Swift Binary Serialization Format

The fundamental unit of distribution for Swift code is a *module.* A module contains declarations as an interface for clients to write code against. It may also contain implementation information for any of these declarations that can be used to optimize client code. Conceptually, the file containing the interface for a module serves much the same purpose as the collection of C header files for a particular library.

Swift's binary serialization format is currently used for several purposes:

- The public interface for a module ("swiftmodule files").
- A representation of captured compiler state after semantic analysis and SIL generation, but before LLVM IR generation ("SIB", for "Swift Intermediate Binary").
- Debug information about types, for proper high-level introspection without running code.
- Debug information about non-public APIs, for interactive debugging.

The first two uses require a module to serve as a container of both AST nodes and SIL entities. As a unit of distribution, it should also be forward-compatible: module files installed on a developer's system in 201X should be usable without updates for years to come, even as the Swift compiler continues to be improved and enhanced. However, they are currently too closely tied to the compiler internals to be useful for this purpose, and it is likely we'll invent a new format instead.

Why LLVM bitcode?

The LLVM bitstream format was invented as a container format for LLVM IR. It is a binary format supporting two basic structures: *blocks,* which define regions of the file, and *records,* which contain data fields that can be up to 64 bits. It has a few nice properties that make it a useful container format for Swift modules as well:

- It is easy to skip over an entire block, because the block's length is recorded at its start.
- It is possible to jump to specific offsets *within* a block without having to reparse from the start of the block.
- A format change doesn't immediately invalidate existing bitstream files, because the stream includes layout information for each record.
- It's a binary format, so it's at least *somewhat* compact. [I haven't done a size comparison against other formats.]

If we were to switch to another container format, we would likely want it to have most of these properties as well. But we're already linking against LLVM... might as well use it!

Versioning

Warning This section is relevant to any forward-compatible format used for a library's public interface. However, as mentioned above this may not be the current binary serialization format.

Today's Swift uses a "major" version number of 0 and an always-incrementing "minor" version number. Every change is treated as compatibility-breaking; the minor version must match exactly for the compiler to load the module.

Persistent serialized Swift files use the following versioning scheme:

- Serialized modules are given a major and minor version number.
- When making a backwards-compatible change, the major and the minor version number both MUST NOT be incremented.
- When making a change such that new modules cannot be safely loaded by older compilers, the minor version number MUST be incremented.
- When making a change such that *old* modules cannot be safely loaded by *newer* compilers, the major version number MUST be incremented. The minor version number MUST then be reset to zero.
- Ideally, the major version number is never incremented.

A serialized file's version number is checked against the client's supported version before it is loaded. If it is too old or too new, the file cannot be loaded.

Note that the version number describes the contents of the file. Thus, if a compiler supports features introduced in file version 1.9, but a particular module only uses features introduced before and in version 1.7, the compiler MAY serialize that module with the version number 1.7. However, doing so requires extra work on the compiler's part to detect which features are in use; a simpler implementation would just use the latest version number supported: 1.9.

This versioning scheme was inspired by Semantic Versioning. *However, it is not compatible with Semantic Versioning because it promises* forward-compatibility *rather than* backward-compatibility.

A High-Level Tour of the Current Module Format

Every serialized module is represented as a single block called the "module block". The module block is made up of several other block kinds, largely for organizational purposes.

- The **block info block** is a standard LLVM bitcode block that contains metadata about the bitcode stream. It is the only block that appears outside the module block; we always put it at the very start of the file. Though it can contain actual semantic information, our use of it is only for debugging purposes.
- The **control block** is always the first block in the module block. It can be processed without loading the rest of the module, and indeed is intended to allow clients to decide whether not the module is compatible with the current AST context. The major and minor version numbers of the format are stored here.
- The **input block** contains information about how to import the module once the client has decided to load it. This includes the list of other modules that this module depends on.
- The **SIL block** contains SIL-level implementations that can be imported into a client's SILModule context. In most cases this is just a performance concern, but sometimes it affects language semantics as well, as in the case of `@_transparent`. The SIL block precedes the AST block because it affects which AST nodes get serialized.
- The **SIL index block** contains tables for accessing various SIL entities by their names, along with a mapping of unique IDs for these to the appropriate bit offsets into the SIL block.
- The **AST block** contains the serialized forms of Decl, DeclContext, and Type AST nodes. Decl nodes may be cross-references to other modules, while types are always serialized with enough info to regenerate them at load time. Nodes are accessed by a file-unique "DeclIDs" (also covering DeclContexts) and "TypeIDs"; the two sets of IDs use separate numbering schemes.

  note: The AST block is currently referred to as the "decls block" in the source.
- The **identifier block** contains a single blob of strings. This is intended for Identifiers—strings uniqued by the ASTContext—but can in theory support any string data. The strings are accessed by a file-unique "IdentifierID".
- The **index block** contains mappings from the AST node and identifier IDs to their offsets in the AST block or identifier block (as appropriate). It also contains various top-level AST information about the module, such as its top-level declarations.

SIL

[to be written]

Cross-reference resilience

[to be written]