REFERENCE

Build settings reference

Active Build Action (ACTION)

A string identifying the build system action being performed.

Additional SDKs (ADDITIONAL_SDKS)

The locations of any sparse SDKs that should be layered on top of the one specified by Base SDK (SDKROOT). If more than one SDK is listed, the first one has highest precedence. Every SDK specified in this setting should be a "sparse" SDK, for example, not an SDK for an entire macOS release.

Alternate Install Group (ALTERNATE_GROUP)

The group name or gid for the files listed under the Alternate Permissions Files (ALTERNATE_PERMISSIONS_FILES) setting.

Alternate Install Permissions (ALTERNATE_MODE)

Permissions used for the files listed under the Alternate Permissions Files (ALTERNATE_PERMISSIONS_FILES) setting.

Alternate Install Owner (ALTERNATE_OWNER)

The owner name or uid for the files listed under the Alternate Permissions Files (ALTERNATE_PERMISSIONS_FILES) setting.

Alternate Permissions Files (ALTERNATE_PERMISSIONS_FILES)

List of files to which the alternate owner, group and permissions are applied.

Always Embed Swift Standard Libraries (ALWAYS_EMBED_SWIFT_STANDARD_LIBRARIES)

Always embed the Swift standard libraries in the target's products, even if the target does not contain any Swift code. For example, this should be enabled if the target is embedding other products which contain Swift, or if it is a test target which does not contain Swift but which is testing a product which does. This setting only applies to wrapped products, not to standalone binary products.

Always Search User Paths (Deprecated) (ALWAYS_SEARCH_USER_PATHS)

This setting is deprecated as of Xcode 8.3 and may not be supported in future versions. It is recommended that you disable the setting.

If enabled, both `#include <header.h>`-style and `#include "header.h"`-style directives search the paths in User Header Search Paths (USER_HEADER_SEARCH_PATHS) before Header Search Paths (HEADER_SEARCH_PATHS). As a consequence, user headers, such as your own `String.h` header, have precedence over system headers when using `#include <header.h>`. This is done using the `-iquote` flag for

the paths provided in User Header Search Paths (USER_HEADER_SEARCH_PATHS). If disabled and your compiler fully supports separate user paths, user headers are only accessible with `#include "header.h"`-style preprocessor directives.

For backwards compatibility reasons, this setting is enabled by default. Disabling it is strongly recommended.

Require Only App-Extension-Safe API (APPLICATION_EXTENSION_API_ONLY)

When enabled, this causes the compiler and linker to disallow use of APIs that are not available to app extensions and to disallow linking to frameworks that have not been built with this setting enabled.

Convert Copied Files (APPLY_RULES_IN_COPY_FILES)

Enabling this setting will cause files in the target's Copy Files build phases to be processed by build rules. For example, property list files (`.plist`) and strings files will be converted as specified by Property List Output Encoding (PLIST_FILE_OUTPUT_FORMAT) and Strings File Output Encoding (STRINGS_FILE_OUTPUT_ENCODING), respectively.

Process Header Files (APPLY_RULES_IN_COPY_HEADERS)

Enabling this setting will cause all Public and Private headers in the target's Copy Headers build phase to be processed by build rules. This allows custom build rules to be defined to process these headers. Custom script rules can define their outputs relative to `HEADER_OUTPUT_DIR`, which will be provided to that script, taking the header visibility into account. The scripts are also passed `SCRIPT_HEADER_VISIBILITY` ("public" or "private"). Files that should not be processed by build rules may need to be moved to a Copy Files build phase when this setting is enabled.

Architectures (ARCHS)

A list of the architectures for which the product will be built. This is usually set to a predefined build setting provided by the platform. If more than one architecture is specified, a universal binary will be produced.

Asset Catalog App Icon Set Name (ASSETCATALOG_COMPILER_APPICON_NAME)

Name of an asset catalog app icon set whose contents will be merged into the `Info.plist`.

Watch Complication Name (ASSETCATALOG_COMPILER_COMPLICATION_NAME)

The name of a watch complication to use from the asset catalog.

Asset Catalog Launch Image Set Name (ASSETCATALOG_COMPILER_LAUNCHIMAGE_NAME)

Name of an asset catalog launch image set whose contents will be merged into the `Info.plist`.

Leaderboard Identifier Prefix (ASSETCATALOG_COMPILER_LEADERBOARD_IDENTIFIER_PREFIX)

Leaderboards in the asset catalog may optionally specify a Game Center identifier. If they do not, their name will be prefixed by this value to form an automatically generated identifier.

Leaderboard Set Identifier Prefix (ASSETCATALOG_COMPILER_LEADERBOARD_SET_IDENTIFIER_PREFIX)

Leaderboard sets in the asset catalog may optionally specify a Game Center identifier. If they do not, their name will be prefixed by this value to form an automatically generated identifier.

Optimization (ASSETCATALOG_COMPILER_OPTIMIZATION)

With no value, the compiler uses the default optimization. You can also specify `time` to optimize for speed of access or `space` to optimize for a smaller compiled asset catalogs.

Sticker Pack Identifier Prefix (ASSETCATALOG_COMPILER_STICKER_PACK_IDENTIFIER_PREFIX)

Sticker Packs in the asset catalog may optionally specify an identifier. If they do not, their name will be prefixed by this value to form an automatically generated identifier.

Show Notices (ASSETCATALOG_NOTICES)

Show notices encountered during the compilation of asset catalogs.

Asset Catalog Other Flags (ASSETCATALOG_OTHER_FLAGS)

Pass additional flags through to the asset catalog compiler.

Show Warnings (ASSETCATALOG_WARNINGS)

Show warnings encountered during the compilation of asset catalogs.

Asset Pack Manifest URL Prefix (ASSET_PACK_MANIFEST_URL_PREFIX)

If set to anything other than the empty string, every URL in the `AssetPackManifest.plist` file will consist of this string with the name of the asset pack appended. If not set, the URLs in the `AssetPackManifest.plist` will be formed as appropriate for the build location of the asset packs. The prefix string is not escaped or quoted in any way, so any necessary escaping must be part of the URL string. This setting affects only URLs in the `AssetPackManifest.plist` file — it does not affect where asset packs are built in the local file system.

Active Build Components (BUILD_COMPONENTS)

A list of components being built during this action.

Build Libraries for Distribution (BUILD_LIBRARY_FOR_DISTRIBUTION)

Ensures that your libraries are built for distribution. For Swift, this enables support for library evolution and generation of a module interface file.

Build Variants (BUILD_VARIANTS)

A list of the build variants of the linked binary that will be produced. By default, only the `normal` variant is produced. Other common values include `debug` and `profile`.

BUILT_PRODUCTS_DIR

Identifies the directory under which all the product's files can be found. This directory contains either product files or symbolic links to them. Run Script build phases can use the value of this build setting as a convenient way to refer to the product files built by one or more targets even when these files are scattered throughout a directory hierarchy (for example, when Deployment Location (DEPLOYMENT_LOCATION) is set to `YES`.

Bundle Loader (BUNDLE_LOADER)

Specifies the executable that will load the bundle output file being linked. Undefined symbols from the bundle are checked against the specified executable as if it is one of the dynamic libraries the bundle was linked with.

Enable C++ Container Overflow Checks (CLANG_ADDRESS_SANITIZER_CONTAINER_OVERFLOW)

Check for C++ container overflow when Address Sanitizer is enabled. This check requires the entire application to be built with Address Sanitizer. If not, it may report false positives.

Allow Non-modular Includes In Framework Modules (CLANG_ALLOW_NON_MODULAR_INCLUDES_IN_FRAMEWORK_MODULES)

Enabling this setting allows non-modular includes to be used from within framework modules. This is inherently unsafe, as such headers might cause duplicate definitions when used by any client that imports both the framework and the non-modular includes.

Dead Stores (CLANG_ANALYZER_DEADCODE_DEADSTORES)

Check for values stored to variables and never read again.

Misuse of Grand Central Dispatch (CLANG_ANALYZER_GCD)

Check for misuses of the Grand Central Dispatch API.

