Versioning 5

Overview

ELF objects processed by the link-editors provide many global symbols to which other objects can bind. These symbols describe the object's *application binary interface* (ABI). During the evolution of an object this interface can change due to the addition or deletion of global symbols. In addition, the objects evolution can involve internal implementation changes.

Versioning refers to several techniques that can be applied to an object to indicate interface and implementation changes. These techniques provide for the objects controlled evolution while maintaining backward compatibility.

This chapter describes how an object's ABI can be defined, classifies how changes to this interface can affect backward compatibility, and presents models by which interface and implementation changes can be incorporated into new releases of the object.

The focus of this chapter is the runtime interfaces of dynamic executables and shared objects. The techniques used to describe and manage changes within these dynamic objects are presented in generic terms. A common set of naming conventions and versioning scenarios, as applied to shared objects, can be found in Appendix B, "Versioning Quick Reference.

It is important that developers of dynamic objects be aware of the ramifications of an interface change, and understand how such changes can be managed, especially in regards to maintaining backward compatibility with previously shipped objects.

5

The global symbols made available by any dynamic object represent the object's *public* interface. Frequently, the number of global symbols remaining in an object at the end of a link-edit are more than you would like to make public. These global symbols derive from the interrelationships required between the relocatable objects used to build the object, and represent *private* interfaces within the object itself.

A precursor to defining an object's binary interface is to first define only those global symbols you wish to make publicly available from the object being created. These public symbols can be established using the link-editor's `-M` option and an associated `mapfile` as part of the final link-edit. This technique is introduced in "Reducing Symbol Scope" on page 38. This public interface establishes one or more *version definitions* within the object being created, and forms the foundation for the addition of new interfaces as the object evolves.

The following sections build upon this initial public interface. First though, it is useful to understand how various changes to an interface can be categorized so that they can be managed appropriately.

Interface Compatibility

There are many types of change that can be made to an object. In their simplest terms these changes can be categorized into one of two groups:

- *compatible* updates. These updates are *additive*, in that all previously available interfaces remain intact.
- *incompatible* updates. These updates have changed the existing interface in such a way that existing users of the interface can fail or behave incorrectly.

The following list attempts to clarify some common object changes into one of the above categorizations:

- the addition of a symbol - a *compatible* update.
- the removal of a symbol - an *incompatible* update.
- the addition of an argument to a non-`varargs(5)` function - an *incompatible* change.
- the removal of an argument from a function - an *incompatible* update.
- the change of size, or content, of a data item to a function or as an external definition - an *incompatible* change.

- a bug fix, or internal enhancement to a function - a *compatible* change providing the semantic properties of the object remain unchanged, otherwise, this is an *incompatible* change.

Note – It is possible, because of interposition, that the addition of a symbol can constitute an *incompatible* update, such that the new symbol might conflict with an applications use of that symbol. However, this does seem rare in practice as source level name space management is commonly used.

Compatible updates can be accommodated by maintaining version definitions *internal* to the object being generated. Incompatible updates can be accommodated by producing a new object with a new *external* versioned name. Both of these versioning techniques allow for the selective binding of applications as well as verification of correct version binding at runtime. These two techniques are explored in more detail in the following sections.

Internal Versioning

A dynamic object can have associated with it one or more internal *version definitions*. Each version definition is commonly associated with one or more symbol names. A symbol name can only be associated with *one* version definition, however a version definition can inherit the symbols from other version definitions. Thus, a structure exists to define one or more independent, or related, version definitions within the object being created. As new changes are made to the object, new version definitions can be added to express these changes.

There are two consequences of providing version definitions within a shared object:

- Dynamic objects that are built against this shared object can record their dependency on the version definitions they bind to. These version dependencies will be verified at runtime to insure that the appropriate interfaces, or functionality, are available for the correct execution of an application.
- Dynamic objects can select only those version definitions of a shared object that they wish to bind to during their link-edit. This mechanism allows developers to control their dependency on a shared object to the interfaces, or functionality, that provide the most flexibility.

Creating a Version Definition

Version definitions commonly consist of an association of symbol names to a unique version *name*. These associations are established within a `mapfile` and supplied to the final link-edit of an object using the link-editor's `-M` option (this technique was introduced in the section "Reducing Symbol Scope" on page 38).

