Document Number: P2701R0
Date: 2022-11-07
Reply-to: Daniel Ruoso <druoso@bloomberg.net>
Audience: SG15

Translating Linker Input Files to Module Metadata Files

Abstract

This paper establishes a convention for the translation of linker input files to the associated module metadata files. This translation should be useful whenever the convention described in P2577R2[1] is in use. The expectation is that this translation should not be architecture-specific, and that it can be adopted whenever the linker input files can be identified early enough in the build process.

Interoperability Premises

This paper starts with the assumption that if a platform would allow for the linker arguments to be translated into linker input files, we should be able to establish a set of transformation steps that will be usable in all of those scenarios.

The main hurdle identified so far for the adoption of this convention is in mechanisms like “auto linking” on Windows. But at that point it’s mostly because we can’t identify the input files to the linker early enough.

Therefore this convention should be applicable in any architecture where the convention established by P2577R2 can be implemented.

Specific architectures and specific file system implementations may introduce restrictions on the length of the directory entries or the length of the full path to the file. This convention assumes that the overhead introduced here, relative to the linker input file, should be negligible. Environments where this becomes an issue will need to establish an independent mechanism to discover those metadata files.

[1] Ruoso, Daniel (2022). C++ Modules Discovery in Prebuilt Library Releases. https://wg21.link/P2577R2

Document Number: P2701R0
Date: 2022-11-07
Reply-to: Daniel Ruoso <druoso@bloomberg.net>
Audience: SG15

Translation Mechanism

No Symlink Following

The input file to the linker will be used by name, even if that is a symbolic link to a different file. It is the role of the package manager to ensure coherency on the installation of those cases, and therefore the build system should not attempt to resolve a realpath of the library file before looking for the module metadata file.

This is particularly relevant in the case of shared objects in the GNU/Linux ecosystem, as the input to the linker is frequently a symbolic link to a specific versioned filename, and the expectation is that the metadata will be colocated with the file that was used as input to the linker, not to the realpath of that file.

Same Directory

This convention establishes that the C++ module metadata files related to a linker input file will be a different entry in the same directory, and only the name of the directory entry will be different.

Directory Entry Name Translation

The basic premise of this convention is that there will be an ordered set of potential directory entries that will be searched based on the name of the input file to the linker. Those will go from the most specific to least specific, which will allow architecture-specific requirements to take precedence over the more generic use cases.

The translation starts by starting with the name of the directory entry of the input file to the linker. Each one of the following sections modifies that directory entry to resolve to the module metadata file, or introduces new entries to the lookup. In cases where more than one criteria adds additional lookups, they should be combined.

Strip Library File Name Extension

In architectures where it is expected that there will be a difference to how the code consuming a module is translated depending on whether or not the library will be linked statically or dynamically, it will be necessary to allow independent metadata for the different builds of the same library, even if they would, in principle, be ABI-equivalent.

In those architectures (e.g.: Windows), the metadata file will use the full name of the directory entry of the linker input file as a prefix. In architectures where that's not the norm (e.g.:

GNU/Linux), the file extension of the library (.a for archives, or .so for shared libraries) will be removed from the prefix used in the file.

Search For Specific Instruction Set Architecture (ISA)

In some architectures it is possible that a library archive contains builds for different ISAs. In those architectures the lookup should always start with the implementation-specific ISA code, and then fallback to the name without the ISA code. This should allow for the specific case to be addressed, while still allowing the majority of cases to be simpler.

The way in which the ISA is encoded in the directory entry name is by appending a dot character, followed by the implementation-specific ISA code.

Architectures where the linker input file always contains a single ISA (e.g.: GNU/Linux) will not perform the additional lookup.

Metadata file name suffix

Finally, the suffix `.module-metadata` is appended to the file name to find the module metadata file related to that particular linker input.

Tooling reuse

While the specification of the convention is enough for interoperability, it is going to be better if build systems are not required to implement that translation independently.

This paper recommends that toolchain implementations should provide a tool that translates linker argument fragments into the path to the related metadata files found according to the specifics of the implementation.

This paper recommends that toolchain implementations should provide a tool that describes what the metadata file path should be for a library, given the implementation-defined characteristics of the library being produced.

Applying to Specific Architectures

While this paper establishes a framework for the convention to be adopted in different architectures, it's also important to specify what are the choices in specific well-known environments:

Reply-to: Daniel Ruoso <druoso@bloomberg.net>

GNU/Linux

- It is not common for different requirements in behavior between static and dynamic versions of a library, the extension of the linker input (e.g.: .a or .so) will be stripped.
- Common practice is that different ISA or ISA extensions are deployed to different directories, the ISA suffix is not added.

Windows

- It is common in Windows for different preprocessor requirements whether a library is going to be linked statically, or dynamically, the library suffix is not stripped from the file name.
- Windows does not support multi-architecture library archives, the ISA code will not be appended to the name.

MacOS

- It is not common for different requirements in behavior between static and dynamic versions of the library, the extension of the linker input (e.g.: .a or .so) will be stripped.
- It is common, but not a requirement, to support multi-architecture binaries. The search will start with an ISA code, but will fallback to searching without the ISA code.