Performance Anti-Patterns with Grand Central Dispatch (CLANG_ANALYZER_GCD_PERFORMANCE)

Check for Grand Central Dispatch idioms that may lead to poor performance.

Violation of IOKit and libkern Reference Counting Rules (CLANG_ANALYZER_LIBKERN_RETAIN_COUNT)

Finds leaks and overreleases associated with objects inheriting from OSObject

Missing Localization Context Comment (CLANG_ANALYZER_LOCALIZABILITY_EMPTY_CONTEXT)

Warn when a call to an `NSLocalizedString()` macro is missing a context comment for the localizer.

Missing Localizability (CLANG_ANALYZER_LOCALIZABILITY_NONLOCALIZED)

Warn when a nonlocalized string is passed to a user interface method expecting a localized string.

Improper Memory Management (CLANG_ANALYZER_MEMORY_MANAGEMENT)

Warn about memory leaks, use-after-free, and other API misuses.

Violation of Mach Interface Generator Conventions (CLANG_ANALYZER_MIG_CONVENTIONS)

Warn when a MIG routine violates memory management conventions.

Misuse of 'nonnull' (CLANG_ANALYZER_NONNULL)

Check for misuses of `nonnull` parameter and return types.

Suspicious Conversions of NSNumber and CFNumberRef (CLANG_ANALYZER_NUMBER_OBJECT_CONVERSION)

Warn when a number object, such as an instance of `NSNumber`, `CFNumberRef`, `OSNumber`, or `OSBoolean` is compared or converted to a primitive value instead of another object.

@synchronized with nil mutex (CLANG_ANALYZER_OBJC_ATSYNC)

Warn on `nil` pointers used as mutexes for `@synchronized`.

Misuse of Collections API (CLANG_ANALYZER_OBJC_COLLECTIONS)

Warn if `CF` collections are created with non-pointer-size values. Check if `NS` collections are initialized with non-Objective-C type elements.

Improper Instance Cleanup in '-dealloc' (CLANG_ANALYZER_OBJC_DEALLOC)

Warn when an instance is improperly cleaned up in `–dealloc`.

Misuse of Objective-C generics (CLANG_ANALYZER_OBJC_GENERICS)

Warn if a specialized generic type is converted to an incompatible type.

Method Signatures Mismatch (CLANG_ANALYZER_OBJC_INCOMP_METHOD_TYPES)

Warn about Objective-C method signatures with type incompatibilities.

Improper Handling of CFError and NSError (CLANG_ANALYZER_OBJC_NSCFERROR)

Warn if functions accepting `CFErrorRef` or `NSError` cannot indicate that an error occurred.

Violation of Reference Counting Rules (CLANG_ANALYZER_OBJC_RETAIN_COUNT)

Warn on leaks and improper reference count management.

Violation of 'self = `super init`' Rule (CLANG_ANALYZER_OBJC_SELF_INIT)

Check that `super init` is properly called within an Objective-C initialization method.

Unused Ivars (CLANG_ANALYZER_OBJC_UNUSED_IVARS)

Warn about private ivars that are never used.

***EXPERIMENTAL* Buffer overflows (CLANG_ANALYZER_SECURITY_BUFFER_OVERFLOW_EXPERIMENTAL)**

Check for potential buffer overflows.

Floating Point Value Used as Loop Counter (CLANG_ANALYZER_SECURITY_FLOATLOOPCOUNTER)

Warn on using a floating point value as a loop counter (CERT: FLP30-C, FLP30-CPP).

Use of 'getpw', 'gets' (Buffer Overflow) (CLANG_ANALYZER_SECURITY_INSECUREAPI_GETPW_GETS)

Warn on uses of `getpw` and `gets`. The functions are dangerous as they may trigger a buffer overflow.

Use of 'mktemp' or Predictable 'mktemps' (CLANG_ANALYZER_SECURITY_INSECUREAPI_MKSTEMP)

Warn on uses of `mktemp`, which produces predictable temporary files. It is obsoleted by `mktemps`. Warn when `mkstemp` is passed fewer than 6 `X`'s in the format string.

Use of 'rand' Functions (CLANG_ANALYZER_SECURITY_INSECUREAPI_RAND)

Warn on uses of `rand`, `random`, and related functions, which produce predictable random number sequences. Use `arc4random` instead.

Use of 'strcpy' and 'strcat' (CLANG_ANALYZER_SECURITY_INSECUREAPI_STRCPY)

Warn on uses of the `strcpy` and `strcat` functions, which can result in buffer overflows. Use `strlcpy` or `strlcat` instead.

Unchecked Return Values (CLANG_ANALYZER_SECURITY_INSECUREAPI_UNCHECKEDRETURN)

Warn on uses of sensitive functions whose return values must be always checked.

Use of 'vfork' (CLANG_ANALYZER_SECURITY_INSECUREAPI_VFORK)

Warn on uses of the `vfork` function, which is inherently insecure. Use the safer `posix_spawn` function instead.

Misuse of Keychain Services API (CLANG_ANALYZER_SECURITY_KEYCHAIN_API)

Check for leaks of keychain attribute lists and data buffers returned by the Keychain Services API.

Use-After-Move Errors in C++ (CLANG_ANALYZER_USE_AFTER_MOVE)

Warn when a C++ object is used after it has been moved from.

C++ Language Dialect (CLANG_CXX_LANGUAGE_STANDARD)

Choose a standard or non-standard C++ language dialect. Options include:

- *C++98:* Accept ISO C++ 1998 with amendments, but not GNU extensions. `-std=c++98`
- *GNU++98:* Accept ISO C++ 1998 with amendments and GNU extensions. `-std=gnu++98`
- *C++11:* Accept the ISO C++ 2011 standard with amendments, but not GNU extensions. `-std=c++11`
- *GNU++11:* Accept the ISO C++ 2011 standard with amendments and GNU extensions. `-std=gnu++11`
- *C++14:* Accept the ISO C++ 2014 standard with amendments, but not GNU extensions. `-std=c++14`
- *GNU++14:* Accept the ISO C++ 2014 standard with amendments and GNU extensions. `-std=gnu++14`
- *C++17:* Accept the ISO C++ 2017 standard with amendments, but not GNU extensions. `-std=c++17`
- *GNU++17:* Accept the ISO C++ 2017 standard with amendments and GNU extensions. `-std=gnu++17`
- *Compiler Default:* Tells the compiler to use its default C++ language dialect. This is normally the best choice unless you have specific needs. (Currently equivalent to GNU++98.)

C++ Standard Library (CLANG_CXX_LIBRARY)

Choose a version of the C++ standard library to use.

- *libstdc++:* A traditional C++ standard library that works with GCC and Clang (default).
- *libc++:* A highly optimized C++ standard library that works only with Clang, and is designed to support new C++11 features.

Debug Information Level (CLANG_DEBUG_INFORMATION_LEVEL)

Toggles the amount of debug information emitted when debug symbols are enabled. This can impact the size of the generated debug information, which may matter in some cases for large projects, such as when using LTO.

Enable Code Coverage Support (CLANG_ENABLE_CODE_COVERAGE)

Enables building with code coverage instrumentation. This is only used when the build has code coverage enabled, which is typically done via the Xcode scheme settings.

Destroy Static Objects (CLANG_ENABLE_CPP_STATIC_DESTRUCTORS)

Controls whether variables with static or thread storage duration should have their exit-time destructors run.

Enable Modules (C and Objective-C) (CLANG_ENABLE_MODULES)

Enables the use of modules for system APIs. System headers are imported as semantic modules instead of raw headers. This can result in faster builds and project indexing.

Enable Clang Module Debugging (CLANG_ENABLE_MODULE_DEBUGGING)

When this setting is enabled, `clang` will use the shared debug info available in `clang` modules and precompiled headers. This results in smaller build artifacts, faster compile times, and more complete debug info. This setting should only be disabled when building static libraries with debug info for distribution.