A version definition is established whenever a version name is specified as part of the `mapfile` directive. In the following example two source files are combined, together with `mapfile` directives, to produce an object with a defined public interface:

```
$ cat foo.c
extern  const char * _foo1;

void foo1()
{
        (void) printf(_foo1);
}

$ cat data.c
const char * _foo1 = "string used by foo1()\n";

$ cat mapfile
SUNW_1.1 {                  # Release X
        global:
                foo1;
        local:
                *;
};
$ cc -o libfoo.so.1 -M mapfile -G foo.o data.o
$ nm -x libfoo.so.1 | grep "foo.$"
[33]    |0x0001058c|0x00000004|OBJT |LOCL |0x0  |17   |_foo1
[35]    |0x00000454|0x00000034|FUNC |GLOB |0x0  |9    |foo1
```

Here, the symbol `foo1` is the *only* global symbol defined to provide the shared object's public interface. The special *auto-reduction* directive "*" causes the reduction of all other global symbols to have local binding within the object being generated (this directive is introduced in "Defining Additional Symbols"

on page 32). The associated version name, `SUNW_1.1`, causes the generation of a version definition. Thus, the shared object's public interface consists of the internal version definition `SUNW_1.1`, associated with the global symbol `foo1`.

Whenever a version definition, or the *auto-reduction* directive, are used to generate an object, a *base version* definition is also created. This base version is defined using the name of the file itself, and is used to associate any reserved symbols generated by the link-editor (see "Generating the Output Image" on page 42 for a list of these reserved symbols).

The version definitions contained within an object can be displayed using `pvs(1)` with the `-d` option:

```
$ pvs -d libfoo.so.1
        libfoo.so.1;
        SUNW_1.1;
```

Here, the object `libfoo.so.1` has an internal version definition named `SUNW_1.1`, together with a base version definition `libfoo.so.1`.

Note – The link-editor's `-z noversion` option allows `mapfile` directed symbol reduction to be performed but suppresses the creation of version definitions.

Starting with this initial version definition, it is possible for the object to evolve by adding new interfaces and updated functionality. For example, a new function, `foo2`, together with its supporting data structures, can be added to the object by updating the source files `foo.c` and `data.c`:

```
$ cat foo.c
extern  const char * _foo1;
extern  const char * _foo2;

void foo1()
{
        (void) printf(_foo1);
}

void foo2()
{
        (void) printf(_foo2);
}

$ cat data.c
const char * _foo1 = "string used by foo1()\n";
const char * _foo2 = "string used by foo2()\n";
```

A new version definition, `SUNW_1.2`, can be created to define a new interface representing the symbol `foo2`. In addition, this new interface can be defined to inherit the original version definition `SUNW_1.1`.

The creation of this new interface is important as it identifies the evolution of the object and enables users to verify and select the interfaces to which they bind. These concepts are covered in more detail in "Binding to a Version Definition" on page 126 and in "Specifying a Version Binding" on page 132.

The following example shows the `mapfile` directives that create these two interfaces:

```
$ cat mapfile
SUNW_1.1 {                      # Release X
        global:
                foo1;
        local:
                *;
};

SUNW_1.2 {                      # Release X+1
        global:
                foo2;
} SUNW_1.1;

$ cc -o libfoo.so.1 -M mapfile -G foo.o data.o
$ nm -x libfoo.so.1 | grep "foo.$"
[33]    |0x00010644|0x00000004|OBJT |LOCL |0x0  |17   |_foo1
[34]    |0x00010648|0x00000004|OBJT |LOCL |0x0  |17   |_foo2
[36]    |0x000004bc|0x00000034|FUNC |GLOB |0x0  |9    |foo1
[37]    |0x000004f0|0x00000034|FUNC |GLOB |0x0  |9    |foo2
```

Here, the symbols `foo1` and `foo2` are both defined to be part of the shared object's public interface. However, each of these symbols is assigned to a different version definition; `foo1` is assigned to `SUNW_1.1`, and `foo2` is assigned to `SUNW_1.2`.

