Zig Build System ⚡ Zig Programming Language

[Download](#) [Learn](#) [News](#) [Source Code](#) [Join a Community](#) 💗[Sponsor the Zig Software Foundation](#)
← Back to the **Learn** section

Zig Build System

When to bust out the Zig Build System?

The fundamental commands `zig build-exe`, `zig build-lib`, `zig build-obj`, and `zig test` are often sufficient. However, sometimes a project needs another layer of abstraction to manage the complexity of building from source.

For example, perhaps one of these situations applies:

- The command line becomes too long and unwieldly, and you want some place to write it down.
- You want to build many things, or the build process contains many steps.
- You want to take advantage of concurrency and caching to reduce build time.
- You want to expose configuration options for the project.
- The build process is different depending on the target system and other options.
- You have dependencies on other projects.
- You want to avoid an unnecessary dependency on cmake, make, shell, msvc, python, etc., making the project accessible to more contributors.
- You want to provide a package to be consumed by third parties.
- You want to provide a standardized way for tools such as IDEs to semantically understand how to build the project.

If any of these apply, the project will benefit from using the Zig Build System.

Getting Started

Simple Executable

This build script creates an executable from a Zig file that contains a public main function definition.

hello.zig

```
const std = @import("std");

pub fn main() !void {
    std.debug.print("Hello World!\n", .{});
}
```

build.zig

```
const std = @import("std");

pub fn build(b: *std.Build) void {
    const exe = b.addExecutable(.{
        .name = "hello",
        .root_source_file = .{ .path = "hello.zig" },
        .target = b.host,
    });

    b.installArtifact(exe);
}
```

```
$ zig build --summary all
Build Summary: 3/3 steps succeeded
install success
└─ install hello success
   └─ zig build-exe hello Debug native success 3s MaxRSS:173M
```

Installing Build Artifacts

The Zig build system, like most build systems, it is based on modeling the project as a directed acyclic graph (DAG) of steps, which are independently and concurrently run.

By default, the main step in the graph is the **Install** step, whose purpose is to copy build artifacts into their final resting place. The Install step starts with no dependencies, and therefore nothing will happen when `zig build` is run. A project's build script must add to the set of things to install, which is what the `installArtifact` function call does above.

Output

```
├── build.zig
├── hello.zig
├── zig-cache
└── zig-out
    └── bin
        └── hello
```

There are two generated directories in this output: `zig-cache` and `zig-out`. The first one contains files that will make subsequent builds faster, but these files are not intended to be checked into source-control and this directory can be completely deleted at any time with no consequences.

The second one, `zig-out`, is an "installation prefix". This maps to the standard file system hierarchy concept. This directory is not chosen by the project, but by the user of `zig build` with the `--prefix` flag (`-p` for short).

You, as the project maintainer, pick what gets put in this directory, but the user chooses where to install it in their system. The build script cannot hardcode output paths because this would break caching, concurrency, and composability, as well as annoy the final user.

Adding a Convenience Step for Running the Application

It is common to add a `Run` step to provide a way to run one's main application directly from the build command.

hello.zig

```
const std = @import("std");

pub fn main() !void {
    std.debug.print("Hello World!\n", .{});
}
```

build.zig

```
const std = @import("std");

pub fn build(b: *std.Build) void {
    const exe = b.addExecutable(.{
        .name = "hello",
        .root_source_file = .{ .path = "hello.zig" },
        .target = b.host,
    });

    b.installArtifact(exe);

    const run_exe = b.addRunArtifact(exe);

    const run_step = b.step("run", "Run the application");
    run_step.dependOn(&run_exe.step);
}
```

```
$ zig build run --summary all
Hello World!
Build Summary: 3/3 steps succeeded
run success
└─ run hello success 10ms MaxRSS:3M
   └─ zig build-exe hello Debug native success 3s MaxRSS:173M
```

The Basics

User-Provided Options

Use `b.option` to make the build script configurable to end users as well as other projects that depend on the project as a package.

