How to Set Up an Edit-Build-Test-Debug Loop

This document describes how to set up a development loop for people interested in contributing to Swift.

If you are only interested in building the toolchain as a one-off, there are a couple of differences:

1. You can ignore the parts related to Sccache.
2. You can stop reading after Building the project for the first time.

Table of Contents

System Requirements

1. Operating system: The supported operating systems for developing the Swift toolchain are: macOS, Ubuntu Linux LTS, and the latest Ubuntu Linux release. At the moment, Windows is not supported as a host development operating system. Experimental instructions for Windows are available under Windows.md.
2. Python 3: Several utility scripts are written in Python.
3. Disk space: Make sure that you have enough available disk space before starting. The source code, including full git history, requires about 3.5 GB. Build artifacts take anywhere between 5 GB to 70 GB, depending on the build settings.
4. Time: Depending on your machine and build settings, a from-scratch build can take a few minutes to several hours, so you might want to grab a beverage while you follow the instructions. Incremental builds are much faster.

Cloning the project

1. Create a directory for the whole project:

   ```
   mkdir swift-project
   cd swift-project
   ```

2. Clone the sources:
   - Via SSH (recommended): If you plan on contributing regularly, cloning over SSH provides a better experience. After you've uploaded your SSH keys to GitHub:

     ```
     git clone git@github.com:apple/swift.git swift
     cd swift
     utils/update-checkout --clone-with-ssh
     ```

   - Via HTTPS: If you want to check out the sources as read-only, or are not familiar with setting up SSH, you can use HTTPS instead:

     ```
     git clone https://github.com/apple/swift.git swift
     cd swift
     utils/update-checkout --clone
     ```

     Note: If you've already forked the project on GitHub at this stage, **do not clone your fork** to start off. We describe how to setup your fork in a subsection below.

3. Double-check that `swift`'s sibling directories are present.

   ```
   ls ..
   ```

   This should list directories like `llvm-project`, `swiftpm` and so on.

4. Checkout the right branch/tag: If you are building the toolchain for local development, you can skip this step, as Step 2 will checkout `swift`'s `main` branch and matching branches for other projects. If you are building the toolchain as a one-off, it is more likely that you want a specific branch or a tag, often corresponding to a specific release or a specific snapshot. You can update the branch/tag for all repositories as follows:

   ```
   utils/update-checkout --scheme mybranchname
   # OR
   utils/update-checkout --tag mytagname
   ```

   Detailed branching information, including names for release branches, can be found in Branches.md.

Note: The commands used in the rest of this guide assumes that the absolute path to your working directory is something like `/path/to/swift-project/swift`. Double-check that running `pwd` prints a path ending with `swift`.

Troubleshooting cloning issues

- If `update-checkout` failed, double-check that the absolute path to your working directory does not have non-ASCII characters.

- If `update-checkout` failed and the absolute path to your working directory had spaces in it, please file a bug report and change the path to work around it.
- Before running `update-checkout`, double-check that `swift` is the only repository inside the `swift-project` directory. Otherwise, `update-checkout` may not clone the necessary dependencies.

Installing dependencies

macOS

⚠ Since version 0.2.14, `sccache` no longer caches compile commands issued by `build-script` because of sccache PR 898, since `build-script` adds the `-arch x86_64` argument twice. The instructions below may install `sccache` 0.2.14 or newer. You may want to instead download and install an older release from their Releases page until this issue is resolved.

1. Install Xcode 13 beta 4 or newer: The required version of Xcode changes frequently and is often a beta release. Check this document or the host information on https://ci.swift.org for the current required version.
2. Install CMake, Ninja and Sccache:
   - Via Homebrew (recommended):

     ```
     brew install cmake ninja sccache
     ```

   - Via Homebrew Bundle:

     ```
     brew bundle
     ```

Linux

1. The latest Linux dependencies are listed in the respective Dockerfiles:
   - Ubuntu 20.04
   - CentOS 7
   - CentOS 8
   - Amazon Linux 2
2. To install sccache (optional):

   ```
   sudo snap install sccache --candidate --classic
   ```

   Note: LLDB currently requires at least `swig-1.3.40` but will successfully build with version 2 shipped with Ubuntu.

Building the project for the first time

Spot check dependencies

- Run `cmake --version`: This should be 3.19.6 or higher.
- Run `python3 --version`: Check that this succeeds.
- Run `ninja --version`: Check that this succeeds.
- Run `sccache --version`: Check that this succeeds.