These version definitions, their inheritance, and their symbol association can be displayed using `pvs(1)` together with the `-d`, `-v` and `-s` options:

```
$ pvs -dsv libfoo.so.1
        libfoo.so.1:
                _end;
                _GLOBAL_OFFSET_TABLE_;
                _DYNAMIC;
                _edata;
                _PROCEDURE_LINKAGE_TABLE_;
                _etext;
        SUNW_1.1:
                foo1;
                SUNW_1.1;
        SUNW_1.2:               {SUNW_1.1}:
                foo2;
                SUNW_1.2
```

Here, the version definition `SUNW_1.2` has a dependency on the version definition `SUNW_1.1`.

The inheritance of one version definition by another is a useful technique that reduces the version information that will eventually be recorded by any object that binds to a version dependency. Version inheritance is covered in more detail in the section "Binding to a Version Definition" on page 126.

Any internal version definition will have an associated *version definition symbol* created. As shown in the previous `pvs(1)` example, these symbols are displayed when using the `-v` option. The use of these symbols will be covered in the section "Binding to a Weak Version Definition" on page 130.

Creating a Weak Version Definition

Internal changes to an object that do not require the introduction of a new interface definition, can be defined by creating a *weak* version definition. Examples of such changes are bug fixes or performance improvements.

Such a version definition is empty, in that it has no global interface symbols associated with it.

For example, if the data file `data.c`, used in the previous examples, is updated to provide more detailed string definitions:

```
$ cat data.c
const char * _foo1 = "string used by function foo1()\n";
const char * _foo2 = "string used by function foo2()\n";
```

then a weak version definition can be introduced to identify this change:

```
$ cat mapfile
SUNW_1.1 {                      # Release X
        global:
                foo1;
        local:
                *;
};

SUNW_1.2 {                      # Release X+1
        global:
                foo2;
} SUNW_1.1;

SUNW_1.2.1 { } SUNW_1.2;        # Release X+2

$ cc -o libfoo.so.1 -M mapfile -G foo.o data.o
$ pvs -dv libfoo.so.1
        libfoo.so.1;
        SUNW_1.1;
        SUNW_1.2:                 {SUNW_1.1};
        SUNW_1.2.1 [WEAK]:        {SUNW_1.2};
```

Here, the empty version definition is signified by the *weak* label. These *weak* version definitions allow applications to verify the existence of a particular implementation by binding to the version definition associated with that functionality. The section, "Binding to a Version Definition" on page 126, illustrates how these definitions can be used in more detail.

Defining Unrelated Interfaces

The previous examples have shown how new version definitions added to an object have inherited any existing version definitions. It is also possible to create version definitions that are unique and independent. In the following example, two new files, `bar1.c` and `bar2.c`, are added to the object `libfoo.so.1`. These files contribute two new symbols, `bar1` and `bar2`, respectively:

```
$ cat bar1.c
extern  void foo1();

void bar1()
{
        foo1();
}
$ cat bar2.c
extern  void foo2();

void bar2()
{
        foo2();
}
```

These two symbols are intended to define two new public interfaces. Neither of these new interfaces are related to each other, however each expresses a dependency on the original `SUNW_1.2` interface.

The following `mapfile` definition creates this required association:

```
$ cat mapfile
SUNW_1.1 {                      # Release X
        global:
                foo1;
        local:
                *;
};

SUNW_1.2 {                      # Release X+1
        global:
                foo2;
} SUNW_1.1;

SUNW_1.2.1 { } SUNW_1.2;        # Release X+2

SUNW_1.3a {                     # Release X+3
        global:
                bar1;
} SUNW_1.2;

SUNW_1.3b {                     # Release X+3
        global:
                bar2;
} SUNW_1.2;
```

Again, the version definitions created in `libfoo.so.1` using this `mapfile`, and their related dependencies, can be inspected using `pvs(1)`:

```
$ cc -o libfoo.so.1 -M mapfile -G foo.o bar.o data.o
$ pvs -dv libfoo.so.1
        libfoo.so.1;
        SUNW_1.1;
        SUNW_1.2:                   {SUNW_1.1};
        SUNW_1.2.1 [WEAK]:          {SUNW_1.2};
        SUNW_1.3a:                  {SUNW_1.2};
        SUNW_1.3b:                  {SUNW_1.2};
```

The following sections explore how these version definition recordings can be used to verify runtime binding requirements and control the binding requirements of an object during its creation.

