolvable text symbols exist in your executable and LD_BIND_NOW is set, the executable will exit with an error, just as if there were unresolvable data symbols.

Compiling with *–Bsymbolic*

<Default Para Font**>**When you compile a DSO with *–Bsymbolic*, the dynamic linker resolves referenced symbols from itself first. If the shared object fails to supply the referenced symbol, then the dynamic linker searches the executable file and other shared objects. For example:

main—defines *x*
x.so—defines and uses *x*

If you compile *x.so* with *–Bsymbolic* on, the linker tries to resolve the use of *x* by looking first for the definition in *x.so* and then by looking in *main*.

In FORTRAN programs, the linker allocates space for **COMMON** symbols and the compiler allocates space for **BLOCK DATA**. The first kind of symbol (with **COMMON** blocks present) appears in the symbol table as **SHN_MIPS_ACOMMON** (uninitialized **DATA**) whereas the second kind of symbol (with **BLOCK DATA** present) appears as **SHN_DATA** (initialized **DATA**). In general, initialized data takes precedence when the dynamic linker tries to resolve a symbol. However, with *–Bsymbolic*, whatever is defined in the current object takes precedence, whether it is initialized or uninitialized.

Variables that are declared at file scope in C with *–cckr* are also treated this way. For example:

```
int foo[100];
```

is **COMMON** if *–cckr* is used and **DATA** if *–xansi* or *–ansi* is used.

For example:

In *main*:

```
COMMON i, j /* definition of i, j with initial values */
DATA i/1/, j/1/
call junk
end
```

In *x.so*:

```
COMMON i, j
/* definition of i, j with NO initial values */
/* initialized by kernel to all zeros */
print *, i, j
end
```

When you build *x.so* using *–Bsymbolic*, this program prints:

```
0 0
```

When you build *x.so* without *–Bsymbolic*, this program prints:

```
1
```

Converting Libraries to DSOs

When you link a program with a DSO, all of the symbols in the DSO become associated with the executable. This can cause unexpected results if archives that contain unresolved externals are converted to DSOs. When linking with a PIC archive, the linker links in only those object files that satisfy unresolved references. If an object file in an archive contains an unresolved external reference, the linker tries to resolve the reference only when that object file is linked in to your program. In contrast, a DSO containing an external data reference that cannot be resolved at run time causes the program to fail. Therefore, you should exercise caution when converting archives with external data references to DSOs.

For example, suppose you have an archive, *mylib.a*, and one of the object files in the archive, *has_extern.o*, references an external variable, *foo*. As long as your program doesn't reference any symbols in *has_extern.o*, the program will link and run properly. If your program references a symbol in *has_extern.o* and doesn't define *foo*, then the link will fail. However, if you convert *mylib.a* to a DSO, then any program that uses the DSO and doesn't define *foo* will fail at run time, regardless of whether the program references any symbols from *has_extern.o*.

Two possible solutions exist for this problem. The first consists of simply adding a "dummy" definition of the data to the DSO. A data definition appearing in the main executable preempts one appearing in the DSO itself. This may, however, be misleading for executables that use the portion of the DSO that needs the data, but that failed to define it in the main program. The second solution consists of separating the routines that use the data definition into a second DSO, and placing dummy functions for them in the first DSO.

The second DSO can then be dynamically loaded the first time any of the dummy functions is accessed. Each of the dummy functions must verify that the second DSO was loaded before calling the real function (which must have a unique name). This way, programs run whether or not they supply the missing external data, as long as they don't call any of the functions that require the data. The first time one of the dummy functions is called, it tries to dynamically load the second DSO. Programs that do not supply the missing data fail at this point. For more information on dynamic loading, see "Dynamic Loading Under Program Control" below.

Dynamic Loading Under Program Control

IRIX provides a library interface to the runtime linker that allows programs to dynamically load and unload DSOs. This interface is called *libdl*, and it consists of four functions listed in Table 3-1.

Table 3-1 *libdl* functions

dlopen()	load a DSO
dlsym()	find a symbol in a loaded DSO
dlclose()	unload a DSO
dlerror()	report errors

To load a DSO, call **dlopen()**:

```
include <dlfcn.h>
void *dlhandle;
...
dlhandle = dlopen("/usr/lib/mylib.so", RTLD_LAZY);
if (dlhandle == NULL) {
/* couldn't open DSO */
printf("Error: %s\n", dlerror());
}
```

The first argument to **dlopen()** is the pathname of the DSO to be loaded. This may be either an absolute or a relative pathname. When you call this routine, the runtime linker tries to load the specified DSO. If any unresolved references exist in the executable that are defined in the DSO, the runtime linker resolves these references on demand. You can also use **dlsym()** to access symbols in the DSO, whether or not the symbols are referenced in your executable.

When a DSO is brought into the address space of a process, it may contain references to symbols whose addresses are not known until the object is loaded. These references must be relocated before the symbols can be accessed. The second argument to **dlopen()** governs when these relocations take place.

This argument can have the following values:

RTLD_LAZY	Under this mode, only references to data symbols are relocated when the object is loaded. References to functions are not relocated until a given function is invoked for the first time. This mode should result in better performance, since a process may not reference all of the functions in any given shared object.
RTLD_NOW	Under this mode, all necessary relocations are performed when the object is first loaded. This may result in some wasted effort if relocations are performed for functions that are never referenced. However, this option is useful for applications that need to know as soon as an object is loaded that all symbols referenced during execution will be available.

