Linker Scripting and Linux Shared Library Versioning

Demystifying linker scripting and shared library exports.

Matt Bisson — cyberbisson.com

November 2023

Shared library versioning is a powerful tool, allowing your library to make new features available without breaking older consumers of them. The problem is — as I've noticed through the years — GCC linker scripting seems like witchcraft to many who'd otherwise use it, and only those with tribal knowledge can maintain the library. I agree: this information can be *hard* to find, and hard to understand. In this article, I'm going to give what I think is a recipe book for using LD version scripting to apply shared library versioning, and keep your library alive and compatible for years.

I write this today because I have personally created external-facing libraries from soup to nuts using these versioning facilities, and I wouldn't say I had an "easy" time figuring out all the details. Hopefully this will help the next person get going a little more quickly. There is a lot here, so please feel free to skip around with the Table of Contents.

1 But… Why?

If you already know the "why," skip right on down to the how(§2)…"

You can certainly do your own versioning scheme for your library, but that gets very messy, very quickly! In my observation, people often choose to do one of the following:

1. *Create new functions with their own names, distinct from the old ones.* So instead of `CreateFile()`, now you have `CreateFileEx()` with different behavior. What a mess, though. What happens if you need to add a third incompatible modification? Do you keep the same name if the inputs are the same, but the behavior is different? Also — and this is important, too — don't you want to choose what the names are in the API? Now your names not only have to describe what they're doing, but they have to include historical context, like "`V1`" in the name.

2. *Create a completely new library for a new version.* Each "dot" release gets its own library file on the system. This might seem like an easy path in the short term, but every time you make a minor breaking change to an interface, you will take a deep sigh, and think about all the work you'll have to do to create a brand new library. Corners will undoubtedly be cut. Plus, what you put onto the operating system will have many redundant copies of executable code that has not changed (see the 10 different JRE versions on your computer: one for each software package that consumes its own specific version). I doubt that every single function has changed from one API version to the next. Further, if you ever want to build all the supported libraries (which you probably do), then you either need to set old versions of the source code in stone, or you need to pepper your code with preprocessor checks to ensure old versions remain unmodified.

3. *Emit API bindings from a definition language, and manually accommodate differences.* This is a decent approach, but it can demand that you build up a *lot* of infrastructure before you can even get off the ground. Often times, too, you have rules about how to add or remove parameters that require manual intervention, as well the need for some initial "negotiation" step that happens at run-time between the library and the client. Perhaps an RPC/IPC mechanism has to facilitate this all, as well.

4. *Just ignore it — nothing but the latest version is OK.* Gee… that's pretty harsh! I mean, if you can do it, go ahead, but stable platforms will lead to more rapid development later on.

Conversely, shared library versioning pretty much Just Works™ — you write your linker script, defining a version and what symbols it comprises. You don't need to give funny version-specific names to your exports, because the linker takes care of this under the covers. You don't need to marshal data through an IPC, because linkages are just `dlopen()` and `dlsym()`. You build everything from one set of source files (if you choose), and you don't *need* to involve the preprocessor for anything. Your OS deployment won't have any redundant copies of anything. Yes, there are a few rules to avoid the "gotchas," but honestly, not too much.

2 How Does This Work?

Let me begin by stating that shared library versioning is not "easy." It requires some discipline to get it right, because you don't want to be adding new functionality that inadvertently blows up the behavior of old functionality. It's sometimes tricky to know what symbols need a version bump (or not!), and what design choices may limit you in the future. Do know, however, that almost all the system libraries in Linux use this functionality, so it's not only possible and reasonable to do this, but *you are not alone.*

As mentioned, it's the GNU linker's scripting language that drives shared library versioning. Historically speaking, this is based on Solaris's shared library versioning schemes (the GNU reference actually directs readers to the Solaris documentation), but the GNU modifications avoid a particular situation where an application runs for some time, then fails unexpectedly when they load an incompatible library. The GNU dynamic linker quickly checks the stated symbols as it initiates the process's execution. The documentation for library scripting is not as clear as it could be, in my opinion. Here are some references:

- See Chapter 3 of The LD manual (or `info` module).
- Specifically the `VERSION` command.
- Section 3 of Ulrich Drepper's How-To.

This is the same compatibility scheme that GNU `libc` uses, though, so there is a lot of "prior art" for empirical examples. You can check out the GLIBC source code repository for files named "`Version`."