Binding to a Version Definition

When a dynamic executable or shared object is built against other shared objects, these dependencies are recorded in the resulting object (see "Shared Object Processing" on page 13 and "Recording a Shared Object Name" on page 86 for more details). If these shared object dependencies also contain version definitions then an associated *version dependency* will be recorded in the resulting object.

The following example takes the data files from the previous section and generates a shared object suitable for a compile time environment. This shared object, `libfoo.so.1`, will be used in following binding examples:

```
$ cc -o libfoo.so.1 -h libfoo.so.1 -M mapfile -G foo.o bar.o \
data.o
$ ln -s libfoo.so.1 libfoo.so
$ pvs -dsv libfoo.so.1
        libfoo.so.1:
                _end;
                _GLOBAL_OFFSET_TABLE_;
                _DYNAMIC;
                _edata;
                _PROCEDURE_LINKAGE_TABLE_;
                _etext;
        SUNW_1.1:
                foo1;
                SUNW_1.1;
        SUNW_1.2:                {SUNW_1.1}:
                foo2;
                SUNW_1.2;
        SUNW_1.2.1 [WEAK]:       {SUNW_1.2}:
                SUNW_1.2.1;
        SUNW_1.3a:               {SUNW_1.2}:
                bar1;
                SUNW_1.3a;
        SUNW_1.3b:               {SUNW_1.2}:
                bar2;
                SUNW_1.3b
```

In effect, there are six public interfaces being offered by this shared object. Four of these interfaces (`SUNW_1.1`, `SUNW_1.2`, `SUNW_1.3a` and `SUNW_1.3b`) define a set of functions, one interface (`SUNW_1.2.1`) describes an internal

implementation change to the shared object, and one interface (`libfoo.so.1`) defines several reserved labels. Dynamic objects that build with this object will record which of these interfaces they bind to.

The following example builds an application that references both symbols `foo1` and `foo2`. The versioning dependency information recorded in the application can be examined using `pvs(1)` with the `-r` option:

```
$ cat prog.c
extern void foo1();
extern void foo2();

main()
{
        foo1();
        foo2();
}
$ cc -o prog prog.c -L. -R. -lfoo
$ pvs -r prog
        libfoo.so.1 (SUNW_1.2, SUNW_1.2.1);
```

In this example, the application `prog` has really bound to the two interfaces `SUNW_1.1` and `SUNW_1.2`, as these interfaces have provided the global symbols `foo1` and `foo2` respectively.

However, since version definition `SUNW_1.1` is defined within `libfoo.so.1` as being inherited by the version definition `SUNW_1.2`, it is only necessary to record the latter version dependency. This *normalization* of version definition dependencies reduces the amount of version information that must be maintained within an object and processed at runtime.

Since the application `prog` was built against the shared object's implementation containing the *weak* version definition `SUNW_1.2.1`, this dependency is also recorded. Even though this version definition is defined to inherit the version definition `SUNW_1.2`, the version's weak nature precludes its normalization with `SUNW_1.1`, and results in a separate dependency recording.

Had there been multiple weak version definitions that inherit from each other then these definitions will be normalized in the same manner as non-weak version definitions are.

Note – The recording of a version dependency can be suppressed by the link-editor's `-z noversion` option.

Having recorded these version definition dependencies, the runtime linker validates the existence of the required version definitions in the objects bound to when the application is executed. This validation can be displayed using `ldd(1)` with the `-v` option. For example, by running `ldd(1)` on the application `prog`, the version definition dependencies are shown to be found correctly in the shared object `libfoo.so.1`:

```
$ ldd -v prog

   find library=libfoo.so.1; required by prog
        libfoo.so.1 =>   ./libfoo.so.1
   find version=libfoo.so.1;
        libfoo.so.1 (SUNW_1.2) =>            ./libfoo.so.1
        libfoo.so.1 (SUNW_1.2.1) =>          ./libfoo.so.1
   ....
```

Note – `ldd(1)` with the `-v` option implies *verbose* output, in that a recursive list of all dependencies, together with all versioning requirements, will be generated.

