Versioning Quick Reference B

ELF objects make available global symbols to which other objects can bind. Some of these global symbols can be identified as providing the object's *public interface*. Other symbols are part of the objects internal implementation and are not intended for external use. An objects interface can evolve from one software release to another, and thus the ability to identify this evolution is desirable.

In addition, identifying the *internal implementation* changes of an object from one software release to another might be desirable.

Both interface and implementation identifications can be recorded within an object by establishing internal *version definitions* (see "Overview" on page 115 for a more complete introduction to the concept of internal versioning).

Shared objects are prime candidates for internal versioning as this technique defines their evolution, provides for interface validation during runtime processing (see "Binding to a Version Definition" on page 126), and provides for the selective binding of applications (see "Specifying a Version Binding" on page 132). Shared objects will be used as the examples throughout this chapter.

The following sections provide a simple overview, or *cheat sheet*, of the internal versioning mechanism provided by the link-editors as applied to shared objects. The examples recommend conventions and mechanisms for versioning shared objects, from their initial construction through several common update scenarios.

B

Naming Conventions

A shared object follows a naming convention that includes a *major* number file suffix (see "Naming Conventions" on page 84). Within this shared object, one or more *version definitions* can be created. Each version definition corresponds to one of the following categories:

- It defines an industry *standard* interface (for example, the *System V Application Binary Interface*).
- It defines a vendor specific *public* interface.
- It defines a vendor specific *private* interface.
- It defines a vendor specific change to the internal implementation of the object.

The following *version definition* naming conventions help indicate which of these categories the definition represents.

The first three of these categories indicate interface definitions. These definitions consist of an association of the global symbol names that comprise the interface, with a version definition name (see "Creating a Version Definition" on page 118). Interface changes within a shared object are often referred to as *minor* revisions. Therefore, version definitions of this type, are suffixed with a *minor* version number which is based off of the filenames *major* version number suffix.

The last category indicates a change having occurred within the object. This definition consists of a version definition acting as a label and has no symbol names associated with it. This definition is referred to as being a *weak* version definition (see "Creating a Weak Version Definition" on page 122). Implementation changes within a shared object are often referred to as *micro* revisions. Therefore, version definitions of this type are suffixed with a *micro* version number based off of the previous *minor* number to which the internal changes have been applied.

Any industry standard interfaces should use a version definition name that reflects the standard. Any vendor interfaces should use a version definition name unique to that vendor (the company's stock symbol is often appropriate).

Private version definitions indicate symbols that have restricted or uncommitted use, and should have the word *private* clearly visible.

All version definitions result in the creation of associated version symbol names. Therefore, the use of unique names and the *minor/micro* suffix convention reduce the chance of symbol collision within the object being built.

The following version definition examples show the use of these naming conventions:

- SVABI.1 - defines the *System V Application Binary Interface* standards interface.
- SUNW_1.1 - defines a SunSoft public interface.
- SUNWprivate_1.1 - defines a SunSoft private interface.
- SUNW_1.1.1 - defines a SunSoft internal implementation change.

Defining a Shared Object's Interface

When establishing a shared object's interface the first task is to determine which global symbols provided by the shared object can be associated to one of the three interface version definition categories:

- Industry standard interface symbols conventionally are defined in publicly available header files and associated manual pages supplied by the vendor, and are also documented in recognized standards literature.
- Vendor public interface symbols conventionally are defined in publicly available header files and associated manual pages supplied by the vendor.
- Vendor private interface symbols can have little or no public definition.

By defining these interfaces, a vendor is indicating the commitment level of each interface of the shared object. Industry standard and vendor public interfaces remain stable from release to release. You are free to bind to these interfaces safe in the knowledge that your application will continue to function correctly from release to release.

Industry standard interfaces might be available on systems provided by other vendors, and thus you can achieve a higher level of binary compatibility by restricting your applications to use these interfaces.

Vendor public interfaces might not be available on systems provided by other vendors, however these interfaces will remain stable during the evolution of the system on which they are provided.

Vendor private interfaces are very unstable, and can change, of even be deleted, from release to release. These interfaces provide for uncommitted or experimental functionality, or are intended to provide access for vendor specific applications only. If you who wish to achieve *any* level of binary compatibility you should avoid using these interfaces.