2.1 What Does the Linker Script Do?

Speaking specifically of the versioning section of the linker script, the script directs the linker to modify the visibility and associated version for shared library exports. *That's it.* Importantly (for reasons to be explained shortly), the script does not consider any symbol with an "`@`" in its name. During processing, it:

1. … gathers any symbols mentioned within a version ("globbing" supported), and if specified as "`local`," *hides* them, if specified as "`global`," makes them *visible*, appending the version name to the end.
2. … moves through the dependent versions recursively, carrying forward any `global` symbols from the dependency, and dropping any `local` ones. It continues to match what has not already been matched.
3. If no versions exist, but an anonymous "version" exists, visibility changes, as with named versions, but all symbols remain in the default, unnamed version. Named and anonymous versions do not coexist.
4. Any symbols that do not match the above processes land in the default unnamed version with their already specified visibility.

Since this occurs at the *linking* phase, the "`-fvisibility`" compilation flag still takes effect. This flag marks symbols as visible or hidden during `.o` compilation, and the linker subsequently gathers all the visible symbols from the `.o` file for consideration. If the default is "`hidden`," the "`visibility`" attribute must decorate the symbol, and it must be set to "`default`." The `__attribute__` keyword overrides any compiler flag to the contrary.

The dynamic linker uses the "`@`" token to label the symbols. Thus, a symbol, `foo` targeted to a specific version (e.g., `VER_1.0`) will appear in the executable as "`foo@VER_1.0`." The "`@@`" token not only indicates the version, but indicates that the given version is the default for any compilation that does not explicitly specify one. The `foo` for `VER_1.1` *and unspecified versions* (during compilation) appear as "`foo@@VER_1.1`."

2.2 Additional Methods of Versioning

The linker script allows us to put named symbols into versions as we please, using explicit and dependent versions, and using wild-cards. Replacing a symbol is not trivial, however, since the script does not allow renaming of symbols. To rename a symbol, this implies that you have two symbols, not one — both named `foo` — that exist in separate versions. If the linker finds the same symbol twice, it reports an error.

Because of this, we need to fix-up the compilation unit itself so that it provides a uniquely named C symbol, and then manually renames the symbol to one with an "`@`" in the name. Recall that "`@`" is a special character that indicates a versioned symbol, and is one that the linker script cannot affect.

GCC and Clang provide the "`.symver`" directive. It takes the form:

```
.symver symbol, name@[@]ver
```

Wrap this in an `__asm__` block in C/C++ code.

Using the example from the previous section, assume we have a "`fooV1`" C symbol to inject into the "`VER_1.0`" version as the shared library export named "`foo`." This export goes directly from the C code into the shared library, without being altered by the linker script:

```
__asm__(".symver fooV1, foo@VER_1.0");
```

2.3 Version Scripting Syntax

The syntax is fairly straightforward. Version specifications in the script take this form:

```
VERSION     ::= NAME '{' ( SCOPE? SYMBOL-LIST )* '}' NAME? ';'
            ::=      '{' ( SCOPE? SYMBOL-LIST )* '}'       ';'

SCOPE       ::= ( 'global' | 'local' ) ':'
SYMBOL-LIST ::= PATTERN ';' SYMBOL-LIST*

NAME        ::= [A-Za-z_][A-Za-z0-9_]*
PATTERN     ::= '"' [A-Za-z0-9_?*]+ '"'
            ::=     [A-Za-z0-9_?*]+
```

A version can have a name, or it cannot. If any version is named, *all* versions must have names. The named `VERSION` form creates a dependency by specifying the version on which it depends after the final closing bracket. This brings the exports from the dependency into the dependent version.

Labels indicate scoping with the "`global`" or "`local`" keywords, but if symbols fall outside the scope labels, they are `global` in scope. Each symbol terminates with a semicolon, and may be surrounded by double-quotes to prevent globbing.

Note that this is a simplification for C-only exports, but the version script does indeed support things like "`extern "C++" ...`," and symbols utilizing valid, un-mangled C++ names.

2.4 Practical Application

The following example illustrates how to create a library using the techniques described in the prior sections.