The roles of different tools

At this point, it is worthwhile to pause for a moment to understand what the different tools do:

1. On macOS and Windows, IDEs (Xcode and Visual Studio resp.) serve as an easy way to install development dependencies such as a C++ compiler, a linker, header files, etc. The IDE's build system need not be used to build Swift. On Linux, these dependencies are installed by the distribution's package manager.
2. CMake is a cross-platform build system for C and C++. It forms the core infrastructure used to *configure* builds of Swift and its companion projects.
3. Ninja is a low-level build system that can be used to *build* the project, as an alternative to Xcode's build system. Ninja is somewhat faster, especially for incremental builds, and supports more build environments.
4. Sccache is a caching tool: If you ever delete your build directory and rebuild from scratch (i.e. do a "clean build"), Sccache can accelerate the new build significantly. There are few things more satisfying than seeing Sccache cut through build times.
5. `utils/update-checkout` is a script to help you work with all the individual git repositories together, instead of manually cloning/updating each one.
6. `utils/build-script` (we will introduce this shortly) is a high-level automation script that handles configuration (via CMake), building (via Ninja or Xcode), caching (via Sccache), running tests and more.

> ***Pro Tip***: *Most tools support `--help` flags describing the options they support. Additionally, both Clang and the Swift compiler have hidden flags (`clang --help-hidden` / `swiftc --help-hidden`) and frontend flags (`clang -cc1 --help` / `swiftc -frontend --help`) and the Swift compiler even has hidden frontend flags (`swiftc -frontend --help-hidden`). Sneaky!*

Phew, that's a lot to digest! Now let's proceed to the actual build itself!

The actual build

1. Make sure you have Sccache running.

   ```
   sccache --start-server
   ```

   (Optional) Sccache defaults to a cache size of 10GB, which is relatively small compared to build artifacts. You can bump it up, say by setting `export SCCACHE_CACHE_SIZE="50G"` in your dotfile(s). For more details, see the Sccache README.

2. Decide if you would like to build the toolchain using Ninja or using Xcode.

   - If you use an editor other than Xcode and/or you want somewhat faster builds, go with Ninja.
   - If you are comfortable with using Xcode and would prefer to use it, go with Xcode. There is also a third option, which is somewhat more involved: using both Ninja and Xcode.

3. Build the toolchain with optimizations, debuginfo, and assertions and run the tests. macOS:

   - Via Ninja:

     ```
     utils/build-script --skip-build-benchmarks \
       --skip-ios --skip-watchos --skip-tvos --swift-darwin-supported-archs
     "$(uname -m)" \
       --sccache --release-debuginfo --swift-disable-dead-stripping --test
     ```

   - Via Xcode:

```
utils/build-script --skip-build-benchmarks \
  --skip-ios --skip-watchos --skip-tvos --swift-darwin-supported-archs
"$(uname -m)" \
  --sccache --release-debuginfo --swift-disable-dead-stripping --test
\
  --xcode
```

Linux (uses Ninja):

```
utils/build-script --release-debuginfo --test --skip-early-swift-
driver
```

This will create a directory `swift-project/build/Ninja-RelWithDebInfoAssert` (with `Xcode` instead of `Ninja` if you used `--xcode`) containing the Swift compiler and standard library and clang/LLVM build artifacts.

- If the build succeeds: Once the build is complete, the tests will run.
  - If the tests are passing: Great! We can go to the next step.
  - If some tests are failing:
    - Consider filing a bug report.
    - Note down which tests are failing as a baseline. This baseline will be handy later when you run the tests after making a change.
- If the build fails: See Troubleshooting build issues.

If you would like to additionally build the Swift corelibs, ie swift-corelibs-libdispatch, swift-corelibs-foundation, and swift-corelibs-xctest, on Linux, add the `--xctest` flag to `build-script`.

In the following sections, for simplicity, we will assume that you are using a `Ninja-RelWithDebInfoAssert` build on macOS running on an Intel-based Mac, unless explicitly mentioned otherwise. You will need to slightly tweak the paths for other build configurations.

Using both Ninja and Xcode

Some contributors find it more convenient to use both Ninja and Xcode. Typically this configuration consists of:

1. A Ninja build created with `--release-debuginfo`.
2. An Xcode build created with `--release-debuginfo --debug-swift`.

The Ninja build can be used for fast incremental compilation and running tests quickly. The Xcode build can be used for debugging with high fidelity.

The additional flexibility comes with two issues: (1) consuming much more disk space and (2) you need to maintain the two builds in sync, which needs extra care when moving across branches.