Objective-C Automatic Reference Counting (CLANG_ENABLE_OBJC_ARC)

Compiles reference-counted Objective-C code (when garbage collection is not enabled) to use Automatic Reference Counting. Code compiled using automated reference counting is compatible with other code (such as frameworks) compiled using either manual reference counting (for example, traditional `retain` and `release` messages) or automated reference counting. Using this mode is currently incompatible with compiling code to use Objective-C Garbage Collection.

Weak References in Manual Retain Release (CLANG_ENABLE_OBJC_WEAK)

Compiles Objective-C code to enable weak references for code compiled with manual retain release (MRR) semantics.

Implicitly Link Objective-C Runtime Support (CLANG_LINK_OBJC_RUNTIME)

When linking a target using Objective-C code, implicitly link in Foundation (and if deploying back to an older OS) a backwards compatibility library to allow newer language features to run on an OS where the runtime support is not natively available. Most targets that use Objective-C should use this, although there are rare cases where a target should opt out of this behavior.

Add attribute annotations (CLANG_MIGRATOR_ANNOTATIONS)

Add attribute annotations to properties and methods.

Infer instancetype for method result type (CLANG_MIGRATOR_INSTANCE_TYPE)

Infer `instancetype` for method result type instead of `id`.

Use NS_ENUM/NS_OPTIONS macros (CLANG_MIGRATOR_NSENUM_MACROS)

Use `NS_ENUM/NS_OPTIONS` macros for enumerators.

Infer designated initializer methods (CLANG_MIGRATOR_OBJC_DESIGNATED_INIT)

Infer `NS_DESIGNATED_INITIALIZER` for designated initializer methods.

ObjC literals (CLANG_MIGRATOR_OBJC_LITERALS)

Enable migration to modern ObjC literals syntax.

ObjC subscripting (CLANG_MIGRATOR_OBJC_SUBSCRIPTING)

Enable migration to modern ObjC subscripting syntax.

Atomicity of inferred properties (CLANG_MIGRATOR_PROPERTY_ATOMICITY)

Choose the atomicity of the inferred properties.

ObjC property-dot syntax (CLANG_MIGRATOR_PROPERTY_DOT_SYNTAX)

Enable migration of setter/getter messages to property-dot syntax.

Infer protocol conformance (CLANG_MIGRATOR_PROTOCOL_CONFORMANCE)

Infer protocol conformance from the interface methods.

Only modify public headers (CLANG_MIGRATOR_PUBLIC_HEADERS_ONLY)

Only modify public headers of a target.

Infer readonly properties (CLANG_MIGRATOR_READONLY_PROPERTY)

Infer readonly properties from getter methods.

Infer readwrite properties (CLANG_MIGRATOR_READWRITE_PROPERTY)

Infer readwrite properties from a getter and setter method.

Link Frameworks Automatically (CLANG_MODULES_AUTOLINK)

Automatically link SDK frameworks that are referenced using `#import` or `#include`. This feature requires also enabling support for modules. This build setting only applies to C-family languages.

Disable Private Modules Warnings (CLANG_MODULES_DISABLE_PRIVATE_WARNING)

Disable warnings related to the recommended use of private module naming. This only makes sense when support for modules is enabled.

Optimization Profile File (CLANG_OPTIMIZATION_PROFILE_FILE)

The path to the file of the profile data to use when Use Optimization Profile (CLANG_USE_OPTIMIZATION_PROFILE) is enabled.

Mode of Analysis for 'Build' (CLANG_STATIC_ANALYZER_MODE)

The depth the static analyzer uses during the Build action. Use `Deep` to exercise the full power of the analyzer. Use `Shallow` for faster analysis.

Mode of Analysis for 'Analyze' (CLANG_STATIC_ANALYZER_MODE_ON_ANALYZE_ACTION)

The depth the static analyzer uses during the Analyze action. Use `Deep` to exercise the full power of the analyzer. Use `Shallow` for faster analysis.

Trivial automatic variable initialization (CLANG_TRIVIAL_AUTO_VAR_INIT)

Specify whether stack variables should be uninitialized, which can cause inadvertent information disclosure when uninitialized stack variables are used, or whether they should be pattern-initialized.

Enable Extra Integer Checks (CLANG_UNDEFINED_BEHAVIOR_SANITIZER_INTEGER)

Check for unsigned integer overflow, in addition to checks for signed integer overflow.

Enable Nullability Annotation Checks (CLANG_UNDEFINED_BEHAVIOR_SANITIZER_NULLABILITY)

Check for violations of nullability annotations in function calls, return statements, and assignments.

Use Optimization Profile (CLANG_USE_OPTIMIZATION_PROFILE)

When this setting is enabled, `clang` will use the optimization profile collected for a target when building it.

Out-of-Range Enum Assignments (CLANG_WARN_ASSIGN_ENUM)

Warn about assigning integer constants to enum values that are out of the range of the enumerated type.

Usage of implicit sequentially-consistent atomics (CLANG_WARN_ATOMIC_IMPLICIT_SEQ_CST)

Warns when an atomic is used with an implicitly sequentially-consistent memory order, instead of explicitly specifying memory order.

Block Capture of Autoreleasing (CLANG_WARN_BLOCK_CAPTURE_AUTORELEASING)

Warn about block captures of implicitly autoreleasing parameters.

Implicit Boolean Conversions (CLANG_WARN_BOOL_CONVERSION)

Warn about implicit conversions to boolean values that are suspicious. For example, writing `if (foo)` where `foo` is the name a function will trigger a warning.

Suspicious Commas (CLANG_WARN_COMMA)

Warn about suspicious uses of the comma operator.

Implicit Constant Conversions (CLANG_WARN_CONSTANT_CONVERSION)

Warn about implicit conversions of constant values that cause the constant value to change, either through a loss of precision, or entirely in its meaning.

Using C++11 extensions in earlier versions of C++ (CLANG_WARN_CXX0X_EXTENSIONS)

When compiling C++ code using a language standard older than C++11, warn about the use of C++11 extensions.

Deleting Instance of Polymorphic Class with No Virtual Destructor (CLANG_WARN_DELETE_NON_VIRTUAL_DTOR)

Warn when deleting an instance of a polymorphic class with virtual functions but without a virtual destructor.

Overriding Deprecated Objective-C Methods (CLANG_WARN_DEPRECATED_OBJC_IMPLEMENTATIONS)

Warn if an Objective-C class either subclasses a deprecated class or overrides a method that has been marked deprecated or unavailable.

Direct usage of 'isa' (CLANG_WARN_DIRECT_OBJC_ISA_USAGE)

Warn about direct accesses to the Objective-C `isa` pointer instead of using a runtime API.

Documentation Comments (CLANG_WARN_DOCUMENTATION_COMMENTS)

Warns about issues in documentation comments (`doxygen`-style) such as missing or incorrect documentation tags.

Empty Loop Bodies (CLANG_WARN_EMPTY_BODY)

Warn about loop bodies that are suspiciously empty.

Implicit Enum Conversions (CLANG_WARN_ENUM_CONVERSION)

Warn about implicit conversions between different kinds of enum values. For example, this can catch issues when using the wrong enum flag as an argument to a function or method.

Implicit Float Conversions (CLANG_WARN_FLOAT_CONVERSION)

Warn about implicit conversions that turn floating-point numbers into integers.

Implicit Signedness Conversions (CLANG_WARN_IMPLICIT_SIGN_CONVERSION)

Warn about implicit integer conversions that change the signedness of an integer value.

Infinite Recursion (CLANG_WARN_INFINITE_RECURSION)

Warn if all paths through a function call itself.

Implicit Integer to Pointer Conversions (CLANG_WARN_INT_CONVERSION)

Warn about implicit conversions between pointers and integers. For example, this can catch issues when one incorrectly intermixes using `NSNumber*`'s and raw integers.

Missing Noescape Annotation (CLANG_WARN_MISSING_NOESCAPE)

Warn about noescape annotations that are missing in a method's signature.

Implicit Non-Literal Null Conversions (CLANG_WARN_NON_LITERAL_NULL_CONVERSION)

Warn about non-literal expressions that evaluate to zero being treated as a null pointer.