build.zig

```
const std = @import("std");

pub fn build(b: *std.Build) void {
    const windows = b.option(bool, "windows", "Target Microsoft Windows")
orelse false;

    const exe = b.addExecutable(.{
        .name = "hello",
        .root_source_file = .{ .path = "example.zig" },
        .target = b.resolveTargetQuery(.{
            .os_tag = if (windows) .windows else null,
        }),
    });

    b.installArtifact(exe);
}
```

```
$ zig build --help

Usage: /home/runner/work/www.ziglang.org/www.ziglang.org/zig/zig build
[steps] [options]

Steps:
  install (default)            Copy build artifacts to prefix path
  uninstall                    Remove build artifacts from prefix path

General Options:
```

```
General Options:
  -p, --prefix [path]          Override default install prefix
  --prefix-lib-dir [path]      Override default library directory path
  --prefix-exe-dir [path]      Override default executable directory path
  --prefix-include-dir [path]  Override default include directory path

  --sysroot [path]             Set the system root directory (usually /)
  --search-prefix [path]       Add a path to look for binaries, libraries,
headers
  --libc [file]                Provide a file which specifies libc paths

  -fdarling,  -fno-darling     Integration with system-installed Darling
to
                               execute macOS programs on Linux hosts
                               (default: no)
  -fqemu,     -fno-qemu        Integration with system-installed QEMU to
execute
                               foreign-architecture programs on Linux
hosts
                               (default: no)
  --glibc-runtimes [path]      Enhances QEMU integration by providing
glibc built
                               for multiple foreign architectures,
allowing
                               execution of non-native programs that link
with glibc.
  -frosetta,  -fno-rosetta     Rely on Rosetta to execute x86_64 programs
on
                               ARM64 macOS hosts. (default: no)
  -fwasmtime, -fno-wasmtime    Integration with system-installed wasmtime
to
                               execute WASI binaries. (default: no)
  -fwine,     -fno-wine        Integration with system-installed Wine to
execute
                               Windows programs on Linux hosts. (default:
no)

  -h, --help                   Print this help and exit
  -l, --list-steps             Print available steps
  --verbose                    Print commands before executing them
  --color [auto|off|on]        Enable or disable colored error messages
  --prominent-compile-errors   Buffer compile errors and display at end
  --summary [mode]             Control the printing of the build summary
    all                        Print the build summary in its entirety
    failures                   (Default) Only print failed steps
    none                       Do not print the build summary
  -j<N>                        Limit concurrent jobs (default is to use
all CPU cores)
  --maxrss <bytes>             Limit memory usage (default is to use
available memory)
  --skip-oom-steps             Instead of failing, skip steps that would
exceed --maxrss
  --fetch                      Exit after fetching dependency tree

Project-Specific Options:
  -Dwindows=[bool]             Target Microsoft Windows

Advanced Options:
  -freference-trace[=num]      How many lines of reference trace should be
shown per compile error
  -fno-reference-trace         Disable reference trace
  --build-file [file]          Override path to build.zig
  --cache-dir [path]           Override path to local Zig cache directory
  --global-cache-dir [path]    Override path to global Zig cache
directory
  --zig-lib-dir [arg]          Override path to Zig lib directory
  --build-runner [file]        Override path to build runner
  --seed [integer]             For shuffling dependency traversal order
(default: random)
  --debug-log [scope]          Enable debugging the compiler
  --debug-pkg-config           Fail if unknown pkg-config flags
encountered
  --verbose-link               Enable compiler debug output for linking
  --verbose-air                Enable compiler debug output for Zig AIR
  --verbose-llvm-ir[=file]     Enable compiler debug output for LLVM IR
  --verbose-llvm-bc=[file]     Enable compiler debug output for LLVM BC
  --verbose-cimport            Enable compiler debug output for C imports
  --verbose-cc                 Enable compiler debug output for C
compilation
  --verbose-llvm-cpu-features  Enable compiler debug output for LLVM CPU
```

```
features
```

example.zig

```
const std = @import("std");

pub fn main() !void {
    std.debug.print("Hello World!\n", .{});
}
```

Please direct your attention to these lines:

```
Project-Specific Options:
  -Dwindows=[bool]             Target Microsoft Windows
```

This part of the help menu is auto-generated based on running the `build.zig` logic. Users can discover configuration options of the build script this way.

Standard Configuration Options

Previously, we used a boolean flag to indicate building for Windows. However, we can do better.

Most projects want to provide the ability to change the target and optimization settings. In order to encourage standard naming conventions for these options, Zig provides the helper functions, `standardTargetOptions` and `standardOptimizeOption`.

Standard target options allows the person running `zig build` to choose what target to build for. By default, any target is allowed, and no choice means to target the host system. Other options for restricting supported target set are available.

Standard optimization options allow the person running `zig build` to select between `Debug`, `ReleaseSafe`, `ReleaseFast`, and `ReleaseSmall`. By default none of the release options are considered the preferable choice by the build script, and the user must make a decision in order to create a release build.

hello.zig

```
const std = @import("std");

pub fn main() !void {
    std.debug.print("Hello World!\n", .{});
}
```

build.zig

```
const std = @import("std");

pub fn build(b: *std.Build) void {
    const target = b.standardTargetOptions(.{});
    const optimize = b.standardOptimizeOption(.{});
    const exe = b.addExecutable(.{
        .name = "hello",
        .root_source_file = .{ .path = "hello.zig" },
        .target = target,
        .optimize = optimize,
    });

    b.installArtifact(exe);
}
```

```
$ zig build -Dtarget=x86_64-windows -Doptimize=ReleaseSmall --summary all
Build Summary: 3/3 steps succeeded
install success
└─ install hello success
   └─ zig build-exe hello ReleaseSmall x86_64-windows success 2s
MaxRSS:143M
```

Now, our `--help` menu contains more items:

```
Project-Specific Options:
  -Dtarget=[string]            The CPU architecture, OS, and ABI to build
for
  -Dcpu=[string]               Target CPU features to add or subtract
  -Doptimize=[enum]            Prioritize performance, safety, or binary
size (-O flag)
                                 Supported Values:
                                   Debug
                                   ReleaseSafe
                                   ReleaseFast
                                   ReleaseSmall
```

It is entirely possible to create these options via `b.option` directly, but this API provides a commonly used naming convention for these frequently used settings.

In our terminal output, observe that we passed `-Dtarget=x86_64-windows -Doptimize=ReleaseSmall`. Compared to the first example, now we see different files in the installation prefix:

```
zig-out/
└── bin
    └── hello.exe
```

Options for Conditional Compilation

To pass options from the build script and into the project's Zig code, use the `Options` step.

app.zig

```
const std = @import("std");
const config = @import("config");

const semver = std.SemanticVersion.parse(config.version) catch
unreachable;

extern fn foo_bar() void;

pub fn main() !void {
    if (semver.major < 1) {
        @compileError("too old");
    }
    std.debug.print("version: {s}\n", .{config.version});

    if (config.have_libfoo) {
        foo_bar();
    }
}
```

build.zig

```
const std = @import("std");

pub fn build(b: *std.Build) void {
    const exe = b.addExecutable(.{
        .name = "app",
        .root_source_file = .{ .path = "app.zig" },
        .target = b.host,
    });

    const version = b.option([]const u8, "version", "application version
string") orelse "0.0.0";
    const enable_foo = detectWhetherToEnableLibFoo();

    const options = b.addOptions();
    options.addOption([]const u8, "version", version);
    options.addOption(bool, "have_libfoo", enable_foo);

    exe.root_module.addOptions("config", options);

    b.installArtifact(exe);
}

fn detectWhetherToEnableLibFoo() bool {
    return false;
}
```

```
$ zig build -Dversion=1.2.3 --summary all
Build Summary: 4/4 steps succeeded
install success
└─ install app success
   └─ zig build-exe app Debug native success 1s MaxRSS:174M
      └─ options success
```

In this example, the data provided by `@import("config")` is comptime-known, preventing the `@compileError` from triggering. If we had passed `-Dversion=0.2.3` or omitted the option, then we would have seen the compilation of app.zig fail with the “too old” error.