Troubleshooting build issues

- Double-check that all projects are checked out at the right branches. A common failure mode is using `git checkout` to change the branch only for `swift` (often to a release branch), leading to an unsupported configuration. See Step 4 of Cloning the Project on how to fix this.
- Double-check that all your dependencies meet the minimum required versions.
- Check if there are spaces in the paths being used by `build-script` in the log. While `build-script` should work with paths containing spaces, sometimes bugs do slip through, such as SR-13441. If this is the

case, please file a bug report and change the path to work around it.

- Check that your `build-script` invocation doesn't have typos. You can compare the flags you passed against the supported flags listed by `utils/build-script --help`.
- Check the error logs and see if there is something you can fix. In many situations, there are several errors, so scrolling further back and looking at the first error may be more helpful than simply looking at the last error.
- Check if others have encountered the same issue on the Swift forums or on Swift JIRA.
- Create a new Swift forums thread in the Development category. Include information about your configuration and the errors you are seeing.
    - You can create a gist with the entire build output and link it, while highlighting the most important part of the build log in the post.
    - Include the output of `utils/update-checkout --dump-hashes`.

Editing code

Setting up your fork

If you are building the toolchain for development and submitting patches, you will need to setup a GitHub fork.

First fork the `apple/swift` repository, using the "Fork" button in the web UI, near the top-right. This will create a repository `username/swift` for your GitHub username. Next, add it as a remote:

```
# Using 'my-remote' as a placeholder name.

# If you set up SSH in step 2
git remote add my-remote git@github.com:username/swift.git

# If you used HTTPS in step 2
git remote add my-remote https://github.com/username/swift.git
```

Finally, create a new branch.

```
# Using 'my-branch' as a placeholder name
git checkout -b my-branch
git push --set-upstream my-remote my-branch
```

First time Xcode setup

If you used `--xcode` earlier, you will see an Xcode project generated under `../build/Xcode-RelWithDebInfoAssert/swift-macosx-x86_64` (or `../build/Xcode-RelWithDebInfoAssert/swift-macosx-arm64` on Apple Silicon Macs). When you open the project, Xcode might helpfully suggest "Automatically Create Schemes". Most of those schemes are not required in day-to-day work, so you can instead manually select the following schemes:

- `swift-frontend`: If you will be working on the compiler.
- `check-swift-all`: This can be used to run the tests. The test runner does not integrate with Xcode though, so it may be easier to run tests directly on the commandline for more fine-grained control over which exact tests are run.

Other IDEs setup

You can also use other editors and IDEs to work on Swift.

IntelliJ CLion

CLion supports CMake and Ninja. In order to configure it properly, build the swift project first using the `build-script`, then open the `swift` directory with CLion and proceed to project settings (`cmd + ,`).

In project settings, locate `Build, Execution, Deployment > CMake`. You will need to create a new profile named `RelWithDebInfoAssert` (or `Debug` if going to point it at the debug build). Enter the following information:

- Name: mirror the name of the build configuration here, e.g. `RelWithDebInfoAssert` or `Debug`
- Build type: This corresponds to `CMAKE_BUILD_TYPE` so should be e.g. `RelWithDebInfoAssert` or `Debug`
  - latest versions of the IDE suggest valid values here. Generally `RelWithDebInfoAssert` is a good one to work with
- Toolchain: Default should be fine
- Generator: Ninja
- CMake options:
  - `-D SWIFT_PATH_TO_CMARK_BUILD=SOME_PATH/swift-project/build/Ninja-RelWithDebInfoAssert/cmark-macosx-arm64 -D LLVM_DIR=SOME_PATH/swift-project/build/Ninja-RelWithDebInfoAssert/llvm-macosx-arm64/lib/cmake/llvm -D Clang_DIR=SOME_PATH/swift-project/build/Ninja-RelWithDebInfoAssert/llvm-macosx-arm64/lib/cmake/clang -D CMAKE_BUILD_TYPE=RelWithDebInfoAssert -G Ninja -S .`
  - replace the `SOME_PATH` to the path where your `swift-project` directory is
  - the CMAKE_BUILD_TYPE should match the build configuration name, so if you named this profile `RelWithDebInfo` the CMAKE_BUILD_TYPE should also be `RelWithDebInfo`

With this done, CLion should be able to successfully import the project and have full autocomplete and code navigation powers.

Editing

Make changes to the code as appropriate. Implement a shiny new feature! Or fix a nasty bug! Update the documentation as you go! The codebase is your oyster!

