The Xcode Build System

This post is a dive into Xcode as a system. To fully understand some of these topics you should familiarize yourself with the following:

- The Project File: Part 1
- The Project File: Part 2
- Managing Xcode
- Xcconfig Guide
- Using Xcode Targets
- Sparse SDKs
- Build Settings Reference

I would recommend using these posts as reference material while reading this, as there is a lot to cover and I cannot do it all in this post :)

Build System Components

DEVELOPER_DIR

At one point, Xcode didn't install as a ".app" on your system. It installed as a couple separate applications and the majority of the tooling was loaded from a single directory called "Developer". The path to this directory was set as `DEVELOPER_DIR` and from there all the components were loaded from the internal structuring of this directory. In modern versions of Xcode this directory still exists, however it is inside the Xcode app bundle instead of in a user-visible place. From this directory Xcode will load a number of bundles.

Plug-ins

Xcode uses plug-ins to dynamically load necessary components, much of the core systems are loaded from these plug-ins. These plug-ins are loaded as necessary depending on loading Xcode GUI or `xcodebuild`. Because this dynamic loading is so flexible it makes it very easy to build and integrate new components into Xcode without a lot of work. This is made even easier with the help of specification files, which we will get to later in more detail, to the point where adding: new file types, build rules, and tooling can be done without touching any code.

Third-Party

There are some third party tools that are implemented as Xcode plugins. Alcatraz is a good source for exploring what you can do with additional plugins. The only caveat here is that Xcode doesn't officially support third party plug-ins. You must add some compatibility support to each plug-in yourself by updating value in the plug-in's `Info.plist` to let Xcode know that the plug-in supports a specific version of Xcode.

Platforms

Platform bundles are a core component of the Xcode build system. They define the systems that can be built for and the types of products that can be built. For example: up until iOS 8 Xcode didn't support building frameworks for iOS as a platform, only static libraries. This didn't mean iOS was incapable of using them, only that Xcode didn't have a product definition for them. In addition to defining what can be built, they also provide the parameters of how it can be build. You cannot build a mac app that is targeting the ARM architecture, as OS X doesn't support running on that. While Xcode uses the default toolchain for most of the tooling used in the build process, there are supplemental tools that are loaded from the platform bundle. These are new or have some platform specific behavior that is desired. You can only use platform specific tools based on the currently selected platform you are targeting.

SDKs

These are bundles contained within each platform bundle. These define the frameworks and libraries that should be used during compilation, they are a variant of the platform. They should only act as details to the platform definition, and don't contain tooling. There are two types of SDK bundles, core and sparse. Core SDK bundles are part of a platform bundle. They contain all the necessary components to build against a particular platform. There are also Sparse SDKs, these are not part of a platform bundle and are used to add supplemental libraries that are not part of a Core SDK. Like platform bundles, Xcode will load any SDK bundles that are in the normal search paths. This only applies to Core SDKs (since Xcode will only search within the defined `DEVELOPER_DIR`), so while it is possible to add new platform and SDK bundles to Xcode this would break the code signature on the application. You can specify paths to Sparse SDKs via the `ADDITIONAL_SDKS` build setting variable.

When performing a build, Xcode will create a composited SDK out of the selected SDK and the Spare SDK bundle located at the `ADDITIONAL_SDKS`

path. This composited SDK is stored in a temporary directory that will be be removed by the system after the build is completed.

Toolchains

This is where all the primary tools of the build system are stored. Prior to Xcode 7.2, there was only support for the single built-in Toolchain bundle that was shipped with Xcode. With the release of Swift, Apple has provided a means to install additional toolchains for building various versions of Swift. Alongside the release of Swift, Apple published Xcode 7.2, which adds the ability to utilize additional toolchains via `xcrun --toolchain <name>`. Not only does the Toolchain include the binary tools used (compilers, linkers, resource processors, etc), it also contains libraries that the tools may depend upon. This is to provide support for libraries that won't exist on the target deployment platform. Such as building any code with Swift (versions 1 and 2), or building apps with ARC that are supposed to run on older, legacy systems. When Xcode performs a build, it will use the selected Toolchain to resolve the tools needed as part of the build process.

Build System Specifications

Each tool, file type, syntax rule, and even the expected behaviors of how the built products are created are defined by specification files that get loaded from disk. Some of these are always loaded (such as file types and compiler tools), and other will only be loaded if they are needed (platform specific definitions). This makes Xcode's build system extremely modular and flexible. These specification files named "xcspec" files, due to their file extension.

Builtin Specifications

Xcode has many builtin "default" specifications. These include languages, the toolchain, syntax rules, and build rules. These are universal and don't change regardless of what Platform or SDK is used. These will sometimes change when you install a new version of Xcode, once such example is when the switch from gcc to llvm happened.

Platform Specifications

Each target on a project file will define a SDK to use when a build should be executed. This defines the type of the built product and is done via the `SDKROOT` build setting variable. This is going to be one of the SDKs available in the directory set to `DEVELOPER_DIR` (the path that is set for `xcode-select`).

Xcode will load some additional specs from this SDK and the Platform bundle it is a part of. These will define some rules about the output and tooling that is platform specific (such as iOS binaries must be signed).

External Specifications

Due to the nature of Xcode being plugin-based (Swift support is implemented via an internal plugin), it will load any specification files it finds in the plugins it loads. This allows for the implementation of third party tooling natively in the GUI. Loading additional specifications allows you to define custom build rules, ship your own versions of the tooling, and even add new build settings to appear in the GUI. (If you want to see an example of this in action you can check out the plugin I made here: https://github.com/gwynne/citrus/pull/1)

The Build Process

Project File

The project file is a means of communicating with the build system. The structure is designed to communicate the “what” should be built and the “how” it should be built. The process of going from source files to the built product is serialized to disk in the project file. Each object that is serialized is used by something and the object type (`isa` identifier on the object) defines the behavior of the object.