Let's first define the API with some function declarations. In a public header, we can simplify the coder's life with some quick macros. Here's one for marking a function as an exported symbol from shared library:

```
#define MY_API_EXPORT __attribute__((visibility ("default")))
```

Now let's make another macro to simplify exporting a symbol, specifying a particular version:

```
#define MY_API_EXPORT_MAPPING(sym, name, ver) \
    __asm__(".symver " #sym "," #name "@MY_API_" #ver)
```

Here's a quick sample interface we'll use to explore some scenarios.

```
MY_API_EXPORT void foo(void);
MY_API_EXPORT void bar(void);
void undecorated(void);
__attribute__((visibility ("hidden"))) void hidden(void);

MY_API_EXPORT void internal(void);
MY_API_EXPORT void unmatched(void);
```

So you can see we have at least a couple of exports here, but the linker script itself is going to decide where they (most of them, anyway) end up. Pass the following linker script to the linker with "`--version-script=<file>`":

```
/* Linker scripts use C-style comments only.
 *
 * Exported symbols: Keep symbols grouped by module, and in alphabetical
 * order. */
MY_API_1.0 {
    global:
        bar;
        /* foo: REPLACED in v1.1. */
        hidden;
        non_existant; /* Not defined in source; no error, though. */
        undecorated;
};

MY_API_1.1 {
    global:
        foo; /* REPLACES v1.0 API. */
} MY_API_1.0;

MY_API_INTERNAL /* Un-versioned. */ {
    global:
        internal;

    local: *;
};
```

Given the header file declarations, and what's in the linker script, we can tell that this library supports "`MY_API_1.0`" and "`MY_API_1.1`" for public consumption, and "`MY_API_INTERNAL`" with no real compatibility guarantees. We can also surmise what goes where, but we'll need the source file to be certain. Here's the completed picture:

```
void foo(void) {
    /* Version 1.1 things... */
}

MY_API_EXPORT void foo_v1(void) {
    /* Version 1.0 things... */
}

MY_API_EXPORT_MAPPING(foo_v1, foo, 1.0);

void bar(void) {
    /* Unchanged since Version 1.0. */
}

void undecorated(void) {
    /* Dependent on -fvisibility setting. */
}

void hidden(void) {
    /* Explicitly hidden in the declaration! */
}

void internal(void) {
    /* Internal, un-versioned functionality. */
}

void unmatched(void) {
    /* Marked as "visible", but not claimed by any version,
     * so it falls to the "local" section, and is hidden. */
}
```

Given all these permutations, we expect these results:

- `foo()`: Explicitly visible, unclaimed by `MY_API_1.0`, it fell to, and was claimed by `MY_API_1.0`.
- `foo_v1()`: Explicitly visible, and explicitly renamed to `foo` in the `MY_API_1.0` version. The linker script never touched it because it had an `@` in the name.
- `bar()`: Explicitly visible, and claimed by the `MY_API_1.0`.
- `hidden()`: This was hidden during compilation, so it was never visible to the linker, and it will not be exported, despite being mentioned in the version script. The linker does not raise an error when it cannot find a symbol specified in the linker script.
- `undecorated()`: This is not explicitly exported in the source, so it depends on the `-fvisibility` setting. If it's visible, it will appear in `MY_API_1.0`; if it's not visible, it will not be exported.
- `internal()`: This function was not claimed by the first two versions, so it falls to `MY_API_INTERNAL`, which claims it.
- `non_existant`: Does not exist, and wasn't part of any object being passed to the linker, but it was specified in the linker script. The linker silently ignores this.
- `unmatched()`: This was explicitly marked for export in the source, but no version claimed it. It falls to the "`*`" glob under `local`, and is not exported.

The `.dynsym` table in the shared library should look something like this:

```
19: 00000000     0 OBJECT  GLOBAL DEFAULT   ABS MY_API_1.1
20: 00000000     0 OBJECT  GLOBAL DEFAULT   ABS MY_API_INTERNAL
25: 00000000     0 OBJECT  GLOBAL DEFAULT   ABS MY_API_1.0
 6: 00001143    31 FUNC    GLOBAL DEFAULT    12 foo@MY_API_1.0
 7: 00001124    31 FUNC    GLOBAL DEFAULT    12 foo@@MY_API_1.1
10: 00001105    31 FUNC    GLOBAL DEFAULT    12 bar@@MY_API_1.0
12: 000011a0    31 FUNC    GLOBAL DEFAULT    12 internal@@MY_API_INTERNAL
```

Any API consumer that links against this (referring to compile time, here) finds the 1.1 version of `foo`. After installing on the system, that client *forever* links against "`foo@MY_API_1.1`." Any applications that predate the 1.1 version link specifically against "`foo@MY_API_1.0`," and will find the "compatibility" version we coded up under `foo_v1()` in the C source.

I want to take a brief detour here to mention that you do not need to export the symbols for the special library start-up and shut-down function (`__attribute__((constructor))` and `__attribute__((destructor))`, respectively). The dynamic linking subsystems take care of this for you, and there are no compatibility concerns when linking against an older executable.

3 Guidance

Now that you have a living example of how shared library versioning and GCC linker scripting works, let's talk about what (I think) you should do in a more generic sense.

