# The Swift Driver, Compilation Model, and Command-Line Experience

*or, "why can't I only compile the files that changed?"*

The Swift compiler's command-line interface resembles that of other compilers, particularly GCC and Clang. However, Swift's compilation model and some of its language features make it a bit tricky to plug into a larger build system. In particular, there's no correct way to specify a "one command per file" build rule for a normal Swift module.

("What?" I know! Read on.)

The target audience for this document is people who want to integrate the Swift compiler into their build system, rather than using Xcode or the package manager ( `swift build` ). If you're looking to work on the driver itself...well, this is probably still useful to you, but you should also check out DriverInternals.md and maybe DependencyAnalysis.md as well. If you're just using Xcode or SwiftPM and want to find out what mysterious command-line options you could be passing, `swiftc --help` is a better choice.

If you're invoking `swift -frontend` directly, and you aren't working on the compiler itself...well, this document should convince you to not do that.

Some terms:

- For the purposes of this document, a *module* is a single distributable unit of API. (We use this term for a lot of other things too, though; check out Lexicon.md for the full list.) "Foundation" is a single module, as is the Swift standard library ("Swift"). An app is a module too.
- A *compilation unit* is a set of source files that are compiled together. In Swift, everything intended to be in the same module must be part of the same compilation unit, and every file in the same compilation unit are assumed to be part of the same module. Doing anything else is unsupported.
- The *driver* is the program that's run when you invoke `swift` or `swiftc` . This doesn't actually compile anything itself; instead it invokes other tools to produce the desired output.
- The *frontend* is the program that actually compiles code (and in interpreter mode, executes it via a JIT). It's hidden behind `swift -frontend` .

  The frontend is an implementation detail. No aspects of its command-line interface are considered stable. Even its existence isn't guaranteed.
- The *REPL* is a mode of the debugger (LLDB) that is launched by the driver when you invoke `swift` with no inputs. It doesn't actually come up again in this document, but it's probably worth mentioning for completeness.

## What gets run?

Part of Swift's model is that a file can implicitly see entities declared elsewhere in the same module as long as they have not been marked private. Consequently, compiling any one file needs some knowledge of all the others. However, we don't want a build model that rules out incremental builds, nor one that forces non-parallel compilation. Consequently, the Swift driver has three stages of subprocesses:

1. "Frontend jobs"
2. "Module merging"
3. Linking

Dependencies between the subprocesses are managed by the driver. Outputs are controlled by `-o` and other various compiler flags; see `swiftc --help` for more information.

### Frontend jobs

A normal Swift compilation starts off with as many *frontend jobs* as input files. Each invocation of the Swift frontend parses every file in the module, but also has a particular file marked as the *primary file*. A job is only responsible for compiling its primary file, and only does as much work as it needs to compile that file, lazily type-checking declarations in other files in the module.

A frontend job emits diagnostics, an object file, dependency information (see "Incremental Builds" below), and a *partial module file*.

### Module merging

Swift doesn't have header files; instead it uses a generated binary format called a *Swift module file* (.swiftmodule). With N frontend jobs, it becomes necessary to stitch all the partial module files together. This job is also performed by the frontend in a step called "module merging". Module merging only produces a single merged module file (and accompanying documentation file with the extension .swiftdoc), which can then be imported in later builds.

The Swift module file format is not stable; using it across compiler versions is not supported. It also currently includes a ton of private information about both your module and your compilation environment, so make sure you don't distribute them with your final build products.

### Linking

The last stage of compilation is linking. In addition to actually invoking the linker, the compiler needs to accomplish two other tasks:

- **Autolinking:** Swift object files encode information about what libraries they depend on. On Apple platforms the linker can read this information directly; on other platforms it's extracted using the `swift-autolink-extract` helper tool. Of course the build system can also provide manual link commands too.
- **Debugging:** In addition to the usual DWARF format, interactive debugging of a Swift program requires access to its module. This is supported via a custom linker flag on Apple platforms and by the `swift-modulewrap` tool elsewhere. Apple platforms also invoke `dsymutil` after linking to produce a dSYM debugging archive, so that the program is still debuggable even once the object files have been cleaned.

## Output File Maps

> *There are three numbers in computer science: 0, 1, and N.*

The problem with running a separate frontend job for each input file is that it suddenly becomes much more complicated to determine where outputs go. If you're just going to link everything together at the end this isn't really a problem, but a lot of build systems just want the compiler to produce object files. If you want to use Swift's cross-file dependency tracking, or get Clang-style serialized diagnostics (.dia files) instead of just text output, those are also generated per-input-file. And some build systems (*cough*Xcode*cough*) want to individually track persisted output files, instead of dumping them into a temporary directory. Trying to specify the outputs for every input file on the command line would be awkward and ridiculous, so instead we use an *output file map*.

An output file map contains a JSON dictionary that looks like this:

```
{
  "/path/to/src/foo.swift": {
    "object": "/path/to/build/foo.o",
    "dependencies": "/path/to/build/foo.d",
    "swift-dependencies": "/path/to/build/foo.swiftdeps",
    "diagnostics": "/path/to/build/foo.dia"
  },
  "/path/to/src/bar.swift": {
    "object": "/path/to/build/bar.o",
    "dependencies": "/path/to/build/bar.d",
    "swift-dependencies": "/path/to/build/bar.swiftdeps",
    "diagnostics": "/path/to/build/bar.dia"
  },
  "": {
    "swift-dependencies": "/path/to/build/main-build-record.swiftdeps"
  }
}
```

The build system is responsible for generating this file. The input file names don't have to be absolute paths, but they do have to match the form used in the invocation of the compiler. No canonicalization is performed. The special `""` entry is used for outputs that apply to the entire build.