PBXProject

Each `pbxproj` file has a single root object, this contains:

- list of defined targets
- list of targets produced by this project file
- references to the project level build settings
- organizational information used within the project
- references to nested project files
- list of files contained and used by the project file

Target Types

There are a lot of target types, many are no longer used are still supported for legacy project files. Each target type defines how it interacts with the build system by using Build Phases and Build Rules.

Build Phases

Build Phases are how we communicate to the build system what needs to be built. Each target contains a list of build phases, each of which dictates a behavior around a specific aspect of a build (compiling files, copying resources, target dependencies, libraries to link, scripts to run, etc). Since the order of these is significant, it is important to not modify these on disk. Additionally, some phases you see in Xcode aren't classified as a build phase when serialized. Target dependencies are not classified as a build phase even though they get displayed as such. This is because they must be run before anything on the target is processed by the build system to prevent one target invalidating the build process of another. The ordering of the build phases is crutial to a successful build. The list of build phases on each target is expected to be operated in a serial manner, processing and executing each one before moving onto the next. While this would seem to make simple builds take longer, this behavior exists to support complex builds that require multiple compilation steps and setup phases.

File References vs Build Files

There are two ways Xcode will classify files. First being file references (`PBXFileReference`), these are representations of a file that is expected to be on disk. These are used to create the representation of your files in file navigator in Xcode. These allow you to organize the display of files in Xcode that doesn't mirror their organization on disk. There are also build files (`PBXBuildFile`), these are linked to file references but are abstractions that are used exclusively by the build system. Build phases use the build files to indicate which files should be processed.

Build Rules

Each phase type contains a list of files that get processed for the action of that particular type of phase. While each phase type has a generic definition of how it works, many use Build Rules to define the processing behavior for each file it contains. Build Rules give phases the ability to contain any types of files, by performing a look up of how each individual file should be processed. The build rules themselves dictate what types of files to look for and what tool to hand that file off to as input. You can create custom build rules that can execute a script to support tooling that isn't a part of Xcode.

The actions you see in Xcode's build log is each file being processed by its build rule. The build rule passes the file has the input to the tool it is associated with.

The tool defines the execution behavior based on the build settings defined in the environment.

Environment

Now that the build system has connected a file to the tool that should be used to process it, it is time to setup the build environment it needs. This is where the build settings come in. When setting up to perform a build Xcode will read in the build configuration that is defined on a target.

Build Settings

The build setting variables are organized into a couple different levels. These are stored in different layers in the “environment” that is created for the build. You can display a list of build settings used by running `xcodebuild MyApp.xcodeproj –showBuildSettings`.

Note

> Xcode will inherit values from the user’s shell environment. To see this, if you setup a “Run Script” phase in your build target to execute the `env` command, it will display all the defined environment variables.

Platform

When Xcode loads the available platforms, it will load the platform’s xcspec files. These contain a set of default values that should be used for a particular platform. For example, the iOS platform sets an environmnet variable to say that code signing is a requirement for deployment. The values loaded on the platform level form the base of the build environment that will be used. Settings on this level are not user-configurable.

Project

Project level build settings are the lowest level of user-configurable setting that apply on a per-project basis. This introduces a new level of settings that are not applicable on the platform level. In addition to being user-configurable, this level of build setting also allows for custom variables and values to be set. The project level build settings directly inherit the values set on the platform level to give enough information to the build system to operate in. Any build settings that are set in both the project level and the platform level will be over-ridden by the value set on the project level.

Target

The target level is the second level of user-configurable build setting, this is also the level that most people are familiar with. This is probably the most heavily used section of all the layers of build settings. When a target is created in an Xcode project file, it gets an associated build setting environment. This is created by taking a template of settings based on the type of target it is. These include values that allow for the configuration of language, compiler, linker, deployment, and packaging options for building a particular product. This level of build setting inherits from values set on the project level, but will over-ride any duplicated values in the same way that the project level does to the platform level.

xcconfig file

The last layer of build settings is defined with xcconfig files. Unlike the other build setting levels, this exists outside of the Xcode project file. An xcconfig file is a plain-text file that resides on disk that the build system will read values out of. This level of build settings was added to Xcode 3 to support more complex and conditional behavior in resolving values to use during a build. There is built-in conditional support for variance based on SDK, architecture, and build configuration. These are often used when using the same target to build against multiple platforms. Xcconfig files are also used to consolidate build settings into a single place that can be reused by many targets.

Output

Build Location

The build location is one of Xcode’s configurable preferences that dictates the resolution of the directory that will be used to perform a build in. The specifics of this are described in more detail in the “Managing Xcode” post linked at the top of this page. This directory is used to store the output of the build process.

Build Artifacts and Intermediates

When performing a build, it doesn’t generate an application binary directly from the source code files. There are many intermediary steps that it must go through first. The build system will produce and store the artifacts and intermediate files inside the build location directory. Xcode breaks this down based on target and build configuration that the build action is performed. Many of the compiler and linker flags that add additional verbosity will write files here

to log output of a build. This is useful for debugging a build that fails inconsistantly, and for investigating linker errors.

Built Products

The final output binary of a build is called a “built product”. This is the app or library that the Xcode project target is setup to build. For Apple platforms this is the .app for an application or .framework/dylib/a for a library. For command line applications or raw executable binaries you will find this as the `$(PRODUCT_NAME)` file in the directory that has a name corresponding to the build configuration within the build directory.