3.1 Best Practices

Let's talk about how to make life simpler.

3.1.1 Minimize (or eliminate) publicly visible memory layouts

First and foremost, any concrete details you expose in your public header files are going to be a liability later on down the line. Things like structures, enumerations, and `typedef`s are not things that the library "exports," and therefore, you cannot version them.

Consider the following:

```
struct func_info {
    char const *some_string;
    char const *something_else;
};

MY_API_EXPORT void use_info(struct func_info const *);
```

Now, we assume that a v1.0 client calls the function with:

```
use_info(&(struct func_info){.some_string = "input",
                             .something_else = "other"});
```

To support this v1.0 API, we now need to ensure that subsequent changes to the structure do not move or remove these two pointers from the 0th and 8th byte offsets in memory. The v1.0 consumer of the API forever arranges the memory in this way — that's hard-coded into the executable. This means:

1. Never add fields to the beginning of a public structure, or insert them between existing fields.
2. Never remove a field from the structure.
3. Even if the functions that access these structures do not change their signatures, they must access the data in a way that is appropriate for the API consumer's version. For example, do not require a populated third field without increasing the version of such a function.

If you ever make an incompatible change to a public structure, you then have to preserve the original structure as an internal compatibility structure, and provide compatibility versions of any exported functions that used that original structure (e.g.: "`use_info@MY_API_1.0`" supports the old structure, and "`use_info@@MY_API_1.1`" understands the new one).

All the same complexity applies to enumerations. Do not insert changes to enumeration constants, do not re-use a numerical constant for a different meaning, and do not remove constants.

Since functions *are* things we can version, I propose the following rule: *hide any structures behind functions.* In a simple case, this can look like a function that simply takes a number of parameters:

```
use_info(/* some_string= */ "input",
         /* something_else= */ "other");
```

...or it can look more like an opaque handle with a creation function, and mutators or accessors, as needed:

```
typedef struct func_info *info_handle;
typedef struct func_info const *const_info_handle;

MY_API_EXPORT info_handle make_info(
    char const *some_string,
    char const *something_else);
MY_API_EXPORT void free_info(info_handle);
MY_API_EXPORT char const *info_get_some_string(const_info_handle);
MY_API_EXPORT void info_set_some_string(info_handle, char const *);

MY_API_EXPORT void use_info(info_handle);
```

Depending on what you need to access, what you need to set, and how frequently you update the consumers of that data, this may be worth the effort, or it may feel like total overkill. I have personally done things both ways at various times, and have found that there's a lot less "explaining" and "evangelizing" when handles are opaque, and access is through functions. In a larger project, it can be hard to observe every modification, so using direct structures allows for a higher probability of mistakes just makes sense.

3.1.2 Callbacks are not versioned: take special care

Although it may initially seem different, this is effectively the same problem as a publicly exposed structure. If your API requires a callback, the type that defines that callback is not versioned. Consider the following:

```
typedef void (my_callback)(char const *data, int a);
MY_API_EXPORT void do_op(my_callback *on_complete);
```

A v1.0 client will dutifully pass a function that matches that signature, look at the pointer, and all is well. If, however, v1.1 changes the type to something like "`int(int a)`," the result will be a v1.0 client accessing memory at whatever the passed-in `int` has for a value, using an "`int a`" from the stack that was not provided by the caller, and never setting the return value for a caller that expects an integer!

For this reason, utilize the same solution as proposed for updating public structures. Create an internal type for the original callback type, then create a legacy version of `do_op()` for v1.0 clients that understands how to execute the old callback. The new version can do whatever it likes from there on out.

3.1.3 Limit `inline` functions

Functions in the public headers that are marked `inline` are built directly into the executable that consumes your library. They are immune to the help that shared library versioning provides. As the (C) `inline` functions are limited to only functionality exposed through the public headers — functionality that the library client themselves can use — simply following proper shared library versioning practices should protect the consumers. This is a slightly different story for C++, however, since encapsulation allows `inline` *member* functions to affect internal details — those in the `private` section — which (non-Machiavellian) library consumers cannot normally access. Take extra caution here, and exercise the same caution with C++ classes as you did with C structures. This can be a very tricky task with inheritance, and so on.

3.1.4 Explicitly label all library exports

I strongly recommend forcing people to mark shared library exports in the public headers with something similar to the example export macro above, "`MY_API_EXPORT`." Tell the compiler to mark all symbols with "`-fvisibility=hidden`" to force the issue. This is a useful warning sign for coders and reviewers, and also serves as a kind of "code as documentation" for readers.