Any global symbols that do not fall into one of the above categories should be reduced to local scope so that they are no longer visible for binding (see "Reducing Symbol Scope" on page 38).

Versioning a Shared Object

Having determined a shared object's available interfaces, the associated version definitions are created using a `mapfile` and the link-editors `-M` option (see "Defining Additional Symbols" on page 32 for an introduction of this `mapfile` syntax).

The following example defines a vendor public interface in the shared object `libfoo.so.1`:

```
$ cat mapfile
SUNW_1.1 {                      # Release X.
        global:
                foo2;
                foo1;
        local:
                *;
};
$ cc -G -o libfoo.so.1 -h libfoo.so.1 -z text -M mapfile foo.c
```

Here the global symbols `foo1` and `foo2` are assigned to the shared objects public interface `SUNW_1.1`. Any other global symbols supplied from the input files are reduced to local by the *auto-reduction* directive "*" (see "Reducing Symbol Scope" on page 38).

Note – Each version definition `mapfile` entry should be accompanied by a comment reflecting the release or date of the update. This information helps coordinate multiple updates of a shared object, possibly by different developers, into one version definition suitable for delivery of the shared object as part of a software release.

Versioning an Existing (Non-versioned) Shared Object

Versioning an existing, non-versioned shared object requires extra care, as the shared object delivered in a previous software release has made available *all* its global symbols for others to bind with. Although it can be possible to determine the shared objects *intended* interfaces, it can be the case that others have discovered and bound to other symbols. Therefore, the removal of any symbols *might* result in an applications failure on delivery of the new versioned shared object.

The internal versioning of an existing, non-versioned shared object can be achieved if the interfaces can be determined, and applied, without breaking any existing applications. The runtime linker's debugging capabilities can be useful to help verify the binding requirements of various applications (see "Debugging Aids" on page 77). However, this determination of existing binding requirements assumes that all users of the shared object are known.

If the binding requirements of an existing, non-versioned shared object can not be determined then it is necessary to create a new shared object file using a new versioned name (see "Coordination of Versioned Filenames" on page 136). In addition to this new shared object, the original shared object must also be delivered so as to satisfy the dependencies of any existing applications.

If the implementation of the original shared object is to be frozen then maintaining and delivering the shared object *binary* might be sufficient. If however, the original shared object might require updating - for example, through patches, or because its implementation must evolve to remain compatible with new platforms - then an alternative source tree from which to generate the shared object can be more applicable.

Updating a Versioned Shared Object

The only changes that can be made to a shared object that can be absorbed by internal versioning are *compatible* changes (see "Interface Compatibility" on page 116). Any *incompatible* changes require producing a new shared object with a new *external* versioned name (see "Coordination of Versioned Filenames" on page 136).

Compatible updates that can be accommodated by internal versioning fall into three basic categories:

- adding new symbols.

- creating new interfaces from existing symbols.
- internal implementation changes.

The first two categories are achieved by associating an interface version definition with the appropriate symbols. The latter is achieved by creating a *weak* version definition that has no associated symbols.

Adding New Symbols

Any compatible new release of a shared object that contains new global symbols should assign these symbols to a new version definition. This new version definition should inherit the previous version definition.

The following `mapfile` example assigns the new symbol `foo3` to the new interface version definition `SUNW_1.2`. This new interface inherits the original interface `SUNW_1.1`:

```
$ cat mapfile
SUNW_1.2 {                      # Release X+1.
        global:
                foo3;
} SUNW_1.1;

SUNW_1.1 {                      # Release X.
        global:
                foo2;
                foo1;
        local:
                *;
};
```

The inheritance of version definitions reduces the amount of version information that must be recorded in any user of the shared object.

Internal Implementation Changes

Any compatible new release of the shared object that consists of an update to the implementation of the object - for example, a bug fix or performance improvement - should be accompanied by a *weak* version definition. This new version definition should inherit the latest version definition present at the time the update occurred.