Static Library

This build script creates a static library from Zig code, and then also an executable from other Zig code that consumes it.

fizzbuzz.zig

```
export fn fizzbuzz(n: usize) ?[*:0]const u8 {
    if (n % 5 == 0) {
        if (n % 3 == 0) {
            return "fizzbuzz";
        } else {
            return "fizz";
        }
    } else if (n % 3 == 0) {
        return "buzz";
    }
    return null;
}
```

demo.zig

```
const std = @import("std");

extern fn fizzbuzz(n: usize) ?[*:0]const u8;

pub fn main() !void {
    const stdout = std.io.getStdOut();
    var bw = std.io.bufferedWriter(stdout.writer());
    const w = bw.writer();
    for (0..100) |n| {
        if (fizzbuzz(n)) |s| {
            try w.print("{s}\n", .{s});
        } else {
            try w.print("{d}\n", .{n});
        }
    }
    try bw.flush();
}
```

build.zig

```
const std = @import("std");

pub fn build(b: *std.Build) void {
    const target = b.standardTargetOptions(.{});
    const optimize = b.standardOptimizeOption(.{});

    const libfizzbuzz = b.addStaticLibrary(.{
        .name = "fizzbuzz",
        .root_source_file = .{ .path = "fizzbuzz.zig" },
        .target = target,
        .optimize = optimize,
    });

    const exe = b.addExecutable(.{
        .name = "demo",
        .root_source_file = .{ .path = "demo.zig" },
        .target = target,
        .optimize = optimize,
    });

    exe.linkLibrary(libfizzbuzz);

    b.installArtifact(libfizzbuzz);

    if (b.option(bool, "enable-demo", "install the demo too") orelse
false) {
        b.installArtifact(exe);
    }
}
```

```
$ zig build --summary all
Build Summary: 3/3 steps succeeded
install success
└─ install fizzbuzz success
   └─ zig build-lib fizzbuzz Debug native success 36ms MaxRSS:81M
```

In this case, only the static library ends up being installed:

```
zig-out/
└── lib
    └── libfizzbuzz.a
```

However, if you look closely, the build script contains an option to also install the demo. If we additionally pass `-Denable-demo`, then we see this in the installation prefix:

```
zig-out/
├── bin
│   └── demo
└── lib
    └── libfizzbuzz.a
```

Note that despite the unconditional call to `addExecutable`, the build system in fact does not waste any time building the `demo` executable unless it is requested with `-Denable-demo`, because the build system is based on a Directed Acyclic Graph with dependency edges.

Dynamic Library

Here we keep all the files the same from the Static Library example, except the `build.zig` file is changed.

build.zig

```
const std = @import("std");

pub fn build(b: *std.Build) void {
    const target = b.standardTargetOptions(.{});
    const optimize = b.standardOptimizeOption(.{});

    const libfizzbuzz = b.addSharedLibrary(.{
        .name = "fizzbuzz",
        .root_source_file = .{ .path = "fizzbuzz.zig" },
        .target = target,
        .optimize = optimize,
        .version = .{ .major = 1, .minor = 2, .patch = 3 },
    });

    b.installArtifact(libfizzbuzz);
}
```

```
$ zig build --summary all
Build Summary: 3/3 steps succeeded
install success
└─ install fizzbuzz success
   └─ zig build-lib fizzbuzz Debug native success 3s MaxRSS:183M
```

Output

```
zig-out
└── lib
    ├── libfizzbuzz.so -> libfizzbuzz.so.1
    ├── libfizzbuzz.so.1 -> libfizzbuzz.so.1.2.3
    └── libfizzbuzz.so.1.2.3
```

As in the static library example, to make an executable link against it, use code like this:

```
exe.linkLibrary(libfizzbuzz);
```

Testing

Individual files can be tested directly with `zig test foo.zig`, however, more complex use cases can be solved by orchestrating testing via the build script.

When using the build script, unit tests are broken into two different steps in the build graph, the **Compile** step and the **Run** step. Without a call to `addRunArtifact`, which establishes a dependency edge between these two steps, the unit tests will not be executed.

The Compile step can be configured the same as any executable, library, or object file, for example by linking against system libraries, setting target options, or adding additional compilation units.

The Run step can be configured the same as any Run step, for example by skipping execution when the host is not capable of executing the binary.