If a non-weak version definition dependency cannot be found, a fatal error will occur during application initialization. Any weak version definition dependency that cannot be found is silently ignored. For example, if the application `prog` was run in an environment in which `libfoo.so.1` only contained the version definition `SUNW_1.1`, then the following fatal error will occur:

```
$ pvs -dv libfoo.so.1
        libfoo.so.1;
        SUNW_1.1;
$ prog
ld.so.1: prog: fatal: libfoo.so.1: version 'SUNW_1.2' not \
found (required by file prog)
```

Had the application `prog` not recorded any version definition dependencies, the nonexistence of the required interface symbol `foo2` will have manifested itself sometime during the execution of the application as a fatal relocation error (see "Relocation Errors" on page 61). This relocation error might occur at process initialization, during process execution, or might not occur at all if the execution path of the application did not call the function `foo2`.

Recording version definition dependencies provides an alternative, and immediate indication of the availability of the interfaces required by the application.

If the application `prog` was run in an environment in which `libfoo.so.1` only contained the version definitions `SUNW_1.1` and `SUNW_1.2`, then all non-weak version definition requirements will be satisfied. The absence of the weak version definition `SUNW_1.2.1` is deemed nonfatal, and so no runtime error condition will be generated. However, `ldd(1)` can be used to display *all* version definitions that cannot be found:

```
$ pvs -dv libfoo.so.1
        libfoo.so.1;
        SUNW_1.1;
        SUNW_1.2:                {SUNW_1.1};
$ prog
string used by foo1()
string used by foo2()
$ ldd prog
        libfoo.so.1 =>   ./libfoo.so.1
        libfoo.so.1 (SUNW_1.2.1) =>          (version not found)
        ...........
```

Note – If an object requires a version definition from a given dependency, and at runtime an implementation of that dependency is found that contains no version definition information, the version verification of the dependency will be silently ignored. This policy provides a level of backward compatibility as the transition from non-versioned to versioned shared objects is taken. `ldd(1)` however, can still be used to display any version requirement discrepancies.

Binding to a Weak Version Definition

Recall that weak version definitions are used to mark an internal implementation change, and are well suited to indicating bug fixes and performance improvements made to an object. If you require the existence of a *weak* version definition for the correct execution of an application, then an explicit dependency on this version definition can be generated.

Establishing such a dependency can be important when a bug fix, or performance improvement, become critical for the application to function correctly.

Each version definition maintained within an object has an absolute *version definition symbol* associated with it. For example, the shared object `libfoo.so.1` containing the version definitions shown in previous examples has the following version definition symbols:

```
$ pvs -dsv libfoo.so.1 | fgrep SUNW_1
        SUNW_1.1:
                SUNW_1.1;
        SUNW_1.2:                   {SUNW_1.1}:
                SUNW_1.2;
        SUNW_1.2.1 [WEAK]:          {SUNW_1.2}:
                SUNW_1.2.1;
        SUNW_1.3a:                  {SUNW_1.2}:
                SUNW_1.3a;
        SUNW_1.3b:                  {SUNW_1.2}:
                SUNW_1.3b;
```

By making explicit reference to a version definition's symbol, an explicit dependency on that version definition is created. This explicit reference also causes the version definition to be *promoted* from a weak to a strong dependency.

Therefore, the application `prog` can be built to enforce the requirement that the `SUNW_1.2.1` interface be made available at runtime. A reference to the version definition can be generated using the link-editor's `-u` option:

```
$ cat prog
extern void foo1();
extern void foo2();

main()
{
        foo1();
        foo2();
}
$ cc -u SUNW_1.2.1 -o prog prog.c -L. -R. -lfoo
$ pvs -r prog
        libfoo.so.1 (SUNW_1.2.1);
```

Here, `prog` has been built with an explicit dependency on the interfaces `SUNW_1.1`, `SUNW_1.2`, and `SUNW_1.2.1`. Because the version definition `SUNW_1.2.1` is promoted to a strong version, it is also normalized with its dependency `SUNW_1.2`. At runtime, if the version definition `SUNW_1.2.1` cannot be found, a fatal error will be generated.

Verifying Versions in Additional Objects

Version definition symbols also provide a mechanism for verifying the version requirements of an object obtained by `dlopen(3X)`. Any object added to the process's address space using this function will have no automatic version dependency verification carried out by the runtime linker. Thus, it is the responsibility of the caller of this function to verify that any versioning requirements are met.

