Steps to Version Your Shared Library

1. Introduction

With the enhancement and evolution of software, there can be growing pains, especially when an existing base of software depends upon earlier functionality. When changes are made to software libraries, established software engineering practices indicate making compatible changes such that installed software investment is preserved. Shared libraries provide a great challenge in this regard as old software dynamically binds with new shared libraries. When changes would break compatibility, versioning provides a mechanism to deliver new functionality while providing compatibility for existing applications. This paper explores the reasons and the techniques that should be employed to version a shared library using whole library versioning, also known as V.4 versioning. Although it provides steps to consider, it is not intended to be a complete recipe, rather steps to consider when a shared library versioning in considered as part of a development plan for a particular library.

1.1 Terms and Definitions

This paper uses the following terms as a common basis for discussion.

Binding

The process the dynamic loader goes through populating a process' procedure linkage tables and data linkage tables with the addresses of shared library routines and data. When a symbol is bound, it is accessible to the program.

Compatible

This paper considers compatibility from the stand point of the software expected to operate. And forward or backward compatibility is in terms of moving that software forward or backward on releases. With that, this paper defines the two terms as follows:

Backward Compatible

The ability to take software compiled on a release and expect it to operate correctly on an earlier release of software. Software is usually not designed with this compatibility in mind.

Forward Compatible

The ability to take software compiled on a release and expect it to operate correctly on future releases of the host system. Software is usually designed this way; compatibility with future releases is an implied requirement.

Deferred Binding

The process of waiting to bind a procedure until a program references it. Deferred binding can save program start-up time. Compare with "Immediate binding."

Dependent Library

When creating a shared library with `ld`, you can place additional shared libraries on the load line using the `-l` option. The created shared library has a dependency on the other referenced libraries. When the first library is loaded, the other "supporting" libraries will also be loaded. It is important to note that the supporting library's current version is made a dependency for the first library.

Dynamic Loader

Code that attaches a shared library to a program.

Explicit loading

The process of using the `shl_load()` function to load a shared library into a running program. The library is dynamically loaded as part of the `shl_load()` processing.

Export Table

A list of symbols in a shared library available for other modules.

Immediate Binding

By default, the dynamic loader attempts to resolve symbols when they are referenced. By using Immediate Binding, all routines and data will be bound when the library is loaded. For implicitly loaded libraries this binding happens at program start-up.

Implicit Loading

Occurs when the dynamic loader automatically loads any required libraries when a program starts execution. Compare with “explicit” loading.

Import Stub

A short code segment generated by the linker for external references to shared library routines.

Incomplete Executable

An executable (a.out) file that uses shared libraries. It is “incomplete” because it does not actually contain the shared library code that it references; instead, the shared library code is attached when the program runs.

Position-Independent Code (PIC)

Object code that contains no absolute addresses. All addresses are specified relative to the program counter or indirectly through the linkage table. Position-independent code can be used to create shared libraries.

Versioning

The methodology and capability of one object referencing a certain vintage of another object based upon when the two objects were linked together. This capability allows for compatibility by providing an unchanging binding between objects even when new vintages of objects are created.

2. Shared Library Background

2.1 What is a shared library?

An HP-UX shared library is a bound collection of routines and data used by other software modules. When a shared library is loaded, either implicitly when a program is started, or explicitly via the `shl_load()` routine, it is mapped into the process's address space by the dynamic loader. Several applications can access the same code space of the shared library, hence the name **shared** library. Applications built with shared libraries do not have copies of routines and data from the shared library, instead they have an import table of symbols to be satisfied by the shared library when the application is run.

Shared library symbols available for use by other modules are provided in an export table. The dynamic loader resolves access symbols, either when the library is loaded or the first time a symbol is accessed. The default is to bind symbols when they are first accessed, that is deferred binding.

One obvious benefit of shared libraries is that they can reduce the amount of disk and memory required.