When using the build system to run unit tests, the build runner and the test runner communicate via stdin and stdout in order to run multiple unit test suites concurrently, and report test failures in a meaningful way without having their output jumbled together. This is one reason why writing to standard out in unit tests is problematic - it will interfere with this communication channel. On the flip side, this mechanism will enable an upcoming feature, which is is the ability for a unit test to expect a panic.

main.zig

```
const std = @import("std");

test "simple test" {
    var list = std.ArrayList(i32).init(std.testing.allocator);
    defer list.deinit();
    try list.append(42);
    try std.testing.expectEqual(@as(i32, 42), list.pop());
}
```

build.zig

```
const std = @import("std");

const test_targets = [_]std.Target.Query{
    .{}, // native
    .{
        .cpu_arch = .x86_64,
        .os_tag = .linux,
    },
    .{
        .cpu_arch = .aarch64,
        .os_tag = .macos,
    },
};

pub fn build(b: *std.Build) void {
    const test_step = b.step("test", "Run unit tests");

    for (test_targets) |target| {
        const unit_tests = b.addTest(.{
            .root_source_file = .{ .path = "main.zig" },
            .target = b.resolveTargetQuery(target),
        });

        const run_unit_tests = b.addRunArtifact(unit_tests);
        test_step.dependOn(&run_unit_tests.step);
    }
}
```

```
$ zig build test --summary all
test
└─ run test failure
error: the host system (x86_64-linux.6.2...6.2-gnu.2.35) is unable to
execute binaries from the target (aarch64-macos.11.7.1...14.1-none)
Build Summary: 5/7 steps succeeded; 1 failed; 2/2 tests passed
test transitive failure
├─ run test 1 passed 804us MaxRSS:1M
│  └─ zig test Debug native success 5s MaxRSS:193M
├─ run test 1 passed 763us MaxRSS:1M
│  └─ zig test Debug x86_64-linux success 5s MaxRSS:226M
└─ run test failure
   └─ zig test Debug aarch64-macos success 5s MaxRSS:229M
error: the following build command failed with exit code 1:
../../../../doctest-032390e1/zig-
cache/o/f1fb63d0b8f47be2d6a526fa1c7567f9/build
/home/runner/work/www.ziglang.org/www.ziglang.org/zig/zig
/home/runner/work/www.ziglang.org/www.ziglang.org/assets/zig-code/build-
system/unit-testing ../../../../doctest-032390e1/zig-cache
/home/runner/.cache/zig --seed 0x81010448 --color on --prefix
../../../../doctest-032390e1/zig-out test --summary all
```

In this case it might be a nice adjustment to enable `skip_foreign_checks` for the unit tests:

```
@@ -23,6 +23,7 @@
         });

         const run_unit_tests = b.addRunArtifact(unit_tests);
+        run_unit_tests.skip_foreign_checks = true;
         test_step.dependOn(&run_unit_tests.step);
     }
 }
```

build.zig (click to expand/collapse)

```
const std = @import("std");

const test_targets = [_]std.Target.Query{
    .{}, // native
    .{
        .cpu_arch = .x86_64,
        .os_tag = .linux,
    },
    .{
        .cpu_arch = .aarch64,
        .os_tag = .macos,
    },
};

pub fn build(b: *std.Build) void {
    const test_step = b.step("test", "Run unit tests");

    for (test_targets) |target| {
        const unit_tests = b.addTest(.{
            .root_source_file = .{ .path = "main.zig" },
            .target = b.resolveTargetQuery(target),
        });

        const run_unit_tests = b.addRunArtifact(unit_tests);
        run_unit_tests.skip_foreign_checks = true;
        test_step.dependOn(&run_unit_tests.step);
    }
}
```

```
$ zig build test --summary all
Build Summary: 6/7 steps succeeded; 1 skipped; 2/2 tests passed
test success
├─ run test 1 passed 877us MaxRSS:1M
│─  └─ zig test Debug native success 2s MaxRSS:194M
├─ run test 1 passed 1ms MaxRSS:1M
│─  └─ zig test Debug x86_64-linux success 2s MaxRSS:195M
└─ run test skipped
   └─ zig test Debug aarch64-macos success 2s MaxRSS:192M
```

Linking to System Libraries

For satisfying library dependencies, there are two choices:

1. Provide these libraries via the Zig Build System (see Package Management and Static Library).
2. Use the files provided by the host system.

For the use case of upstream project maintainers, obtaining these libraries via the Zig Build System provides the least friction and puts the configuration power in the hands of those maintainers. Everyone who builds this way will have reproducible, consistent results as each other, and it will work on every operating system and even support cross-compilation. Furthermore, it allows the project to decide with perfect precision the exact versions of its entire dependency tree it wishes to build against. This is expected to be the generally preferred way to depend on external libraries.

However, for the use case of packaging software into repositories such as Debian, Homebrew, or Nix, it is mandatory to link against system libraries. So, build scripts must detect the mode and configure accordingly.

build.zig

```
const std = @import("std");

pub fn build(b: *std.Build) void {
    const exe = b.addExecutable(.{
        .name = "zip",
        .root_source_file = .{ .path = "zip.zig" },
        .target = b.host,
    });

    exe.linkSystemLibrary("z");
    exe.linkLibC();

    b.installArtifact(exe);
}
```

```
$ zig build --summary all
Build Summary: 3/3 steps succeeded
install success
└─ install zip success
   └─ zig build-exe zip Debug native success 4s MaxRSS:220M
```

Users of `zig build` may use `--search-prefix` to provide additional directories that are considered “system directories” for the purposes of finding static and dynamic libraries.

Generating Files

Running System Tools

This version of hello world expects to find a `word.txt` file in the same path, and we want to use a system tool to generate it starting from a JSON file.

Be aware that system dependencies will make your project harder to build for your users. This build script depends on `jq`, for example, which is not present by default in most Linux distributions and which might be an unfamiliar tool for Windows users.