The presence of a required version definition can be verified by looking up the associated version definition symbol using `dlsym(3X)`. The following example shows the shared object `libfoo.so.1` being added to a process by `dlopen(3X)` and verified to insure that the interface `SUNW_1.2` is available:

```
#include        <stdio.h>
#include        <dlfcn.h>

main()
{
    void *       handle;
    const char * file = "libfoo.so.1";
    const char * vers = "SUNW_1.2";
    ....

    if ((handle = dlopen(file, RTLD_LAZY)) == NULL) {
            (void) printf("dlopen: %s\n", dlerror());
            exit (1);
    }

    if (dlsym(handle, vers) == NULL) {
            (void) printf("fatal: %s: version `%s' not found\n",
                 file, vers);
            exit (1);
    }
    ....
```

Specifying a Version Binding

When building a dynamic object against a shared object containing version definitions, it is possible to instruct the link-editor to limit the binding to specific version definitions. Effectively, the link-editor allows you to control an object's binding to specific interfaces.

An object's binding requirements can be controlled using a *file control directive*. This directive is supplied using the link-editor's `-M` option and an associated `mapfile`. The syntax for these version control `mapfile` directives is shown below:

```
name - version [ version ... ] ;
```

- *name* - represents the name of the shared object dependency. This name should match the shared object's compilation environment name as used by the link-editor (see "Library Naming Conventions" on page 14).
- *version* - represents the version definition name within the shared object that should be made available for binding. Multiple version definitions can be specified.

There are a couple of scenarios where this binding control can be useful:

- If a shared object has been versioned to define unique and independent versions, possibly defining different standards interfaces, then the application can insure that its bindings meet the requirements of a specific interface.
- If a shared object has been versioned over several software releases, application developers can restrict themselves to the interfaces that were available in a previous software release. Thus, an application can be built using the latest release of the shared object in the knowledge that the application's interface requirements can be met by a previous release of the shared object.

The following is an example of using the version control mechanism. This example continues to use the shared object `libfoo.so.1` containing the following version interface definitions:

```
$ pvs -ds libfoo.so.1
        libfoo.so.1:
                _end;
                _GLOBAL_OFFSET_TABLE_;
                _DYNAMIC;
                _edata;
                _PROCEDURE_LINKAGE_TABLE_;
                _etext;
        SUNW_1.1:
                foo1;
        SUNW_1.2:
                foo2;
```

The version definitions SUNW_1.1 and SUNW_1.2 represent interfaces within libfoo.so.1 that were made available in software Release X and Release X+1 respectively. An application can be built to bind only to the interfaces available in Release X by using the following version control mapfile directive:

```
$ cat mapfile
libfoo.so - SUNW_1.1;
```

For example, if you develope an application, prog, and wish to insure that the application will run on Release X, then the application can only use the interfaces available in that release. If the application mistakenly references the symbol foo2, then the application's noncompliance to the required interface will be signalled by the link-editor as an undefined symbol error:

```
$ cat prog.c
extern void foo1();
extern void foo2();

main()
{
        foo1();
        foo2();
}
$ cc -o prog prog.c -M mapfile -L. -R. -lfoo
Undefined                       first referenced
 symbol                             in file
foo2                                prog.o  (symbol belongs to \
unavailable version ./libfoo.so (SUNW_1.2))
ld: fatal: Symbol referencing errors. No output written to prog
```

To be compliant with the SUNW_1.1 interface, you must remove the *reference* to foo2. This can be achieved either by reworking the application to remove the requirement on foo2, or by adding an implementation of foo2 to the build of the application.

Relocatable Objects

The preceding sections have described how version information can be recorded and used within dynamic objects. Relocatable objects can maintain versioning information in a similar manner, however there are one or two subtle differences in how this information is used.

Any version definitions supplied to the link-edit of a relocatable object are recorded in exactly the same format as has been described in previous examples. However, by default, symbol reduction is not carried out on the object being created. Instead, when the relocatable object is finally used as input to the generation of a dynamic object, the version recording itself will be used to determine the symbol reductions to apply.

In addition, any version definitions found in relocatable objects will be propagated to the dynamic object. For an example of version processing in relocatable objects see “Reducing Symbol Scope” on page 38.