2.2 What is V.4 Versioning?

Library level versioning, also known as V.4 versioning, is a technique where the whole library is replaced with an updated library. This means that there are multiple complete libraries, one with the latest behavior and feature set and one or more previous versions with older behavior. A symbolic link is used to point to the latest library using a standard name, so that the programmer doesn't need to be concerned about which version is the current version. By convention, library names end with a `.`*`numeric`* suffix (e.g. `.1`, `.2`, etc.) The latest library version has the largest numbered suffix. The standard library name ends in a `.sl` suffix, and it will be a symbolic link pointing at the latest numbered library.

At a high level, a shared library is versioned by incrementing the numeric suffix and changing the `.sl` symbolic link to point to the new name. There are other considerations and details covered later in this paper.

3. Why Version your Shared Library?

Versioning of a shared library should not be done capriciously as it complicates the amount of support required for a library by having a second version. Complications can also spill over to other libraries if a dependency releationship exists between the two libraries.

3.1 Incompatible Changes

Libraries, areusually required to grow and change as a result of ever increasing tasks the software is required to perform. Many times software evolution can be done in such a way to be compatible, which is desirable. At other times it is not possible or desirable to maintain compatibility, possibly due to new specifications, standard compliance or technical constraints.

3.1.1 Determining if your Change is Incompatible

Shared library compatibility is maintained when an older software module, either a program or another shared library, still operates correctly with a shared library. Changes to functions and data that are not exported do not break compatibility. Changes to exported functions and data elements cause compatibility problems with the modules that import those symbols. Although not absolute, the following describe incompatible changes:

For functions, the type of changes that are typically incompatible include:

- Changing the number of parameters, parameter order, types or meanings.
- Changing return value types or meanings.
- Changing a function to take advantage of another library's change.
- Depending on another shared library that has changed, even if you don't know why theother library has changed.
- Changing macro implementations of functions in public header files to be incompatible with previous macros.
- Replacing a public function with a macro, thus removing the function as an exported symbol.

For exported data elements, the following changes are typically incompatible:

- Changing the size.
- Changing the default value.
- Changing the element locations in structures.
- For C++ libraries, changing the type of a function parameter, even if the size remains the same.
- Changing interface variable types.
- Changing non-opaque elements of structures in public header.

In general, if the new version of a shared library can be used in place of the old shared library, compatibility is maintained.

3.1.2 Making Changes Compatible

Before considering library versioning, it would be good to look at the possibility of altering a change to make it compatible. It may be possible to add new entry points or data types to expand the behavior of functions without affecting compatibility. Even if it involves more work to make a change compatible, it will likely be less effort when the costs of versioning, and testing and maintain both versions are considered. In general, the following types of changes do not break compatibility:

For exported functions:

- Adding a new exported function to handle new behavior
- If a function has an "operations type" argument, adding new operations.
- Increasing the domain and/or range of argument and return values if the old domain and range work as they did previously.
- Adding new parameters if those parameters are not needed by the function when called in a previously acceptable manner.
- Adding new argument types to functions that parse arguments with `stdarg's` or `vararg's` based upon other criteria.

For exported data elements:

- Increasing the range of values as long as the new values were previously represented with undefined behavior.
- Adding new exported data elements.

For header files associated with a shared library

- Changing opaque data structure elements as long as non-opaque element locations are not changed and the overall size of the structure is unchanged.
- Adding new #define constants or macros.
- Adding new typedef's or variable types.

3.2 Changes in Other Libraries

Sometimes the changes in one library may cause changes in other libraries as well. This can be due to sudden interface changes or that an enhanced interface needs to be used consistently by both direct and indirect users of a library. It may also come about simply by recombining the dependent library. This would be due to changes in the header files associated with the versioning library.