To access symbols that are not referenced in your program, use **dlsym()**:

```
#include <dlfcn.h>
void *dlhandle;
int (*funcptr)(int);
int i,j;
... load DSO ...
funcptr = (int (*)(int)) dlsym(dlhandle, "factorial");
if (funcptr == NULL) {
/* couldn't locate the symbol */
}
i = (*funcptr)(j);
```

In this example, we look up the address of a function called **factorial()** and assign it to the function pointer **funcptr**.

If you encounter an error (**dlopen()** or **dlsym()** returns NULL), you can get diagnostic information by calling **dlerror().** The **dlerror()** function returns a string describing the cause of the latest error. You should only call **dlerror()** after an error has occurred; at other times, its return value is undefined.

To unload a DSO, call **dlclose()**:

```
#include <dlfcn.h>
void *dlhandle;
... load DSO, use DSO symbols ...
dlclose(dlhandle);
```

The **dlclose** function frees up the virtual address space **mmap**ed by the **dlopen** call of that file (similar to a **munmap** call). The difference, however, is that **dlclose** on a file that has been opened multiple times (either through **dlopen** or program startup) does not cause the file to be **munmap**ed until the file is no longer needed by the process.

Versioning of DSOs

In the 5.0.1 release, a mechanism for the versioning of shared objects was introduced for SGI-specific shared objects and executables. Note that this mechanism is outside the scope of the ABI, and, thus, must not be relied on for code that must be ABI-compliant and run on non-SGI platforms. Currently, all executables produced on SGI systems are marked SGI_ONLY to allow use of the versioning mechanism.

Versioning is mainly of interest to developers of shared objects. It may not be of interest to you if you simply *use* shared objects. Versioning allows a developer to update a shared object in a way that may be incompatible with executables previously linked against the shared object. This is accomplished by renaming the original shared object and providing it along with the (incompatible) new version.

What Is a Version?

A version is part or all of an identifying *version_string* that can be associated with a shared object by using the *–set_version version_string* option to *ld*(1) when the shared object is created.

A *version_string* consists of one or more versions separated by colons (:). A single version has the form:

[*comment*#]**sgi***major*.*minor*

where:

comment is a comment string, which is ignored by the versioning mechanism. It consists of any sequence of characters followed by a pound sign (#). The comment is optional.

sgi	is the literal string *sgi*.
major	is the major version number, which is a string of digits [0-9].
.	is a literal period.
minor	is the minor version number, which is a string of digits [0-9].

Follow these instructions when building your shared library:

When you first build your shared library, give it an initial version, for example, *sgi1.0*. Add the option *–set_version sgi1.0* to the command to build your shared library (*cc –shared*, *ld –shared*).

Whenever you make a *compatible* change to the shared object, create another version by changing the minor version number (for example, *sgi1.1*) and add it to the end of the *version_string*. The command to set the version of the shared library now looks like *–set_version "sgi1.0:sgi1.1"*.

When you make an *incompatible* change to the shared object:

1. Change the file name of the old shared object by adding a dot followed by the major number of one of the versions to the file name of the shared object. Do not change the *soname* of the shared object or its contents. Simply rename the file.
2. Update the major version number and set the *version_string* of the shared object (when you create it) to this new version; for example, *–set_version sgi2.0*.

This versioning mechanism affects executables in the following ways:

- When an executable is linked against a shared object, the last version in the shared object's *version_string* is recorded in the executable as part of the *liblist*. You can examine this using *elfdump –Dl*.
- When you run an executable, *rld* looks for the proper file name in its usual search routine.
- If a file is found with the correct name, the version specified in the executable for this shared object is compared to each of the versions in the *version_string* in the shared object. If one of the versions in the *version_string* matches the executable's version exactly (ignoring comments), then that library is used.

- If no proper match is found, a new file name for the shared object is built by combining the *soname* specified in the executable for this shared object and the *major* number found in the version specified in the executable for this shared object (*soname.major*). Remember that you did *not* change the *soname* of the object, only the file name. The new file is searched for using *rld*'s usual search procedure.

For example, suppose you have a shared object *foo.so* with initial version *sgi10.0*. Over time, you make two compatible changes for *foo.so* that result in the following final *version_string* for *foo.so*:

```
initial_version#sgi10.0:upgrade#sgi10.1:new_devices#sgi10.2
```

You then link an executable that uses this shared object, *useoldfoo*. This executable specifies version *sgi10.2* for *soname foo.so*. (Remember that the executable inherits the last version in the *version_string* of the shared object.)

The time comes to upgrade *foo.so* in an incompatible way. Note that the *major* version of *foo.so* is 10, so you move the existing *foo.so* to the file name *foo.so.10* and create a new *foo.so* with the *version_string*:

```
efficient_interfaces#sgi11.0
```

New executables linked with *foo.so* use it directly. Older executables, like *useoldfoo*, attempt to use *foo.so*, but find that its version (*sgi11.0*) is not the version they need (*sgi10.2*). They then attempt to find a *foo.so* in the file name *foo.so.10* with version *sgi10.2*.