External Versioning

Runtime references to a shared object should always refer to the files *version* filename. Commonly this is expressed as a filename suffixed with a version number. When a shared object’s interface changes in an *incompatible* manner - such that it will break old applications - a new shared object should be distributed with a new versioned filename. In addition, the original versioned filename must still be distributed to provide the interfaces required by the old applications.

By providing shared objects as separate versioned filenames within the runtime environment, applications built over a series of software releases can be guaranteed that the interface against which they were built is available for them to bind during their execution.

The following section describes how to coordinate the binding of an interface between the compilation and runtime environments.

Coordination of Versioned Filenames

In the section "Naming Conventions" on page 84 it was stated that during a link-edit the most common method to input shared objects was to use the `-l` option. This option will use the link-editor's library search mechanism to locate shared objects that are prefixed with `lib` and suffixed with `.so`.

However, at runtime any shared object dependencies should exist in their *versioned* name form. Instead of maintaining two distinct shared objects that follow these naming conventions, the most common mechanism of coordinating these objects involves creating file system links between the two filenames.

To make the runtime shared object `libfoo.so.1` available to the compilation environment it is necessary to provide a symbolic link from the compilation filename to the runtime filename. For example:

```
$ cc -o libfoo.so.1 -G -K pic foo.c
$ ln -s libfoo.so.1 libfoo.so
$ ls -l libfoo*
lrwxrwxrwx  1 usr grp          11 1991 libfoo.so -> libfoo.so.1
-rwxrwxr-x  1 usr grp        3136 1991 libfoo.so.1
```

Note – Either a symbolic *or* hard link can be used. However, as a documentation and diagnostic aid, symbolic links are more useful.

Here, the shared object `libfoo.so.1` has been generated for the runtime environment. Generating a symbolic link `libfoo.so`, has also enabled this file's use in a compilation environment. For example:

```
$ cc -o prog main.o -L. -lfoo
```

Here the link-editor will process the relocatable object `main.o` with the interface described by the shared object `libfoo.so.1` which it will find by following the symbolic link `libfoo.so`.

If over a series of software releases, new versions of this shared object are distributed with changed interfaces, the compilation environment can be constructed to use the interface that is applicable by changing the symbolic link. For example:

```
$ ls -l libfoo*
lrwxrwxrwx  1 usr grp          11 1993 libfoo.so -> libfoo.so.3
-rwxrwxr-x  1 usr grp        3136 1991 libfoo.so.1
-rwxrwxr-x  1 usr grp        3237 1992 libfoo.so.2
-rwxrwxr-x  1 usr grp        3554 1993 libfoo.so.3
```

Here, three major versions of the shared object are available. Two of these shared objects, `libfoo.so.1` and `libfoo.so.2`, provide the dependencies for existing applications. `libfoo.so.3` offers the latest major release for building and running new applications.

Using this symbolic link mechanism itself is insufficient to coordinate the correct binding of a shared object from its use in the compilation environment to its requirement in the runtime environment. As the example presently stands, the link-editor will record in the dynamic executable `prog` the *filename* of the shared object it has processed, which in this case will be the compilation environment filename:

```
$ dump -Lv prog

prog:
 **** DYNAMIC SECTION INFORMATION ****
.dynamic:
[INDEX] Tag      Value
[1]     NEEDED   libfoo.so
.........
```

This means that when the application `prog` is executed, the runtime linker will search for the dependency `libfoo.so`, and consequently this will bind to whichever file this symbolic link is pointing.

To provide the correct runtime name to be recorded as a dependency, the shared object `libfoo.so.1` should be built with an *soname* definition. This definition identifies the shared objects runtime name, and is used as the

dependency name by any object that links against this shared object. This definition can be provided using the `-h` option during the link-edit of the shared object itself. For example:

```
$ cc -o libfoo.so.1 -G -K pic -h libfoo.so.1 foo.c
$ ln -s libfoo.so.1 libfoo.so
$ cc -o prog main.o -L. -lfoo
$ dump -Lv prog

prog:
 **** DYNAMIC SECTION INFORMATION ****
.dynamic:
[INDEX] Tag      Value
[1]     NEEDED   libfoo.so.1
.........
```

This symbolic link and the *soname* mechanism has established a robust coordination between the shared object naming conventions of the compilation and runtime environments, one in which the interface processed during the link-edit is accurately recorded in the output file generated. This recording ensures that the intended interface will be furnished at runtime.