3.2.1 Latent Versioning

Even if the immediate changes of a versioning library do not cause a dependent library to version, it creates a situation where versioning may be required in the future. This can be explained in fig 1. In the dia-

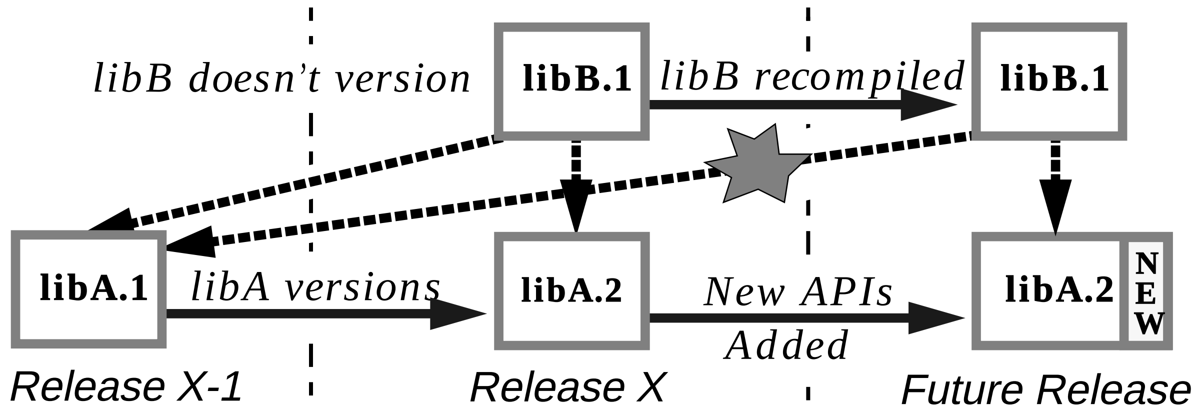

FIGURE 1. Latent Versioning

gram, libA versions at release X, but it is determined that libB doesn’t need to vefrsion. This creates the case where a single library, libB.1 is expected to operate correctly with both libA.1 and libA.2. Even though there are no immediate difficulties with this situation, there is the possibility of future versioning of libB when it depends on API’s or capabilities only available in libA.2.

Both library providers need to monitor this situation, not only at the time of libA versioning, but until the point when libB versions as well.

4. How to Version your Library

Once you determine you need to version your library, what step do you need to go through to version your library. The practice for naming a new version of a library is to use the old name of the library with a numeric suffix one greater than the previous version (e.g. use `libxyz.2` when versioning `libxyz.1`). If your library is delivered directly with a `.sl` suffix, see section 4.4.1 below.

4.1 Source Tree

For the most part, your source files should not be affected by versioning. Most likely you want to symbolically tag the appropriate revision of all files used to build the previous version of the library. This will provide a means of recreating that library should a patch be required. This tagging should take place before any other step.

4.2 Building the New Library

The dynamic loader relies on the internal name of a library to resolve which file to use at runtime. The internal name is set with the "`+h` *`internal_name`*" option to the `ld` command. Change your makefile or build script to have the new versions name as the argument to the `+h` option.

4.3 Delivering the New Library

Normally the new version of library will be delivered in the same location that the old one was. Since the library names have different suffixes, there isn't a name collision. You also want to make sure that your old library is delivered in future releases, both for installation and for updates. The old library provides run-time support for software previously linked with it.

4.3.1 Handling the Old Library

The old version of the library needs to be preserved so that you can provide it on the installation media. You will also need to keep around an appropriate older compilation/testing environment for the old library so that you are able to support the library. This is not much different than supporting a non versioned library, as you need to keep a development environment corresponding to the oldest release you support.

The old version of the library needs to be delivered so that old applications continue to operate correctly. You may want to make installation of the old library optional if those installing the library have a clear way of knowing whether or not they need the old library version.