The "dependencies" entries refer to Make-style dependency files (similar to GCC and Clang's `-MD` option), which can be used to track cross-module and header file dependencies. "swift-dependencies" entries are required to perform incremental builds (see below). The "diagnostics" entries are only for your own use; if you don't need Clang-style serialized diagnostics they can be omitted.

The output file map accepts other entries, but they should not be considered stable. Please stick to what's shown here.

(Note: In the example output file map above, all of the per-file outputs are being emitted to the same directory. SR-327 covers adding a flag that would infer this behavior given a directory path.)

## Whole-Module Optimization

When the `-whole-module-optimization` flag is passed to Swift, the compilation model changes significantly. In this mode, the driver only invokes the frontend once, and there is no primary file. Instead, every file is parsed and type-checked, and then all generated code is optimized together.

Whole-module builds actually have two modes: threaded and non-threaded. The default is non-threaded, which produces a single object file for the entire compilation unit (.o). If you're just producing object files, you can use the usual `-o` option with `-c` to control where the single output goes.

In threaded mode, one object file is produced for every input source file, and the "backend" processing of the generated code (basically, everything that's not Swift-language-specific) is performed on multiple threads. Like non-whole-module compilation, the locations of the object files is controlled by the output file map. Only one file is produced for each non-object output, however. The location of these additional outputs is controlled by command-line options, except for Make-style dependencies and serialized diagnostics, which use an entry under `""` in the output file map.

Threaded mode is controlled by the `-num-threads` command-line option rather than `-j` used to control the number of jobs (simultaneous subprocesses spawned by the driver). Why? While changing the number of jobs

should never affect the final product, using threaded vs. non-threaded compilation does.

Specifics of whole-module optimization mode are subject to change, particularly in becoming more like non-whole-module builds.

## Incremental Builds

Incremental builds in Swift work primarily by cross-file dependency analysis, described in DependencyAnalysis.md. Compiling a single file might be necessary because that file has changed, but it could also be because that file depends on something else that might have changed. From a build system perspective, the files in a particular module can't be extracted from each other; a top-level invocation of the compiler will result in a valid compilation of the entire module, but manually recompiling certain files is not guaranteed to do anything sensible.

Performing an incremental build is easy; just pass `-incremental` and be sure to put "swift-dependencies" entries in your output file map.

Incremental builds don't mix with whole-module builds, which get their whole-module-ness by rebuilding everything every time *just in case* there's some extra optimizations that can be made by repeated inlining, or by *really* being sure that a method is never overridden.

(Can this sort of information be cached? Of course. But there's a question of whether it's going to be easier to make whole-module builds be more incremental or to make incremental builds support more cross-file optimization.)

A particular nasty bit about incremental builds: because in general the compiler only knows that a file *has* changed and not *what* changed, it's possible that the driver will start off assuming that some file won't need to be recompiled, and then discover later on that it should be. This means that it's not possible to use classic build models like Makefiles to make builds incremental, because Makefiles don't accommodate dependency graph changes during the build.

Only the "frontend" tasks are considered skippable; module-merging and linking always occurs. (It is assumed that if *none* of the inputs changed, the build system wouldn't have invoked the compiler at all.)

Currently Swift does not provide any way to report tasks to be compiled out to some other build system. This is definitely something we're interested in, though---it would enable better integration between the compiler and llbuild (the build system used by the package manager), which would lead to faster compile times when many dependencies were involved.

The format of the "swiftdeps" files used to track cross-file dependencies should be considered fragile and opaque; it is not guaranteed to remain stable across compiler versions.

### `-embed-bitcode`

Apple's "Embed Bitcode" feature adds an extra bit of complexity: to be sure builds from the embedded bitcode are identical to those built locally, the driver actually splits compilation in two. The first half of compilation produces LLVM IR, then the driver runs so-called *backend jobs,* which compile the IR the rest of the way into binaries.

There's not really any secrets here, since it's all there in the source, but the particular mechanism isn't guaranteed by any means---Apple could decide to change it whenever they want. It's probably best not to special-case any logic here; just using the driver as intended should work fine.

## So, how should I work Swift into my build system?

Step 0 is to see if you can use the Swift package manager instead. If so, it's mostly just `swift build`. But if you're reading this document you're probably past that, so:

1. Generate an output file map that contains all the per-file outputs you care about. Most likely this is just the object files and incremental build dependency files; everything else is an intermediate. (There should probably be a tool that does this, perhaps built on what the package manager does.)

2. Set TMPDIR to somewhere you don't mind uninteresting intermediate files going.

3. Do one of the following:

   - Invoke `swiftc -emit-executable` or `swiftc -emit-library`. Pass all the Swift files, even if you're building incrementally (`-incremental`). Pass `-j <N>` and/or `-num-threads <N>` if you have cores to spare. Pass `-emit-module-path <path>` if you're building a library that you need to import later.

     If you want debugging that's more than `-gline-tables-only`, this is the only supported way to do it today. (SR-2637 and SR-2660 are aimed at improving on this.) On the plus side, this mode doesn't strictly need an output file map if you give up incremental builds.

   - Invoke `swiftc -c`, then pass the resulting object files to your linker. All the same options from above apply, but you'll have to manually deal with the work the compiler would have done automatically for you.

   Whatever you do, do *not* invoke the frontend directly.

### Questions

*Can I link all the object files together in the same binary, even if they came from multiple modules?*

This is not currently supported, and debugging probably won't work. (See SR-2637 and SR-2660 for more details.) However, if you are using `-gnone` or `-gline-tables-only`, the worst you will suffer is more symbols being visible than are strictly necessary.