The next section will replace `jq` with a Zig tool included in the source tree, which is the preferred approach.

words.json

```
{
  "en": "world",
  "it": "mondo",
  "ja": "世界"
}
```

main.zig

```
const std = @import("std");

pub fn main() !void {
    var arena_state =
std.heap.ArenaAllocator.init(std.heap.page_allocator);
    defer arena_state.deinit();
    const arena = arena_state.allocator();

    const self_exe_dir_path = try std.fs.selfExeDirPathAlloc(arena);
    var self_exe_dir = try std.fs.cwd().openDir(self_exe_dir_path, .{});
    defer self_exe_dir.close();

    const word = try self_exe_dir.readFileAlloc(arena, "word.txt", 1000);

    try std.io.getStdOut().writer().print("Hello {s}\n", .{word});
}
```

build.zig

```
const std = @import("std");

pub fn build(b: *std.Build) void {
    const lang = b.option([]const u8, "language", "language of the
greeting") orelse "en";
    const tool_run = b.addSystemCommand(&.{"jq"});
    tool_run.addArgs(&.{
        b.fmt(
            \\.["{s}"]
        , .{lang}),
        "-r", // raw output to omit quotes around the selected string
    });
    tool_run.addFileArg(.{ .path = "words.json" });

    const output = tool_run.captureStdOut();

    b.getInstallStep().dependOn(&b.addInstallFileWithDir(output, .prefix,
"word.txt").step);

    const target = b.standardTargetOptions(.{});
    const optimize = b.standardOptimizeOption(.{});
    const exe = b.addExecutable(.{
        .name = "hello",
        .root_source_file = .{ .path = "src/main.zig" },
        .target = target,
        .optimize = optimize,
    });

    const install_artifact = b.addInstallArtifact(exe, .{
        .dest_dir = .{ .override = .prefix },
    });
    b.getInstallStep().dependOn(&install_artifact.step);
}
```

```
$ zig build -Dlanguage=ja --summary all
Build Summary: 5/5 steps succeeded
install success
├─ install generated to word.txt success
│─  └─ run jq success 29ms MaxRSS:3M
└─ install hello success
   └─ zig build-exe hello Debug native success 1s MaxRSS:175M
```

Output

```
zig-out
├── hello
└── word.txt
```

Note how `captureStdOut` creates a temporary file with the output of the `jq` invocation.

Running the Project's Tools

This version of hello world expects to find a `word.txt` file in the same path, and we want to produce it at build-time by invoking a Zig program on a JSON file.

tools/words.json

```
{
  "en": "world",
  "it": "mondo",
  "ja": "世界"
}
```

main.zig

```
const std = @import("std");

pub fn main() !void {
    var arena_state =
std.heap.ArenaAllocator.init(std.heap.page_allocator);
    defer arena_state.deinit();
    const arena = arena_state.allocator();

    const self_exe_dir_path = try std.fs.selfExeDirPathAlloc(arena);
    var self_exe_dir = try std.fs.cwd().openDir(self_exe_dir_path, .{});
    defer self_exe_dir.close();

    const word = try self_exe_dir.readFileAlloc(arena, "word.txt", 1000);

    try std.io.getStdOut().writer().print("Hello {s}\n", .{word});
}
```

tools/word_select.zig (click to expand/collapse)

```
const std = @import("std");

const usage =
    \\Usage: ./word_select [options]
    \\
    \\Options:
    \\  --input-file INPUT_JSON_FILE
    \\  --output-file OUTPUT_TXT_FILE
    \\  --lang LANG
    \\
;

pub fn main() !void {
    var arena_state =
std.heap.ArenaAllocator.init(std.heap.page_allocator);
    defer arena_state.deinit();
    const arena = arena_state.allocator();

    const args = try std.process.argsAlloc(arena);

    var opt_input_file_path: ?[]const u8 = null;
    var opt_output_file_path: ?[]const u8 = null;
    var opt_lang: ?[]const u8 = null;

    {
        var i: usize = 1;
        while (i < args.len) : (i += 1) {
            const arg = args[i];
            if (std.mem.eql(u8, "-h", arg) or std.mem.eql(u8, "--help",
arg)) {
                try std.io.getStdOut().writeAll(usage);
                return std.process.cleanExit();
            } else if (std.mem.eql(u8, "--input-file", arg)) {
                i += 1;
                if (i > args.len) fatal("expected arg after '{s}'", .
{arg});
                if (opt_input_file_path != null) fatal("duplicated {s}
argument", .{arg});
                opt_input_file_path = args[i];
            } else if (std.mem.eql(u8, "--output-file", arg)) {
                i += 1;
                if (i > args.len) fatal("expected arg after '{s}'", .
{arg});
                if (opt_output_file_path != null) fatal("duplicated {s}
argument", .{arg});
                opt_output_file_path = args[i];
            } else if (std.mem.eql(u8, "--lang", arg)) {
                i += 1;
                if (i > args.len) fatal("expected arg after '{s}'", .
{arg});
                if (opt_lang != null) fatal("duplicated {s} argument", .
{arg});
                opt_lang = args[i];
            } else {
                fatal("unrecognized arg: '{s}'", .{arg});
            }
        }
    }

    const input_file_path = opt_input_file_path orelse fatal("missing --
input-file", .{});
input-file", .{});
    const output_file_path = opt_output_file_path orelse fatal("missing -
-output-file", .{});
    const lang = opt_lang orelse fatal("missing --lang", .{});

    var input_file = std.fs.cwd().openFile(input_file_path, .{}) catch
|err| {
        fatal("unable to open '{s}': {s}", .{ input_file_path,
@errorName(err) });
    };
    defer input_file.close();

    var output_file = std.fs.cwd().createFile(output_file_path, .{})
catch |err| {
        fatal("unable to open '{s}': {s}", .{ output_file_path,
@errorName(err) });
    };
    defer output_file.close();

    var json_reader = std.json.reader(arena, input_file.reader());
    var words = try std.json.ArrayHashMap([]const u8).jsonParse(arena,
&json_reader, .{
        .allocate = .alloc_if_needed,
        .max_value_len = 1000,
    });

    const w = words.map.get(lang) orelse fatal("Lang not found in JSON
file", .{});

    try output_file.writeAll(w);
    return std.process.cleanExit();
}

fn fatal(comptime format: []const u8, args: anytype) noreturn {
    std.debug.print(format, args);
    std.process.exit(1);
}
```