4.3.2 Symlinks

As part of versioning your library, you want to make sure the "`.sl`" symlink points at the new library. If you previously delivered your library as `libxyz.sl`, then see section 4.4.1 for more details on how to create the initial version of your library. For your library `libxyz`, there should be a symbolic link `libxyz.sl` that point at the new library version. For example if you are changing from version 1 to version 2, your library's entries in the installation directory before and after versioning will look like the files found in the directory listing below:

Before versioning libxyz

```
-r-xr-xr-x 1 bin bin 21312 Dec  1 1994 /usr/lib/libxyz.1
lrwxr-xr-x 1 bin bin    17 Dec  1 1994 /usr/lib/libxyz.sl -> /usr/lib/libxyz.1
```

After versioning libxyz from version 1 to version 2

```
-r-xr-xr-x 1 bin bin 21312 Dec  1 1994 /usr/lib/libxyz.1
-r-xr-xr-x 1 bin bin 32854 Jul 11 1995 /usr/lib/libxyz.2
lrwxr-xr-x 1 bin bin    17 Jul 11 1995 /usr/lib/libxyz.sl -> /usr/lib/libxyz.2
```

After the versioning was completed, a new file existed, libxyz.2, and the symbolic link, libxyz.sl, now points at that version.

4.4 Other Considerations

4.4.1 Pre V.4 Libraries

If you deliver your library as `libxyz.sl` and haven't been building it with the +h option, then you'll need to change how you deliver the old library as well as the new library. The old library should be delivered as `libxyz.0` in the same directory that you were delivering the `.sl` version. The new library should be delivered as `libxyz.1` and you should create a symbolic link, `libxyz.sl` that points to `libxyz.1`. you should also follow the other steps outlined in sections 4.2 and 4.3.

4.4.2 Source Compatibility

When you make changes that necessitate versioning, you still want to consider the source implications to those who use your library. Versioning provides for binary compatibility for existing applications, therefore if you make your changes in a source compatible manner, you provide a migration path for all users of your library. Sometimes the nature of you changes do not allow for 100% source compatibility. Your goal then, should be to minimize to amount of rework that users have to go through to compile with the new library.

If you are not able to make your changes in a source compatible manner, provide clear documentation that outlines the changes needed to be made when using your library.

4.4.3 Allowing for Future Development

Since your newly versioned library doesn't have a legacy code set to operate with, you have some freedom to make changes with the future direction of your library in mind. You'll need to consider these expansion related changes in light of providing source compatibility, but it is usually better to version your library less often due to the development and support costs. Therefore, if you are aware of interface modifications needed to support future technology, make them now to save those the costs of a second versioning effort.

Some of the techniques for allowing for future development that you may consider:

Structure Growth - Provide space in structures for future elements. This is especially important for structures that are allocated directly by the calling code.

Constructor Functions - An even better method for handling future structure growth is to provide functions that are used to allocate instance of interface structures for your library. Providing such function make it easier to extend structures when all structures are created this way, as code calling you library doesn't know the `sizeof` your structures.

Accessor Methods - For the most part, it is better to provide an expandable macro or function to access structure values, especially flags and bit masks. This insulates somewhat the user of a structure from its implementation, thus allowing for future expansion.

Versioning Structures - If you are making structure changes, you may want to provide a field used to version the structure layout. That way, you can change the structure and have different code read/write the various versions of the structure.

Expandable Macros - Although macro often simplify the written of code through substitution, they have greater compatibility implications than functions, as macros contents are substituted and compiled in-line into the calling object code. One method for providing for future macro changes is to provide a macro that uses a library or structure conditional variable to predicate the macro implementation or to call an equivalent function. For example consider a macro used to access the file descriptor of some structure S:

```
#define get_file_desc(S)      ((S->version == 1)?S->fd:GetFD(S))
```

The field `version` is a structure versioning field and `GetFD()` is a function to get a file descriptor for any version of the structure. This macro is basically a quick implementation for version 1 of a structure. If the structure version is changed, then the function is called.