:construction::construction_worker::building_construction:

Now that you have made some changes, you will need to rebuild...

Incremental builds with Ninja

To rebuild the compiler:

```
ninja -C ../build/Ninja-RelWithDebInfoAssert/swift-macosx-$(uname -m) swift-frontend
```

To rebuild everything, including the standard library:

```
ninja -C ../build/Ninja-RelWithDebInfoAssert/swift-macosx-$(uname -m)
```

Incremental builds with Xcode

Rebuilding works the same way as with any other Xcode project; you can use ⌘+B or Product → Build.

Spot checking an incremental build

As a quick test, go to `lib/Basic/Version.cpp` and tweak the version printing code slightly. Next, do an incremental build as above. This incremental build should be much faster than the from-scratch build at the beginning. Now check if the version string has been updated:

```
../build/Ninja-RelWithDebInfoAssert/swift-macosx-$(uname -m)/bin/swift-frontend --
version
```

This should print your updated version string.

Reproducing an issue

Starter bugs typically have small code examples that fit within a single file. You can reproduce such an issue in various ways, such as compiling it from the commandline using `/path/to/swiftc MyFile.swift`, pasting the code into Compiler Explorer (aka godbolt) or using an Xcode Playground.

For files using frameworks from an SDK bundled with Xcode, you need the pass the SDK explicitly. Here are a couple of examples:

```
# Compile a file to an executable for your local machine.
xcrun -sdk macosx /path/to/swiftc MyFile.swift

# Say you are trying to compile a file importing an iOS-only framework.
xcrun -sdk iphoneos /path/to/swiftc -target arm64-apple-ios13.0 MyFile.swift
```

You can see the full list of `-sdk` options using `xcodebuild -showsdks`, and check some potential `-target` options for different operating systems by skimming the compiler's test suite under `test/`.

Sometimes bug reports come with SwiftPM packages or Xcode projects as minimal reproducers. While we do not add packages or projects to the compiler's test suite, it is generally helpful to first reproduce the issue in context before trying to create a minimal self-contained test case. If that's the case with the bug you're working on, check out our instructions on building packages and Xcode projects with a locally built compiler.

Running tests

There are two main ways to run tests:

1. `utils/run-test`: By default, `run-test` builds the tests' dependencies before running them.

   ```
   # Rebuild all test dependencies and run all tests under test/.
   utils/run-test --lit ../llvm-project/llvm/utils/lit/lit.py \
     ../build/Ninja-RelWithDebInfoAssert/swift-macosx-$(uname -m)/test-
   macosx-$(uname -m)

   # Rebuild all test dependencies and run tests containing "MyTest".
   utils/run-test --lit ../llvm-project/llvm/utils/lit/lit.py \
   ```

```
  ../build/Ninja-RelWithDebInfoAssert/swift-macosx-$(uname -m)/test-
macosx-$(uname -m) \
  --filter="MyTest"
```

2. `lit.py` : lit doesn't know anything about dependencies. It just runs tests.

```
# Run all tests under test/.
../llvm-project/llvm/utils/lit/lit.py -s -vv \
  ../build/Ninja-RelWithDebInfoAssert/swift-macosx-$(uname -m)/test-
macosx-$(uname -m)

# Run tests containing "MyTest"
../llvm-project/llvm/utils/lit/lit.py -s -vv \
  ../build/Ninja-RelWithDebInfoAssert/swift-macosx-$(uname -m)/test-
macosx-$(uname -m) \
  --filter="MyTest"
```

   The `-s` and `-vv` flags print a progress bar and the executed commands respectively.

If you are making small changes to the compiler or some other component, you'll likely want to incrementally rebuild only the relevant Ninja/Xcode target and use `lit.py` with `--filter`. One potential failure mode with this approach is accidental use of stale binaries. For example, say that you want to rerun a SourceKit test but you only incrementally rebuilt the compiler. Then your changes will not be reflected when the test runs because the `sourcekitd` binary was not rebuilt. Using `run-test` instead is the safer option, but it will lead to a longer feedback loop due to more things getting rebuilt.

If you want to rerun all the tests, you can either rebuild the whole project and use `lit.py` without `--filter` or use `run-test` to handle both aspects.

Recall the baseline failures mentioned in the build section. If your baseline had failing tests, make sure you compare the failures seen after your changes to the baseline. If some test failures look totally unrelated to your changes, there is a good chance that they were already failing as part of the baseline.