build.zig

```
const std = @import("std");

pub fn build(b: *std.Build) void {
    const lang = b.option([]const u8, "language", "language of the
greeting") orelse "en";
    const tool = b.addExecutable(.{
        .name = "word_select",
        .root_source_file = .{ .path = "tools/word_select.zig" },
        .target = b.host,
    });

    const tool_step = b.addRunArtifact(tool);
    tool_step.addArg("--input-file");
    tool_step.addFileArg(.{ .path = "tools/words.json" });
    tool_step.addArg("--output-file");
    const output = tool_step.addOutputFileArg("word.txt");
    tool_step.addArgs(&.{ "--lang", lang });

    b.getInstallStep().dependOn(&b.addInstallFileWithDir(output, .prefix,
"word.txt").step);

    const target = b.standardTargetOptions(.{});
    const optimize = b.standardOptimizeOption(.{});
    const exe = b.addExecutable(.{
        .name = "hello",
        .root_source_file = .{ .path = "src/main.zig" },
        .target = target,
        .optimize = optimize,
    });

    const install_artifact = b.addInstallArtifact(exe, .{
        .dest_dir = .{ .override = .prefix },
    });
    b.getInstallStep().dependOn(&install_artifact.step);
}
```

```
$ zig build -Dlanguage=ja --summary all
Build Summary: 6/6 steps succeeded
install success
├─ install generated to word.txt success
│  └─ run word_select (word.txt) success 606us MaxRSS:1M
│     └─ zig build-exe word_select Debug native success 1s MaxRSS:192M
└─ install hello success
   └─ zig build-exe hello Debug native success 1s MaxRSS:173M
```

Output

```
zig-out
├── hello
└── word.txt
```

Producing Assets for `@embedFile`

This version of hello world wants to `@embedFile` an asset generated at build time, which we're going to produce using a tool written in Zig.

tools/words.json

```
{
  "en": "world",
  "it": "mondo",
  "ja": "世界"
}
```

main.zig

```
const std = @import("std");
const word = @embedFile("word");

pub fn main() !void {
    try std.io.getStdOut().writer().print("Hello {s}\n", .{word});
}
```

tools/word_select.zig (click to expand/collapse)

```
const std = @import("std");

const usage =
    \\Usage: ./word_select [options]
    \\
    \\Options:
    \\  --input-file INPUT_JSON_FILE
    \\  --output-file OUTPUT_TXT_FILE
    \\  --lang LANG
    \\
;

pub fn main() !void {
    var arena_state =
std.heap.ArenaAllocator.init(std.heap.page_allocator);
    defer arena_state.deinit();
    const arena = arena_state.allocator();

    const args = try std.process.argsAlloc(arena);

    var opt_input_file_path: ?[]const u8 = null;
    var opt_output_file_path: ?[]const u8 = null;
    var opt_lang: ?[]const u8 = null;

    {
        var i: usize = 1;
        while (i < args.len) : (i += 1) {
            const arg = args[i];
            if (std.mem.eql(u8, "-h", arg) or std.mem.eql(u8, "--help",
arg)) {
                try std.io.getStdOut().writeAll(usage);
                return std.process.cleanExit();
            } else if (std.mem.eql(u8, "--input-file", arg)) {
                i += 1;
                if (i > args.len) fatal("expected arg after '{s}'", .
{arg});
                if (opt_input_file_path != null) fatal("duplicated {s}
argument", .{arg});
```

```
                opt_input_file_path = args[i];
            } else if (std.mem.eql(u8, "--output-file", arg)) {
                i += 1;
                if (i > args.len) fatal("expected arg after '{s}'", .
{arg});
                if (opt_output_file_path != null) fatal("duplicated {s}
argument", .{arg});
                opt_output_file_path = args[i];
            } else if (std.mem.eql(u8, "--lang", arg)) {
                i += 1;
                if (i > args.len) fatal("expected arg after '{s}'", .
{arg});
                if (opt_lang != null) fatal("duplicated {s} argument", .
{arg});
                opt_lang = args[i];
            } else {
                fatal("unrecognized arg: '{s}'", .{arg});
            }
        }
    }

    const input_file_path = opt_input_file_path orelse fatal("missing --
input-file", .{});
    const output_file_path = opt_output_file_path orelse fatal("missing -
-output-file", .{});
    const lang = opt_lang orelse fatal("missing --lang", .{});

    var input_file = std.fs.cwd().openFile(input_file_path, .{}) catch
|err| {
        fatal("unable to open '{s}': {s}", .{ input_file_path,
@errorName(err) });
    };
    defer input_file.close();

    var output_file = std.fs.cwd().createFile(output_file_path, .{})
catch |err| {
        fatal("unable to open '{s}': {s}", .{ output_file_path,
@errorName(err) });
    };
    defer output_file.close();

    var json_reader = std.json.reader(arena, input_file.reader());
    var words = try std.json.ArrayHashMap([]const u8).jsonParse(arena,
&json_reader, .{
        .allocate = .alloc_if_needed,
        .max_value_len = 1000,
    });

    const w = words.map.get(lang) orelse fatal("Lang not found in JSON
file", .{});

    try output_file.writeAll(w);
    return std.process.cleanExit();
}

fn fatal(comptime format: []const u8, args: anytype) noreturn {
    std.debug.print(format, args);
    std.process.exit(1);
}
```