Incorrect Uses of Nullable Values (CLANG_WARN_NULLABLE_TO_NONNULL_CONVERSION)

Warns when a nullable expression is used somewhere it's not allowed, such as when passed as a `_Nonnull` parameter.

Implicit ownership types on out parameters (CLANG_WARN_OBJC_EXPLICIT_OWNERSHIP_TYPE)

Warn about implicit ownership types on Objective-C object references as out parameters. For example, declaring a parameter with type `NSObject**` will produce a warning because the compiler will assume that the out parameter's ownership type is `__autoreleasing`.

Implicit Atomic Objective-C Properties (CLANG_WARN_OBJC_IMPLICIT_ATOMIC_PROPERTIES)

Warn about `@property` declarations that are implicitly atomic.

Implicit retain of 'self' within blocks (CLANG_WARN_OBJC_IMPLICIT_RETAIN_SELF)

Warn about implicit retains of `self` within blocks, which can create a retain-cycle.

Interface Declarations of Instance Variables (CLANG_WARN_OBJC_INTERFACE_IVARS)

Warn about instance variable declarations in `@interface`.

Implicit Objective-C Literal Conversions (CLANG_WARN_OBJC_LITERAL_CONVERSION)

Warn about implicit conversions from Objective-C literals to values of incompatible type.

Implicit Synthesized Properties (CLANG_WARN_OBJC_MISSING_PROPERTY_SYNTHESIS)

Starting in Xcode 4.4, Apple Clang will implicitly synthesize properties that are not explicitly synthesized using `@synthesize`. This setting warns about such implicit behavior, even though the property is still synthesized. This is essentially a backwards compatibility warning, or for those who wish to continue to explicitly use `@synthesize`.

Repeatedly using a __weak reference (CLANG_WARN_OBJC_REPEATED_USE_OF_WEAK)

Warn about repeatedly using a weak reference without assigning the weak reference to a strong reference. This is often symptomatic of a race condition where the weak reference can become `nil` between accesses, resulting in unexpected behavior. Assigning to temporary strong reference ensures the object stays alive during the related accesses.

Unintentional Root Class (CLANG_WARN_OBJC_ROOT_CLASS)

Warn about classes that unintentionally do not subclass a root class, such as `NSObject`.

Suspicious Pragma Pack (CLANG_WARN_PRAGMA_PACK)

Warn when a translation unit is missing terminating '#pragma pack (pop)' directives or when the '#pragma pack' state immediately after an #include is different from the state immediately before.

Outdated Private Module Map (CLANG_WARN_PRIVATE_MODULE)

Warn about private modules that do not use the recommended private module layout.

Quoted Include In Framework Header (CLANG_WARN_QUOTED_INCLUDE_IN_FRAMEWORK_HEADER)

Warns when a quoted include is used instead of a framework style include in a framework header.

Range-based For Loops (CLANG_WARN_RANGE_LOOP_ANALYSIS)

Warn about ranged-based for loops.

Semicolon Before Method Body (CLANG_WARN_SEMICOLON_BEFORE_METHOD_BODY)

Warn about ignored semicolon between a method implementation's signature and body.

Strict Prototypes (CLANG_WARN_STRICT_PROTOTYPES)

Warn about non-prototype declarations.

Suspicious Implicit Conversions (CLANG_WARN_SUSPICIOUS_IMPLICIT_CONVERSION)

Warn about various implicit conversions that can lose information or are otherwise suspicious.

Suspicious Moves (CLANG_WARN_SUSPICIOUS_MOVE)

Warn about suspicious uses of `std::move`.

Unguarded availability (CLANG_WARN_UNGUARDED_AVAILABILITY)

Warn if an API that is newer than the deployment target is used without "if (@available(...))" guards.

Unreachable Code (CLANG_WARN_UNREACHABLE_CODE)

Warns about potentially unreachable code.

Ambiguous C++ Parsing Situation (CLANG_WARN_VEXING_PARSE)

Warn about a parsing ambiguity between a variable declaration and a function-style cast.

Using __bridge Casts Outside of ARC (CLANG_WARN__ARC_BRIDGE_CAST_NONARC)

Warn about using `__bridge` casts when not using ARC, where they have no effect.

Duplicate Method Definitions (CLANG_WARN__DUPLICATE_METHOD_MATCH)

Warn about declaring the same method more than once within the same `@interface`.

Exit-Time C++ Destructors (CLANG_WARN__EXIT_TIME_DESTRUCTORS)

Warn about destructors for C++ objects that are called when an application is terminating.

Enable Additional Vector Extensions (CLANG_X86_VECTOR_INSTRUCTIONS)

Enables the use of extended vector instructions. Only used when targeting Intel architectures.

Code Signing Entitlements (CODE_SIGN_ENTITLEMENTS)

The path to a file specifying code-signing entitlements.

Code Signing Identity (CODE_SIGN_IDENTITY)

The name, also known as the *common name*, of a valid code-signing certificate in a keychain within your keychain path. A missing or invalid certificate will cause a build error.

Code Signing Inject Base Entitlements (CODE_SIGN_INJECT_BASE_ENTITLEMENTS)

Automatically inject entitlements from the platform's BaseEntitlements.plist into the code signatures of executables.

Code Sign Style (CODE_SIGN_STYLE)

This setting specifies the method used to acquire and locate signing assets. Choose `Automatic` to let Xcode automatically create and update profiles, app IDs, and certificates. Choose `Manual` to create and update these yourself on the developer website.

COMBINE_HIDPI_IMAGES

Combines image files at different resolutions into one multi-page TIFF file that is HiDPI compliant for macOS 10.7 and later. Only image files in the same directory and with the same base name and extension are combined. The file names must conform to the naming convention used in HiDPI.

Enable Index-While-Building Functionality (COMPILER_INDEX_STORE_ENABLE)

Control whether the compiler should emit index data while building.

Compress PNG Files (COMPRESS_PNG_FILES)

If enabled, PNG resource files are compressed as they are copied.

CONFIGURATION

Identifies the build configuration, such as `Debug` or `Release`, that the target uses to generate the product.

Per-configuration Build Products Path (CONFIGURATION_BUILD_DIR)

The base path where build products will be placed during a build for a given configuration. By default, this is set to `$(BUILD_DIR)/$(CONFIGURATION)`.

Per-configuration Intermediate Build Files Path (CONFIGURATION_TEMP_DIR)

The base path where intermediates will be placed during a build for a given configuration. By default, this is set to `$(PROJECT_TEMP_DIR)/$(CONFIGURATION)`.

CONTENTS_FOLDER_PATH

Specifies the directory inside the generated bundle that contains the product's files.

Preserve HFS Data (COPYING_PRESERVES_HFS_DATA)

Causes the copying of resources to preserve resource forks and Finder info.

Run unifdef on Product Headers (COPY_HEADERS_RUN_UNIFDEF)

If enabled, headers are run through the `unifdef(1)` tool when copied to the product.

Unifdef Flags for Product Headers (COPY_HEADERS_UNIFDEF_FLAGS)

Specifies the flags to pass to `unifdef(1)` when invoking that tool to copy headers. This setting has no effect unless Run unifdef on Product Headers (COPY_HEADERS_RUN_UNIFDEF) is enabled.

Strip Debug Symbols During Copy (COPY_PHASE_STRIP)

Specifies whether binary files that are copied during the build, such as in a Copy Bundle Resources or Copy Files build phase, should be stripped of debugging symbols. It does not cause the linked product of a target to be stripped—use Strip Linked Product (STRIP_INSTALLED_PRODUCT) for that.

CoreML Model Class Generation Language (COREML_CODEGEN_LANGUAGE)

The Source-code language to use for generated CoreML model class. By default "Automatic" will analyze your project to determine the correct language. Adjust this setting to explicitly select "Swift" or "Objective-C", or select "None" to disable model class generation.

CoreML Generated Model Inherits NSObject (COREML_CODEGEN_SWIFT_GLOBAL_MODULE)

Generate Swift model classes that are marked with @objc and are descendants of NSObject, in order to be accessible and usable in Objective-C. This setting has no effect if "CoreML Model Class Generation Language" is set to "Objective-C".