The following `mapfile` example generates a weak version definition `SUNW_1.1.1`. This new interface indicates that the internal changes were made to the implementation offered by the previous interface `SUNW_1.1`:

```
$ cat mapfile
SUNW_1.1.1 { } SUNW_1.1;      # Release X+1.

SUNW_1.1 {                    # Release X.
        global:
                foo2;
                foo1;
        local:
                *;
};
```

New Symbols and Internal Implementation Changes

If both internal changes and the addition of a new interface has occurred during the same release, both a weak version and a interface version definition should be created. The following example shows the addition of a version

definition SUNW_1.2 and an interface change SUNW_1.1.1 which are added during the same release cycle. Both interfaces inherit the original interface SUNW_1.1:

```
% cat mapfile
SUNW_1.2 {                      # Release X+1.
        global:
                foo3;
} SUNW_1.1;

SUNW_1.1.1 { } SUNW_1.1;        # Release X+1.

SUNW_1.1 {                      # Release X.
        global:
                foo2;
                foo1;
        local:
                *;
};
```

Note – The comments for the SUNW_1.1 and SUNW_1.1.1 version definitions indicate that they have both been applied to the same release.

Migrating Symbols to a Standard Interface

Occasionally, symbols offered by a vendors interface become absorbed into a new industry standard. When creating a new standard interface it is important to maintain the original interface definitions provided by the shared object. To accomplish this it is necessary to create intermediate version definitions on which the new standard, and original interface definitions, can be built.

The following `mapfile` example shows the addition of a new industry standard interface `STAND.1`. This interface contains the new symbol `foo4` and the existing symbols `foo3` and `foo1` which were originally offered through the interfaces `SUNW_1.2` and `SUNW_1.1` respectively:

```
$ cat mapfile
STAND.1 {                       # Release X+2.
        global:
                foo4;
} STAND.0.1 STAND.0.2;

SUNW_1.2 {                      # Release X+1.
        global:
                SUNW_1.2;
} STAND.0.1 SUNW_1.1;

SUNW_1.1.1 { } SUNW_1.1;        # Release X+1.

SUNW_1.1 {                      # Release X.
        global:
                foo2;
        local:
                *;
} STAND.0.2;
                                # Subversion - providing for
STAND.0.1 {                     # SUNW_1.2 and STAND.1 interfaces.
        global:
                foo3;
};
                                # Subversion - providing for
STAND.0.2 {                     # SUNW_1.1 and STAND.1 interfaces.
        global:
                foo1;
};
```

Here the symbols `foo3` and `foo1` are pulled into their own intermediate interface definitions which are used to build the original and new interface definitions.

Note – The new definition of the SUNW_1.2 interface has referenced its own version definition symbol. Without this reference the SUNW_1.2 interface would have contained no immediate symbol references and hence would be categorized as a weak version definition.

When migrating symbol definitions to a standards interface the requirement is that any original interface definitions continue to represent the same symbol list. This requirement can be validated using pvs(1). The following example shows the symbol list of the SUNW_1.2 interface as it existed in the software release X+1:

```
$ pvs -ds -N SUNW_1.2 libfoo.so.1
        SUNW_1.2:
                foo3;
        SUNW_1.1:
                foo2;
                foo1;
```

Although the introduction of the new standards interface in software release X+2 has changed the interface version definitions available, the list of symbols provided by each of the original interfaces remains constant. The following example shows that interface SUNW_1.2 still provides symbols foo1, foo2 and foo3:

```
$ pvs -ds -N SUNW_1.2 libfoo.so.1
        SUNW_1.2:
        STAND.0.1:
                foo3;
        SUNW_1.1:
                foo2;
        STAND.0.2:
                foo1;
```

It is possible that an application might only reference one of the new subversions, in which case any attempt to run the application on a previous release will result in a runtime versioning error (see "Binding to a Version Definition" on page 126).

In this case an applications version binding can be promoted by directly referencing an existing version name (see "Binding to a Weak Version Definition" on page 130).

For example, if an application only references the symbol `foo1` from the shared object `libfoo.so.1`, then its version reference will be to `STAND.0.2`. To allow this application to be run on previous releases, the version binding can be promoted to `SUNW_1.1` using the link-editor's `-u` option:

```
% cat prog.c
extern void foo1();

main()
{
        foo1();
}
% cc -o prog prog.c -L. -R. -lfoo
% pvs -r prog
        libfoo.so.1 (STAND.0.2);

% cc -u SUNW_1.1 -o prog prog.c -L. -R. -lfoo
% pvs -r prog
        libfoo.so.1 (SUNW_1.1);
```

In practice it is rarely necessary to promote a version binding in this manner, as the introduction of new standards binary interfaces is rare, and most applications reference many symbols from an interface family.

B