3.1.5 Always create an internal version (containing `local`)

If you have multiple libraries, all deployed at the same time, having some "private" exports may help you. If, from its inception, you structure your script so the internal version is always last in your script, and contains the "`local: *`" statement, this means less work in the future. Without the `local` directive being there, you have to keep moving it manually into new versions as you add them. This is a very minor annoyance, but forgetting to place the directive can lead to confusing linker errors.

3.1.6 Leave obsolete declarations and export information out of public headers

Assuming you do not want your library clients to refer to legacy functions — only the latest ones — leave the legacy functions out of the public headers. In the examples above, I did this with `foo_v1()`. This is clutter to the API consumer, and possibly provides functionality you don't want to expose. You won't even need a declaration for it, since the `.symver` directive does all the work to explicitly add the symbol to the shared library.

3.1.7 Be diligent about documenting when a symbol moves from version to version

Whenever I move a symbol from one version to another (e.g., updating a function from an existing version), I like to put a formulaic comment in place where the symbol resided originally. Notice the example, where we move `foo` into `MY_API_1.1`. Under `MY_API_1.0`, there is a comment that reads, "foo: REPLACED in v1.1." Just a simple comment like this helps the reader know where symbols should show up in the exported symbol table, and when new symbols were added.

3.1.8 Take extreme caution when changing versions across branches

Things can get very hairy, very quickly when making changes to non-trunk branches.

To use a concrete example, an API at version 1.5 on `trunk` should contain the *exact same functionality* as API version 1.5 on "`prod/8.0`." This means that if `prod/8.0` is currently at API version 1.3, you can't just cross-port API version 1.5 over from `trunk`, but you would have to cross-port absolutely everything from 1.4 and 1.5 to the branch! Further modification to 1.5 on `trunk`, too, would need to arrive on `prod/8.0`. Ideally, the branches would be in lock-step before any new API, and the API that you place will be unchanged after that point — subsequent changes arrive in another new API version. You can see how tricky this branch mobility can be, so it is best to avoid it altogether.

3.1.9 Build a version number into your library name

In an extreme, forward-looking sense, building a version number into the name of your library can save you some grief. Assuming you decide that a new version of the library *cannot* be kept compatible with older versions, you can simply bump the version number on the file. You can then choose to set the old library in stone, or not create it at all, leaving clients to go find the new version, and integrate with it. Definitely do this from the very start.

3.1.10 Name your shared library script "`.ld`"

I have noticed some conventions, like in GLIBC, where the scripts are named `Version`, and are placed in various subdirectories. The only modification I might make to this is to suffix the files with "`.ld`" so it's slightly easier to make file-type associations with these files. If you have a single library per script, too, it makes sense to give a file name that reflects that library. These are very minor points, though.

3.2 Checking Your Work

You can verify your changes with a tool like `objdump` or `readelf`.

```
> objdump -T libtestlib.so.1 | grep foo
0000000000001105 g    DF .text  000000000000001f  MY_API_1.1  foo
0000000000001124 g    DF .text  000000000000001f (MY_API_1.0) foo
```

The `objdump` command reports single-version mappings with a "`()`" around the version name, and default mappings without.

```
> readelf --dyn-syms libtestlib.so.1 | grep foo
  8: 0000000000001105    31 FUNC    GLOBAL DEFAULT   12 foo@@MY_API_1.1
  9: 0000000000001124    31 FUNC    GLOBAL DEFAULT   12 foo@MY_API_1.0
```

The `readelf` command uses the familiar "`@`" for single-version mappings, and "`@@`" for default version mappings. In both commands, we can see `foo` mapped to `MY_API_1.0` for those compiled directly against it (only), and to `MY_API_1.1` for anyone else.

The same applies to checking the executable, if you want to ensure that it uses the versions you expect:

```
> readelf --dyn-syms testexe | grep MY_API
  3: 0000000000000000     0 FUNC    GLOBAL DEFAULT   UND bar@MY_API_1.0
  5: 0000000000000000     0 FUNC    GLOBAL DEFAULT   UND foo@MY_API_1.1
```

Looks like we've got a 1.1 API consumer!

4 Summary

Maintaining backward compatibility is difficult, but worth the effort. For Linux applications, I recommend giving serious consideration to the shared library versioning available via LD linker scripting. You must remain cognizant of a few important problems with shared library versioning, but I don't believe this is any different with almost any other versioning scheme. Remember:

- Once released, a version of the API should never change, except for bug fixes.
- Limit what is visible to the API client. Never remove or change existing fields and enumeration values that the library client can access directly.
- Create a new version when *anything* in the behavior or memory layout changes.