Cpp Other Preprocessor Flags (CPP_OTHER_PREPROCESSOR_FLAGS)

Other flags to pass to the C preprocessor when using the standalone C Preprocessor rule.

Cpp Prefix File (CPP_PREFIX_HEADER)

Implicitly include the given file when preprocessing using the standalone C Preprocessor rule. The path given should either be a project relative path or an absolute path.

Cpp Preprocessor Definitions (CPP_PREPROCESSOR_DEFINITIONS)

Space-separated list of preprocessor macros of the form `foo` or `foo=bar`. These macros are used when preprocessing using the standalone C Preprocessor rule.

Create Info.plist Section in Binary (CREATE_INFOPLIST_SECTION_IN_BINARY)

Enabling this setting will create a section in the product's linked binary containing the processed `Info.plist` file for the target. This setting only applies to command-line tool targets.

CURRENT_ARCH

The name of the active architecture being processed.

Current Project Version (CURRENT_PROJECT_VERSION)

This setting defines the current version of the project. The value must be a integer or floating point number, such as `57` or `365.8`.

CURRENT_VARIANT

The name of the active variant being processed.

Dead Code Stripping (DEAD_CODE_STRIPPING)

Activating this setting causes the `-dead_strip` flag to be passed to `ld(1)` via `cc(1)` to turn on dead code stripping.

Debug Information Format (DEBUG_INFORMATION_FORMAT)

The type of debug information to produce.

- *DWARF:* Object files and linked products will use DWARF as the debug information format. `dwarf`
- *DWARF with dSYM File:* Object files and linked products will use DWARF as the debug information format, and Xcode will also produce a dSYM file containing the debug information from the individual object files (except that a dSYM file is not needed and will not be created for static library or object file products). `dwarf-with-dsym`

Defines Module (DEFINES_MODULE)

If enabled, the product will be treated as defining its own module. This enables automatic production of LLVM module map files when appropriate, and allows the product to be imported as a module.

Deployment Location (DEPLOYMENT_LOCATION)

If enabled, built products are placed in their installed locations in addition to the built products folder.

Deployment Postprocessing (DEPLOYMENT_POSTPROCESSING)

If enabled, indicates that binaries should be stripped and file mode, owner, and group information should be set to standard values.

DERIVED_FILE_DIR

Identifies the directory into which derived source files, such as those generated by `lex` and `yacc`, are placed.

Derive Mac Catalyst Product Bundle Identifier (DERIVE_MACCATALYST_PRODUCT_BUNDLE_IDENTIFIER)

When enabled, Xcode will automatically derive a bundle identifier for this target from its original bundle identifier when it's building for Mac Catalyst.

Development Assets (DEVELOPMENT_ASSET_PATHS)

Files and directories used only for development. Archive and install builds will exclude this content.

Development Team (DEVELOPMENT_TEAM)

The team ID of a development team to use for signing certificates and provisioning profiles.

Output Log Level (DISTILL_LOGLEVEL)

This option controls the output of the `distill` tool.

Distillation Mode (DISTILL_MODE)

This option controls the mode under which the `distill` tool is run—either it can regenerate its output entirely or it can incrementally edit the existing output.

DOCUMENTATION_FOLDER_PATH

Identifies the directory that contains the bundle's documentation files.

Don't Force Info.plist Generation (DONT_GENERATE_INFOPLIST_FILE)

If enabled, don't automatically generate an Info.plist file for wrapped products when the Info.plist File (INFOPLIST_FILE) build setting is empty.

Installation Build Products Location (DSTROOT)

The path at which all products will be rooted when performing an install build. For instance, to install your products on the system proper, set this path to `/`. Defaults to `/tmp/$(PROJECT_NAME).dst` to prevent a *test* install build from accidentally overwriting valid and needed data in the ultimate install path.

Typically this path is not set per target, but is provided as an option on the command line when performing an `xcodebuild install`. It may also be set in a build configuration in special circumstances.

Other DTrace Flags (DTRACE_OTHER_FLAGS)

Space-separated list of additional flags to pass to the `dtrace` compiler. Be sure to backslash-escape any arguments that contain spaces or special characters, such as path names that may contain spaces. Use this setting if Xcode does not already provide UI for a particular `dtrace` flag.

Compatibility Version (DYLIB_COMPATIBILITY_VERSION)

Determines the compatibility version of the resulting library, bundle, or framework binary. See Dynamic Library Design Guidelines in Dynamic Library Programming Topics for details on assigning version numbers of dynamic libraries.

Current Library Version (DYLIB_CURRENT_VERSION)

This setting defines the current version of any framework built by the project. As with Current Project Version (CURRENT_PROJECT_VERSION), the value must be an integer or floating point number, such as `57` or `365.8`. By default, it is set to `$(CURRENT_PROJECT_VERSION)`. See Dynamic Library Design Guidelines in Dynamic Library Programming Topics for details on assigning version numbers of dynamic libraries.

Dynamic Library Install Name Base (DYLIB_INSTALL_NAME_BASE)

Sets the base value for the internal `install path` (`LC_ID_DYLIB`) in a dynamic library. This will be combined with the EXECUTABLE_PATH to form the full install path. Setting Dynamic Library Install Name (LD_DYLIB_INSTALL_NAME) directly will override this setting. This setting defaults to the target's Installation Directory (INSTALL_PATH). It is ignored when building any product other than a dynamic library.

Embed Asset Packs In Product Bundle (EMBED_ASSET_PACKS_IN_PRODUCT_BUNDLE)

Embed all the built asset packs inside the product bundle. Since this negates the performance benefits of the On Demand Resources feature, it is only useful for testing purposes when it isn't practical to use an asset pack server.

Enable Bitcode (ENABLE_BITCODE)

Activating this setting indicates that the target or project should generate bitcode during compilation for platforms and architectures that support it. For Archive builds, bitcode will be generated in the linked binary for submission to the App Store. For other builds, the compiler and linker will check whether the code complies with the requirements for bitcode generation, but will not generate actual bitcode.

Enable Hardened Runtime (ENABLE_HARDENED_RUNTIME)

Enable hardened runtime restrictions.

ENABLE_HEADER_DEPENDENCIES

Specifies whether to automatically track dependencies on included header files.

Enable Incremental Distill (ENABLE_INCREMENTAL_DISTILL)

Enabled the incremental `distill` option in the asset catalog compiler. This feature is experimental and should only be enabled with caution.

Enable Foundation Assertions (ENABLE_NS_ASSERTIONS)

Controls whether assertion logic provided by `NSAssert` is included in the preprocessed source code or is elided during preprocessing. Disabling assertions can improve code performance.

Build Active Resources Only (ENABLE_ONLY_ACTIVE_RESOURCES)

Omit inapplicable resources when building for a single device. For example, when building for a device with a Retina display, exclude 1x resources.

Enable On Demand Resources (ENABLE_ON_DEMAND_RESOURCES)

If enabled, tagged assets—files and asset catalog entries—are built into asset packs based on their combination of tags. Untagged resources are treated normally.

Enable Previews (ENABLE_PREVIEWS)

If enabled, the product will be built with options appropriate for supporting previews.

Enable Strict Checking of objc_msgSend Calls (ENABLE_STRICT_OBJC_MSGSEND)

Controls whether `objc_msgSend` calls must be cast to the appropriate function pointer type before being called.

Enable Testability (ENABLE_TESTABILITY)

When this setting is activated, the product will be built with options appropriate for running automated tests, such as making private interfaces accessible to the tests. This may result in tests running slower than they would without testability enabled.

Sub-Directories to Exclude in Recursive Searches (EXCLUDED_RECURSIVE_SEARCH_PATH_SUBDIRECTORIES)

This is a list of `fnmatch()`-style patterns of file or directory names to exclude when performing a recursive search. By default, this is set to `*.nib *.lproj *.framework *.gch *.xcode (*) .DS_Store CVS .svn .git .hg`. Normally, if you override this value you should include the default values via the `$(inherited)` macro.