For more details on running tests and understanding the various Swift-specific lit customizations, see Testing.md. Also check out the lit documentation to understand how the different lit commands work.

Debugging issues

In this section, we briefly describe two common ways of debugging: print debugging and using LLDB.

Depending on the code you're interested in, LLDB may be significantly more effective when using a debug build. Depending on what components you are working on, you could turn off optimizations for only a few things. Here are some example invocations:

```
# optimized Stdlib + debug Swiftc + optimized Clang/LLVM
utils/build-script --release-debuginfo --debug-swift # other flags...

# debug Stdlib + optimized Swiftc + optimized Clang/LLVM
utils/build-script --release-debuginfo --debug-swift-stdlib # other flags...

# optimized Stdlib + debug Swiftc (except typechecker) + optimized Clang/LLVM
utils/build-script --release-debuginfo --debug-swift --force-optimized-typechecker
```

```
# Last resort option, it is highly unlikely that you will need this
# debug Stdlib + debug Swiftc + debug Clang/LLVM
utils/build-script --debug # other flags...
```

Debug builds have two major drawbacks:

- A debug compiler is much slower, leading to longer feedback loops in case you need to repeatedly compile the Swift standard library and/or run a large number of tests.
- The build artifacts consume a lot more disk space.

DebuggingTheCompiler.md goes into a LOT more detail on how you can level up your debugging skills! Make sure you check it out in case you're trying to debug a tricky issue and aren't sure how to go about it.

Print debugging

A large number of types have `dump(..)` / `print(..)` methods which can be used along with `llvm::errs()` or other LLVM streams. For example, if you have a variable `std::vector<CanType> canTypes` that you want to print, you could do:

```
auto &e = llvm::errs();
e << "canTypes = [";
llvm::interleaveComma(canTypes, e, [&](auto ty) { ty.dump(e); });
e << "]\n";
```

You can also crash the compiler using `assert` / `llvm_unreachable` / `llvm::report_fatal_error`, after accumulating the result in a stream:

```
std::string msg; llvm::raw_string_ostream os(msg);
os << "unexpected canTypes = [";
llvm::interleaveComma(canTypes, os, [&](auto ty) { ty.dump(os); });
os << "] !!!\n";
llvm::report_fatal_error(os.str());
```

Debugging using LLDB

When the compiler crashes, the commandline arguments passed to it will be printed to stderr. It will likely look something like:

```
/path/to/swift-frontend <args>
```

- Using LLDB on the commandline: Copy the entire invocation and pass it to LLDB.

  ```
  lldb -- /path/to/swift-frontend <args>
  ```

  Now you can use the usual LLDB commands like `run`, `breakpoint set` and so on. If you are new to LLDB, check out the official LLDB documentation and nesono's LLDB cheat sheet.

- Using LLDB within Xcode: Select the current scheme 'swift-frontend' → Edit Scheme → Run phase → Arguments tab. Under "Arguments Passed on Launch", copy-paste the `<args>` and make sure that

"Expand Variables Based On" is set to swift-frontend. Close the scheme editor. If you now run the compiler (⌘+R or Product → Run), you will be able to use the Xcode debugger.

Xcode also has the ability to attach to and debug Swift processes launched elsewhere. Under Debug → Attach to Process by PID or name..., you can enter a compiler process's PID or name (`swift-frontend`) to debug a compiler instance invoked elsewhere. This can be helpful if you have a single compiler process being invoked by another tool, such as SwiftPM or another open Xcode project.

> ***Pro Tip**: Xcode 12's terminal does not support colors, so you may see explicit color codes printed by `dump()` methods on various types. To avoid color codes in dumped output, run `expr llvm::errs().enable_color(false)` .*

Next steps

Make sure you check out the following resources:

- LLVM Coding Standards: A style guide followed by both LLVM and Swift. If there is a mismatch between the LLVM Coding Standards and the surrounding code that you are editing, please match the style of existing code.
- LLVM Programmer's Manual: A guide describing common programming idioms and data types used by LLVM and Swift.
- docs/README.md: Provides a bird's eye view of the available documentation.
- Lexicon.md: Provides definitions for jargon. If you run into a term frequently that you don't recognize, it's likely that this file has a definition for it.
- Testing.md and DebuggingTheCompiler.md: These cover more ground on testing and debugging respectively.
- Development Tips: Tips for being more productive.

If you see mistakes in the documentation (including typos, not just major errors) or identify gaps that you could potentially improve the contributing experience, please start a discussion on the forums, submit a pull request or file a bug report on Swift JIRA. Thanks!