build.zig

```
const std = @import("std");

pub fn build(b: *std.Build) void {
    const lang = b.option([]const u8, "language", "language of the
greeting") orelse "en";
    const tool = b.addExecutable(.{
        .name = "word_select",
        .root_source_file = .{ .path = "tools/word_select.zig" },
        .target = b.host,
    });

    const tool_step = b.addRunArtifact(tool);
    tool_step.addArg("--input-file");
    tool_step.addFileArg(.{ .path = "tools/words.json" });
    tool_step.addArg("--output-file");
    const output = tool_step.addOutputFileArg("word.txt");
    tool_step.addArgs(&.{ "--lang", lang });

    const target = b.standardTargetOptions(.{});
    const optimize = b.standardOptimizeOption(.{});
    const exe = b.addExecutable(.{
        .name = "hello",
        .root_source_file = .{ .path = "src/main.zig" },
        .target = target,
        .optimize = optimize,
    });

    exe.root_module.addAnonymousImport("word", .{
        .root_source_file = output,
    });

    b.installArtifact(exe);
}
```

```
$ zig build -Dlanguage=ja --summary all
Build Summary: 5/5 steps succeeded
install success
└─ install hello success
   └─ zig build-exe hello Debug native success 1s MaxRSS:174M
      └─ run word_select (word.txt) success 576us MaxRSS:1M
         └─ zig build-exe word_select Debug native success 1s MaxRSS:193M
```

Output

```
zig-out/
└── bin
    └── hello
```

Generating Zig Source Code

This build file uses a Zig program to generate a Zig file and then exposes it to the main program as a module dependency.

main.zig

```
const std = @import("std");
const Person = @import("person").Person;

pub fn main() !void {
    const p: Person = .{};
    try std.io.getStdOut().writer().print("Hello {any}\n", .{p});
}
```

generate_struct.zig

```
const std = @import("std");

pub fn main() !void {
    var arena_state =
std.heap.ArenaAllocator.init(std.heap.page_allocator);
    defer arena_state.deinit();
    const arena = arena_state.allocator();

    const args = try std.process.argsAlloc(arena);

    if (args.len != 2) fatal("wrong number of arguments", .{});

    const output_file_path = args[1];

    var output_file = std.fs.cwd().createFile(output_file_path, .{})
catch |err| {
        fatal("unable to open '{s}': {s}", .{ output_file_path,
@errorName(err) });
    };
    defer output_file.close();

    try output_file.writeAll(
        \\pub const Person = struct {
        \\   age: usize = 18,
        \\   name: []const u8 = "foo"
        \\};
    );
    return std.process.cleanExit();
}

fn fatal(comptime format: []const u8, args: anytype) noreturn {
    std.debug.print(format, args);
    std.process.exit(1);
}
```

build.zig

```
const std = @import("std");

pub fn build(b: *std.Build) void {
    const tool = b.addExecutable(.{
        .name = "generate_struct",
        .root_source_file = .{ .path = "tools/generate_struct.zig" },
        .target = b.host,
    });

    const tool_step = b.addRunArtifact(tool);
    const output = tool_step.addOutputFileArg("person.zig");

    const target = b.standardTargetOptions(.{});
    const optimize = b.standardOptimizeOption(.{});
    const exe = b.addExecutable(.{
        .name = "hello",
        .root_source_file = .{ .path = "src/main.zig" },
        .target = target,
        .optimize = optimize,
    });

    exe.root_module.addAnonymousImport("person", .{
        .root_source_file = output,
    });

    b.installArtifact(exe);
}
```

```
$ zig build --summary all
Build Summary: 5/5 steps succeeded
install success
└─ install hello success
   └─ zig build-exe hello Debug native success 1s MaxRSS:175M
      └─ run generate_struct (person.zig) success 546us MaxRSS:1M
         └─ zig build-exe generate_struct Debug native success 1s
MaxRSS:175M
```

Output

```
zig-out/
└── bin
    └── hello
```

Dealing With One or More Generated Files

The **WriteFiles** step provides a way to generate one or more files which share a parent directory. The generated directory lives inside the local zig-cache, and each generated file is independently available as a `std.Build.LazyPath`. The parent directory itself is also available as a LazyPath.

This API supports writing arbitrary strings to the generated directory as well as copying files into it.

main.zig

```
const std = @import("std");

pub fn main() !void {
    std.debug.print("hello world\n", .{});
}
```

build.zig

```
const std = @import("std");

pub fn build(b: *std.Build) void {
    const exe = b.addExecutable(.{
        .name = "app",
        .root_source_file = .{ .path = "src/main.zig" },
        .target = b.host,
    });

    const version = b.option([]const u8, "version", "application version
string") orelse "0.0.0";

    const wf = b.addWriteFiles();
    const app_exe_name = b.fmt("project/{s}", .{exe.out_filename});
    _ = wf.addCopyFile(exe.getEmittedBin(), app_exe_name);
    _ = wf.add("project/version.txt", version);

    const tar = b.addSystemCommand(&.{ "tar", "czf" });
    tar.setCwd(wf.getDirectory());
    const out_file = tar.addOutputFileArg("project.tar.gz");
    tar.addArgs(&.{"project/"});

    const install_tar = b.addInstallFileWithDir(out_file, .prefix,
"project.tar.gz");
    b.getInstallStep().dependOn(&install_tar.step);
}
```

```
$ zig build --summary all
Build Summary: 5/5 steps succeeded
install success
└─ install generated to project.tar.gz success
   └─ run tar (project.tar.gz) success 108ms MaxRSS:3M
      └─ WriteFile project/app success
         └─ zig build-exe app Debug native success 1s MaxRSS:173M
```

Output

```
zig-out/
└── project.tar.gz
```

Mutating Source Files in Place

It is uncommon, but sometimes the case that a project commits generated files into version control. This can be useful when the generated files are seldomly updated and have burdensome system dependencies for the update process, but *only* during the update process.