Excluded Source File Names (EXCLUDED_SOURCE_FILE_NAMES)

A list of patterns (as defined by `fnmatch(3)`) specifying the names of source files to explicitly *exclude* when processing the files in the target's build phases (see also Included Source File Names (INCLUDED_SOURCE_FILE_NAMES)). This setting can be used to define complex filters for which files from the phase should be built in response to other build settings; for example, a value of `*.$(CURRENT_ARCH).c` could serve to exclude particular files based on the architecture being built.

EXECUTABLES_FOLDER_PATH

Identifies the directory that contains additional binary files.

Executable Extension (EXECUTABLE_EXTENSION)

This is the extension used for the executable product generated by the target, which has a default value based on the product type.

EXECUTABLE_FOLDER_PATH

Identifies the directory that contains the binary the target builds.

EXECUTABLE_NAME

Specifies the name of the binary the target produces.

EXECUTABLE_PATH

Specifies the path to the binary the target produces within its bundle.

Executable Prefix (EXECUTABLE_PREFIX)

The prefix used for the executable product generated by the target, which has a default value based on the product type.

EXECUTABLE_SUFFIX

Specifies the suffix of the binary filename, including the character that separates the extension from the rest of the bundle name.

Exported Symbols File (EXPORTED_SYMBOLS_FILE)

This is a project-relative path to a file that lists the symbols to export. See `ld -exported_symbols_list` for details on exporting symbols.

FRAMEWORKS_FOLDER_PATH

Specifies the directory that contains the product's embedded frameworks.

Framework Search Paths (FRAMEWORK_SEARCH_PATHS)

This is a list of paths to folders containing frameworks to be searched by the compiler for both included or imported header files when compiling C, Objective-C, C++, or Objective-C++, and by the linker for frameworks used by the product. Paths are delimited by whitespace, so any paths with spaces in them must be properly quoted.

Framework Version (FRAMEWORK_VERSION)

Framework bundles are versioned by having contents in subfolders of a version folder that has links to the current version and its contents.

'char' Type Is Unsigned (GCC_CHAR_IS_UNSIGNED_CHAR)

Enabling this setting causes `char` to be unsigned by default, disabling it causes `char` to be signed by default.

CodeWarrior/MS-Style Inline Assembly (GCC_CW_ASM_SYNTAX)

Enable the CodeWarrior/Microsoft syntax for inline assembly code in addition to the standard GCC syntax.

C Language Dialect (GCC_C_LANGUAGE_STANDARD)

Choose a standard or non-standard C language dialect.

- *ANSI C:* Accept ISO C90 and ISO C++, turning off GNU extensions that are incompatible. `-ansi` Incompatible GNU extensions include the `asm`, `inline`, and `typeof` keywords (but not the equivalent `__asm__`, `__inline__`, and `__typeof__` forms), and the `//` syntax for comments. This setting also enables trigraphs.
- *C89:* Accept ISO C90 (1990), but not GNU extensions. `-std=c89`
- *GNU89:* Accept ISO C90 and GNU extensions. `-std=gnu89`
- *C99:* Accept ISO C99 (1999), but not GNU extensions. `-std=c99`
- *GNU99:* Accept ISO C99 and GNU extensions. `-std=gnu99`
- *C11:* Accept ISO C11 (2011), but not GNU extensions. `-std=c11`
- *GNU11:* Accept ISO C11 and GNU extensions. `-std=gnu11`

- *Compiler Default:* Tells the compiler to use its default C language dialect. This is normally the best choice unless you have specific needs. (Currently equivalent to GNU99.)

Generate Position-Dependent Code (GCC_DYNAMIC_NO_PIC)

Faster function calls for applications. Not appropriate for shared libraries, which need to be position-independent.

Allow 'asm', 'inline', 'typeof' (GCC_ENABLE_ASM_KEYWORD)

Controls whether `asm`, `inline`, and `typeof` are treated as keywords or whether they can be used as identifiers.

Recognize Builtin Functions (GCC_ENABLE_BUILTIN_FUNCTIONS)

Controls whether builtin functions that do not begin with `__builtin_` as prefix are recognized.

GCC normally generates special code to handle certain builtin functions more efficiently; for instance, calls to `alloca` may become single instructions that adjust the stack directly, and calls to `memcpy` may become inline copy loops. The resulting code is often both smaller and faster, but since the function calls no longer appear as such, you cannot set a breakpoint on those calls, nor can you change the behavior of the functions by linking with a different library. In addition, when a function is recognized as a builtin function, GCC may use information about that function to warn about problems with calls to that function, or to generate more efficient code, even if the resulting code still contains calls to that function. For example, warnings are given with `-Wformat` for bad calls to `printf`, when `printf` is built in, and `strlen` is known not to modify global memory.

Enable C++ Exceptions (GCC_ENABLE_CPP_EXCEPTIONS)

Enable C++ exception handling. Generates extra code needed to propagate exceptions. For some targets, this implies GCC will generate frame unwind information for all functions, which can produce significant data size overhead, although it does not affect execution. If you do not specify this option, GCC will enable it by default for languages like C++ that normally require exception handling, and disable it for languages like C that do not normally require it. However, you may need to enable this option when compiling C code that needs to interoperate properly with exception handlers written in C++.

Enable C++ Runtime Types (GCC_ENABLE_CPP_RTTI)

Enable generation of information about every class with virtual functions for use by the C++ runtime type identification features (`dynamic_cast` and `typeid`). If you don't use those parts of the language, you can save some space by using this flag. Note that exception handling uses the same information, but it will generate it as needed.

Enable Exceptions (GCC_ENABLE_EXCEPTIONS)

Enable exception handling. Generates extra code needed to propagate exceptions. For some targets, this implies GCC will generate frame unwind information for all functions, which can produce significant data size overhead, although it does not affect execution. If you do not specify this option, GCC will enable it by default for languages like C++ and Objective-C that normally require exception handling, and disable it for languages like C that do not normally require it. However, you may need to enable this option when compiling C code that needs to interoperate properly with exception handlers written in other languages. You may also wish to disable this option if you are compiling older programs that don't use exception handling.

Generate Floating Point Library Calls (GCC_ENABLE_FLOATING_POINT_LIBRARY_CALLS)

Generate output containing library calls for floating point.

Kernel Development Mode (GCC_ENABLE_KERNEL_DEVELOPMENT)

Activating this setting enables kernel development mode.

Enable Objective-C Exceptions (GCC_ENABLE_OBJC_EXCEPTIONS)

This setting enables `@try`/`@catch`/`@throw` syntax for handling exceptions in Objective-C code. Only applies to Objective-C.

Recognize Pascal Strings (GCC_ENABLE_PASCAL_STRINGS)

Recognize and construct Pascal-style string literals. Its use in new code is discouraged.

Pascal string literals take the form `"\pstring"` . The special escape sequence `\p` denotes the Pascal length byte for the string, and will be replaced at compile time with the number of characters that follow. The `\p` may only appear at the beginning of a string literal, and may not appear in wide string literals or as an integral constant.

Enable SSE3 Extensions (GCC_ENABLE_SSE3_EXTENSIONS)

Specifies whether the binary uses the builtin functions that provide access to the SSE3 extensions to the IA-32 architecture.

Enable SSE4.1 Extensions (GCC_ENABLE_SSE41_EXTENSIONS)

Specifies whether the binary uses the builtin functions that provide access to the SSE4.1 extensions to the IA-32 architecture.

Enable SSE4.2 Extensions (GCC_ENABLE_SSE42_EXTENSIONS)

Specifies whether the binary uses the builtin functions that provide access to the SSE4.2 extensions to the IA-32 architecture.

Enable Trigraphs (GCC_ENABLE_TRIGRAPHS)

Controls whether or not trigraphs are permitted in the source code.

Relax IEEE Compliance (GCC_FAST_MATH)

Enables some floating point optimizations that are not IEEE754-compliant, but which usually work. Programs that require strict IEEE compliance may not work with this option.