4.4.4 Communicating Changes

If you version your library, you need communicate with those developers who depend on your library the changes you made to the library and any effects the versioning will have on the way to use your library. Some of the issues you may need to communicate include:

- Libraries that may or may not be used with the new version of the library.
- The compatibility the library provides when used in a legacy environment. This should include the support for old hardware, obsoleted features, or previous data formats, etc.
- Any changes to compile or link with the new library. Include source incompatibilities, changes in include file usage.
- The capabilities and features the new library version provides.

4.4.5 Message Catalogs

If you library is localized, you will need to decide how to handle message catalogs. This is primarily due to changing messages can affect both the old and new library. Since the old library is some what fixed in time, the messages accessed by the old library do not change. If you use `.msg` files to store your messages, you can add comments to all messages referenced by the old library. You can also adopt the policy that when changing a message, you also change the message number. If you use `insertmesg`, you may want to specify a new set number to use with new messages. That way you correlate your message sets with the library version.

4.4.6 Explicitly Loaded Libraries

For shared libraries loaded explicitly via `shl_load()`, you may need to handle them a little differently from an implicitly loaded library. These libraries typically are used to implement some variable behavior, either based upon attached hardware, some environment variable or the like.

If you are versioning a library that dynamically loads another shared library, you will need to consider carefully the effect on the changes to the versioning library to the explicitly loaded library. Consider again figure

1 as the same kind of dependency exists between the two libraries. If the dynamically loaded library, libB in the figure, needs to change any time after libA versions, there is the distinct possibility for incompatibilities.

If you end up versioning a explicitly loaded shared library, there may be some concerns as to how it is used in the old version of the loading library. If the library has always been loaded with a name `/some/path/libdynamic.sl` and is really a symbolic link to `/some/path/libdynamic.1` or is a library that has never versioned, you will need to make provisions for uniquely loading the proper version. If you are versioning your explicitly loaded library for the first time, use the suffix ".1" with the library, e.g. `/some/path/libdynamic.1`. Also when you go to load the library, use the file name with the ".1" suffix.

4.4.7 Removing Old Code

Since you are versioning your library, it may be a time to remove code no longer required to be part of the library. For example if you have obsoleted some functionality or have used HP's intra-library (routine level) versioning, versioning your library provides a good opportunity to remove such dead code.

4.5 Testing

After completing the tasks necessary to version your library, you need to verify its correctness. The normal library testing processes you go through should be sufficient to verify proper functionality of the library, but there are some special steps to verify that the versioning process was accomplished successfully. After the versioning is complete, you will have another library to test.

4.5.1 Testing the Old Library

To verify proper operation of the older library, you will need to execute test suites compiled with the old library on a system containing both the old and new library versions. Since those using your the old version of your library are not able to compile and link with it on the latest release, your tests need to be compiled and linked against the old library in a similar environment. In the same way you preserved the previous version of your library, you can preserve a compiled and linked copy of your test suite. Then to test the old version of the library, you install and run the old test suite without recompilation.

Should you have to patch the old version of your library, you may need to be able to augment the test suite for the old library. This will likely require a earlier release environment to develop the old version of the tests much in the same way that bug fixes are made to the old library in an appropriate older release environment.

4.5.2 Testing the New Library

The new library shouldn't require testing beyond the normal qualification you go through to certify your library. You will need to watch for the latent versioning situation described in section 3.2.1, especially when future changes are made to the new library version.

Summary

With the requirements of software growing, especially in unforeseen ways, software development techniques need to be developed to maintain compatibility with legacy systems while at the same time providing for growth to accommodate the future. V.4 versioning is a not to costly method to provide for this compatibility and growth. While other methods have been explored, including HP's own routine level versioning, the current state of the art for Unix systems is V.4 versioning.

Although this papered outlines considerations and techniques to accomplish V.4 versioning, it should be understood that versioning is a technique that has its costs, in terms of disk space and support overhead. For versioning to be successfully employed, its must be used wisely.

References

1. Hewlett-Packard Company, *Programming on HP-UX*, January 1995, B2355-90652