For this, **WriteFiles** provides a way to accomplish this task. This is a feature that will be extracted from WriteFiles into its own Build Step in a future Zig version.

Be careful with this functionality; it should not be used during the normal build process, but as a utility run by a developer with intention to update source files, which will then be committed to version control. If it is done during the normal build process, it will cause caching and concurrency bugs.

proto_gen.zig

```
const std = @import("std");

pub fn main() !void {
    var arena_state =
std.heap.ArenaAllocator.init(std.heap.page_allocator);
    defer arena_state.deinit();
    const arena = arena_state.allocator();

    const args = try std.process.argsAlloc(arena);

    if (args.len != 2) fatal("wrong number of arguments", .{});

    const output_file_path = args[1];

    var output_file = std.fs.cwd().createFile(output_file_path, .{})
catch |err| {
        fatal("unable to open '{s}': {s}", .{ output_file_path,
@errorName(err) });
    };
    defer output_file.close();

    try output_file.writeAll(
        \\pub const Header = extern struct {
        \\    magic: u64,
        \\    width: u32,
        \\    height: u32,
        \\};
    );
    return std.process.cleanExit();
}

fn fatal(comptime format: []const u8, args: anytype) noreturn {
    std.debug.print(format, args);
    std.process.exit(1);
}
```

main.zig

```
const std = @import("std");
const Protocol = @import("protocol.zig");

pub fn main() !void {
    const header = try
std.io.getStdIn().reader().readStruct(Protocol.Header);
    std.debug.print("header: {any}\n", .{header});
}
```

protocol.zig

```
pub const Header = extern struct {
    magic: u64,
    width: u32,
    height: u32,
};
```

build.zig

```
const std = @import("std");

pub fn build(b: *std.Build) void {
    const exe = b.addExecutable(.{
        .name = "demo",
        .root_source_file = .{ .path = "src/main.zig" },
    });
    b.installArtifact(exe);

    const proto_gen = b.addExecutable(.{
        .name = "proto_gen",
        .root_source_file = .{ .path = "tools/proto_gen.zig" },
    });

    const run = b.addRunArtifact(proto_gen);
    const generated_protocol_file = run.addOutputFileArg("protocol.zig");

    const wf = b.addWriteFiles();
    wf.addCopyFileToSource(generated_protocol_file, "src/protocol.zig");

    const update_protocol_step = b.step("update-protocol", "update
src/protocol.zig to latest");
    update_protocol_step.dependOn(&wf.step);
}

fn detectWhetherToEnableLibFoo() bool {
    return false;
}
```

```
$ zig build update-protocol --summary all
Build Summary: 4/4 steps succeeded
update-protocol success
└─ WriteFile success
   └─ run proto_gen (protocol.zig) success 401us MaxRSS:1M
      └─ zig build-exe proto_gen Debug native success 1s MaxRSS:183M
```

After running this command, `src/protocol.zig` is updated in place.

Handy Examples

Build for multiple targets to make a release

In this example we're going to change some defaults when creating an `InstallArtifact` step in order to put the build for each target into a separate subdirectory inside the install path.

build.zig

```
const std = @import("std");

const targets: []const std.Target.Query = &.{
    .{ .cpu_arch = .aarch64, .os_tag = .macos },
    .{ .cpu_arch = .aarch64, .os_tag = .linux },
    .{ .cpu_arch = .x86_64, .os_tag = .linux, .abi = .gnu },
    .{ .cpu_arch = .x86_64, .os_tag = .linux, .abi = .musl },
    .{ .cpu_arch = .x86_64, .os_tag = .windows },
};

pub fn build(b: *std.Build) !void {
    for (targets) |t| {
        const exe = b.addExecutable(.{
            .name = "hello",
            .root_source_file = .{ .path = "hello.zig" },
            .target = b.resolveTargetQuery(t),
            .optimize = .ReleaseSafe,
        });

        const target_output = b.addInstallArtifact(exe, .{
            .dest_dir = .{
                .override = .{
                    .custom = try t.zigTriple(b.allocator),
                },
            },
        });

        b.getInstallStep().dependOn(&target_output.step);
    }
}
```

```
$ zig build --summary all
Build Summary: 11/11 steps succeeded
install success
├─ install hello success
│─  └─ zig build-exe hello ReleaseSafe aarch64-macos success 12s
MaxRSS:217M
├─ install hello success
│─  └─ zig build-exe hello ReleaseSafe aarch64-linux success 22s
MaxRSS:222M
├─ install hello success
│─  └─ zig build-exe hello ReleaseSafe x86_64-linux-gnu success 21s
MaxRSS:211M
├─ install hello success
│─  └─ zig build-exe hello ReleaseSafe x86_64-linux-musl success 22s
MaxRSS:212M
└─ install hello success
   └─ zig build-exe hello ReleaseSafe x86_64-windows success 24s
MaxRSS:204M
```

hello.zig

```
const std = @import("std");

pub fn main() !void {
    std.debug.print("Hello World!\n", .{});
}
```

Output

```
zig-out
├── aarch64-linux
│   └── hello
├── aarch64-macos
│   └── hello
├── x86_64-linux-gnu
│   └── hello
├── x86_64-linux-musl
│   └── hello
└── x86_64-windows
    ├── hello.exe
    └── hello.pdb
```