Generate Debug Symbols (GCC_GENERATE_DEBUGGING_SYMBOLS)

Enables or disables generation of debug symbols. When debug symbols are enabled, the level of detail can be controlled by the Debug Information Format (DEBUG_INFORMATION_FORMAT) setting.

Generate Legacy Test Coverage Files (GCC_GENERATE_TEST_COVERAGE_FILES)

Activating this setting causes a `notes` file to be produced that the `gcov` code-coverage utility can use to show program coverage.

Increase Sharing of Precompiled Headers (GCC_INCREASE_PRECOMPILED_HEADER_SHARING)

Enabling this option will enable increased sharing of precompiled headers among targets that share the same prefix header and precompiled header directory.

Xcode distinguishes between precompiled header (PCH) files by generating a hash value based on the command-line options to the compiler used to create the PCH. Enabling this option will exclude certain compiler options from that hash. Presently this option will exclude search path options (`–I`, `–iquote`, `–isystem`, `–F`, `–L`) from the hash.

Enabling increased sharing of PCH files carries some risk—if two targets use the same prefix header but have different include paths that cause the prefix header to include different files when they are precompiled, then subtle problems may result because one target will use a PCH that was built using files included by the other target. In this case, this option must be turned off in order to enforce correctness.

Inline Methods Hidden (GCC_INLINES_ARE_PRIVATE_EXTERN)

When enabled, out-of-line copies of inline methods are declared `private extern`.

Compile Sources As (GCC_INPUT_FILETYPE)

Specifies whether to compile each source file according to its file type, or whether to treat all source files in the target as if they are of a specific language.

Instrument Program Flow (GCC_INSTRUMENT_PROGRAM_FLOW_ARCS)

Activating this setting indicates that code should be added so program flow arcs are instrumented.

Enable Linking With Shared Libraries (GCC_LINK_WITH_DYNAMIC_LIBRARIES)

Enabling this option allows linking with the shared libraries. This is the default for most product types.

No Common Blocks (GCC_NO_COMMON_BLOCKS)

In C, allocate even uninitialized global variables in the data section of the object file, rather than generating them as common blocks. This has the effect that if the same variable is declared (without `extern`) in two different compilations, you will get an error when you link them.

Optimization Level (GCC_OPTIMIZATION_LEVEL)

Specifies the degree to which the generated code is optimized for speed and binary size.

- *None:* Do not optimize. `–O0` With this setting, the compiler's goal is to reduce the cost of compilation and to make debugging produce the expected results. Statements are independent—if you stop the program with a breakpoint between statements, you can then assign a new value to any variable or change the program counter to any other statement in the function and get exactly the results you would expect from the source code.

- *Fast:* Optimizing compilation takes somewhat more time, and a lot more memory for a large function. `–01` With this setting, the compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time. In Apple's compiler, strict aliasing, block reordering, and inter-block scheduling are disabled by default when optimizing.
- *Faster:* The compiler performs nearly all supported optimizations that do not involve a space-speed tradeoff. `–02` With this setting, the compiler does not perform loop unrolling or function inlining, or register renaming. As compared to the `Fast` setting, this setting increases both compilation time and the performance of the generated code.
- *Fastest:* Turns on all optimizations specified by the `Faster` setting and also turns on function inlining and register renaming options. This setting may result in a larger binary. `–03`
- *Fastest, Smallest:* Optimize for size. This setting enables all `Faster` optimizations that do not typically increase code size. It also performs further optimizations designed to reduce code size. `–0s`
- *Fastest, Aggressive Optimizations:* This setting enables `Fastest` but also enables aggressive optimizations that may break strict standards compliance but should work well on well-behaved code. `–0fast`
- *Smallest, Aggressive Size Optimizations:* This setting enables additional size savings by isolating repetitive code patterns into a compiler generated function. `–0z`

Precompile Prefix Header (GCC_PRECOMPILE_PREFIX_HEADER)

Generates a precompiled header for the prefix header, which should reduce overall build times.

Precompiling the prefix header will be most effective if the contents of the prefix header or any file it includes change rarely. If the contents of the prefix header or any file it includes change frequently, there may be a negative impact to overall build time.

Prefix Header (GCC_PREFIX_HEADER)

Implicitly include the named header. The path given should either be a project relative path or an absolute path.

Preprocessor Macros (GCC_PREPROCESSOR_DEFINITIONS)

Space-separated list of preprocessor macros of the form `foo` or `foo=bar`.

Preprocessor Macros Not Used In Precompiled Headers (GCC_PREPROCESSOR_DEFINITIONS_NOT_USED_IN_PRECOMPS)

Space-separated list of preprocessor macros of the form `foo` or `foo=bar`. These macros are not used when precompiling a prefix header file.

Make Strings Read-Only (GCC_REUSE_STRINGS)

Reuse string literals.

Short Enumeration Constants (GCC_SHORT_ENUMS)

Make enums only as large as needed for the range of possible values.

This setting generates code that may not binary compatible with code generated without this setting or with macOS frameworks.

Enforce Strict Aliasing (GCC_STRICT_ALIASING)

Optimize code by making more aggressive assumptions about whether pointers can point to the same objects as other pointers. Programs that use pointers a lot may benefit from this, but programs that don't strictly follow the ISO C rules about the type with which an object may be accessed may behave unexpectedly.

Symbols Hidden by Default (GCC_SYMBOLS_PRIVATE_EXTERN)

When enabled, all symbols are declared `private extern` unless explicitly marked to be exported using `__attribute__((visibility("default")))` in code. If not enabled, all symbols are exported unless explicitly marked as `private extern`. See Controlling Symbol Visibility in C++ Runtime Environment Programming Guide.

Statics are Thread-Safe (GCC_THREADSAFE_STATICS)

Emits extra code to use the routines specified in the C++ ABI for thread-safe initialization of local statics. You can disable this option to reduce code size slightly in code that doesn't need to be thread-safe.

Treat Missing Function Prototypes as Errors (GCC_TREAT_IMPLICIT_FUNCTION_DECLARATIONS_AS_ERRORS)

Causes warnings about missing function prototypes to be treated as errors. Only applies to C and Objective-C.

Treat Incompatible Pointer Type Warnings as Errors (GCC_TREAT_INCOMPATIBLE_POINTER_TYPE_WARNINGS_AS_ERRORS)

Enabling this option causes warnings about incompatible pointer types to be treated as errors.

Treat Warnings as Errors (GCC_TREAT_WARNINGS_AS_ERRORS)

Enabling this option causes all warnings to be treated as errors.

Unroll Loops (GCC_UNROLL_LOOPS)

Unrolls loops. Unrolling makes the code larger, but may make it faster by reducing the number of branches executed.

Use Standard System Header Directory Searching (GCC_USE_STANDARD_INCLUDE_SEARCHING)

Controls whether the standard system directories are searched for header files. When disabled, only the directories you have specified with `-I` options (and the directory of the current file, if appropriate) are searched.

Compiler for C/C++/Objective-C (GCC_VERSION)

The compiler to use for C, C++, and Objective-C.

Implicit Conversion to 32 Bit Type (GCC_WARN_64_TO_32_BIT_CONVERSION)

Warn if a value is implicitly converted from a 64-bit type to a 32-bit type. This is a subset of the warnings provided by -Wconversion.

Deprecated Functions (GCC_WARN_ABOUT_DEPRECATED_FUNCTIONS)

Warn about the use of deprecated functions, variables, and types (as indicated by the `deprecated` attribute).

Undefined Use of offsetof Macro (GCC_WARN_ABOUT_INVALID_OFFSETOF_MACRO)

Unchecking this setting will suppress warnings from applying the `offsetof` macro to a non-POD type. According to the 1998 ISO C++ standard, applying `offsetof` to a non-POD type is undefined. In existing C++ implementations, however, `offsetof` typically gives meaningful results even when applied to certain kinds of non-POD types, such as a simple struct that fails to be a POD type only by virtue of having a constructor. This flag is for users who are aware that they are writing nonportable code and who have deliberately chosen to ignore the warning about it.

The restrictions on `offsetof` may be relaxed in a future version of the C++ standard.

Missing Fields in Structure Initializers (GCC_WARN_ABOUT_MISSING_FIELD_INITIALIZERS)

Warn if a structure's initializer has some fields missing. For example, the following code would cause such a warning because `x.h` is implicitly zero:

```
struct s { int f, g, h; };
struct s x = { 3, 4 };
```

This option does not warn about designated initializers, so the following modification would not trigger a warning:

```
struct s { int f, g, h; };
struct s x = { .f = 3, .g = 4 };
```

Missing Newline At End Of File (GCC_WARN_ABOUT_MISSING_NEWLINE)

Warn when a source file does not end with a newline.

Missing Function Prototypes (GCC_WARN_ABOUT_MISSING_PROTOTYPES)

Causes warnings to be emitted about missing prototypes.

Pointer Sign Comparison (GCC_WARN_ABOUT_POINTER_SIGNEDNESS)

Warn when pointers passed via arguments or assigned to a variable differ in sign.

Mismatched Return Type (GCC_WARN_ABOUT_RETURN_TYPE)

Causes warnings to be emitted when a function with a defined return type (not `void`) contains a return statement without a return-value or when it does not contain any return statements. Also emits a warning when a function with a void return type tries to return a value.

Incomplete Objective-C Protocols (GCC_WARN_ALLOW_INCOMPLETE_PROTOCOL)

Warn if methods required by a protocol are not implemented in the class adopting it. Only applies to Objective-C.

Check Switch Statements (GCC_WARN_CHECK_SWITCH_STATEMENTS)

Warn whenever a switch statement has an index of enumeral type and lacks a case for one or more of the named codes of that enumeration. The presence of a default label prevents this warning. Case labels outside the enumeration range also provoke warnings when this option is used.

Four Character Literals (GCC_WARN_FOUR_CHARACTER_CONSTANTS)

Warn about four-char literals (for example, macOS-style `OSTypes: 'APPL'`).

Overloaded Virtual Functions (GCC_WARN_HIDDEN_VIRTUAL_FUNCTIONS)

Warn when a function declaration hides virtual functions from a base class.

For example, in the following example, the `A` class version of `f()` is hidden in `B`.

```
struct A {
  virtual void f();
};
struct B: public A {
  void f(int);
};
```

As a result, the following code will fail to compile.

```
B* b;
b-&gt;f();
```

This setting only applies to C++ and Objective-C++ sources.

Inhibit All Warnings (GCC_WARN_INHIBIT_ALL_WARNINGS)

Inhibit all warning messages.

Initializer Not Fully Bracketed (GCC_WARN_INITIALIZER_NOT_FULLY_BRACKETED)

Warn if an aggregate or union initializer is not fully bracketed. In the following example, the initializer for `a` is not fully bracketed, but the initializer for `b` is fully bracketed.

```
int a`2``2` = { 0, 1, 2, 3 };
int b`2``2` = { { 0, 1 }, { 2, 3 } };
```

Missing Braces and Parentheses (GCC_WARN_MISSING_PARENTHESES)

Warn if parentheses are omitted in certain contexts, such as when there is an assignment in a context where a truth value is expected, or when operators are nested whose precedence causes confusion. Also, warn about constructions where there may be confusion as to which `if` statement an `else` branch belongs. For example:

```
{
  if (a)
    if (b)
      foo ();
  else
    bar ();
}
```

In C, every `else` branch belongs to the innermost possible `if` statement, which in the example above is `if (b)`. This is often not what the programmer expects, as illustrated by indentation used in the example above. This build setting causes GCC to issue a warning when there is the potential for this confusion. To eliminate the warning, add explicit braces around the innermost `if` statement so there is no way the `else` could belong to the enclosing `if`. For example:

```
{
  if (a)
    {
      if (b)
        foo ();
      else
        bar ();
    }
}
```

Nonvirtual Destructor (GCC_WARN_NON_VIRTUAL_DESTRUCTOR)

Warn when a class declares an nonvirtual destructor that should probably be virtual, because it looks like the class will be used polymorphically. This is only active for C++ or Objective-C++ sources.

Pedantic Warnings (GCC_WARN_PEDANTIC)

Issue all the warnings demanded by strict ISO C and ISO C++; reject all programs that use forbidden extensions, and some other programs that do not follow ISO C and ISO C++. For ISO C, follows the version of the ISO C standard specified by any `-std` option used.

Hidden Local Variables (GCC_WARN_SHADOW)

Warn whenever a local variable shadows another local variable, parameter or global variable or whenever a builtin function is shadowed.

Sign Comparison (GCC_WARN_SIGN_COMPARE)

Warn when a comparison between signed and unsigned values could produce an incorrect result when the signed value is converted to unsigned.

Strict Selector Matching (GCC_WARN_STRICT_SELECTOR_MATCH)

Warn if multiple methods with differing argument and/or return types are found for a given selector when attempting to send a message using this selector to a receiver of type `id` or `Class`. When this setting is disabled, the compiler will omit such warnings if any differences found are confined to types that share the same size and alignment.

Typecheck Calls to printf/scanf (GCC_WARN_TYPECHECK_CALLS_TO_PRINTF)

Check calls to `printf` and `scanf` to make sure that the arguments supplied have types appropriate to the format string specified, and that the conversions specified in the format string make sense.

Undeclared Selector (GCC_WARN_UNDECLARED_SELECTOR)

Warn if a `@selector(...)` expression referring to an undeclared selector is found. A selector is considered undeclared if no method with that name has been declared before the `@selector(...)` expression, either explicitly in an `@interface` or `@protocol` declaration, or implicitly in an `@implementation` section. This option always performs its checks as soon as a `@selector(...)` expression is found, while `–Wselector` only performs its checks in the final stage of compilation. This also enforces the coding style convention that methods and selectors must be declared before being used.

Uninitialized Variables (GCC_WARN_UNINITIALIZED_AUTOS)

Warn if a variable might be clobbered by a `setjmp` call or if an automatic variable is used without prior initialization.

The compiler may not detect all cases where an automatic variable is initialized or all usage patterns that may lead to use prior to initialization. You can toggle between the normal uninitialized value checking or the more aggressive (conservative) checking, which finds more issues but the checking is much stricter.

Unknown Pragma (GCC_WARN_UNKNOWN_PRAGMAS)

Warn when a `#pragma` directive is encountered that is not understood by GCC. If this command line option is used, warnings will even be issued for unknown pragmas in system header files. This is not the case if the warnings were only enabled by the `–Wall` command-line option.

Unused Functions (GCC_WARN_UNUSED_FUNCTION)

Warn whenever a static function is declared but not defined or a noninline static function is unused.

Unused Labels (GCC_WARN_UNUSED_LABEL)

Warn whenever a label is declared but not used.

Unused Parameters (GCC_WARN_UNUSED_PARAMETER)

Warn whenever a function parameter is unused aside from its declaration.

Unused Values (GCC_WARN_UNUSED_VALUE)

Warn whenever a statement computes a result that is explicitly not used.

Unused Variables (GCC_WARN_UNUSED_VARIABLE)

Warn whenever a local variable or nonconstant static variable is unused aside from its declaration.

Perform Single-Object Prelink (GENERATE_MASTER_OBJECT_FILE)

Activating this setting will cause the object files built by a target to be prelinked using `ld -r` into a single object file, and that object file will then be linked into the final product. This is useful to force the linker to resolve symbols and link the object files into a single module before building a static library. Also, a separate set of link flags can be applied to the prelink allowing additional control over, for instance, exported symbols.

Force Package Info Generation (GENERATE_PKGINFO_FILE)

Forces the `PkgInfo` file to be written to wrapped products even if this file is not expected.

Generate Profiling Code (GENERATE_PROFILING_CODE)

Activating this setting will cause the compiler and linker to generate profiling code. For example, GCC will generate code suitable for use with `gprof(1)`.

Enable Text-Based Stubs Generation (GENERATE_TEXT_BASED_STUBS)

Enables the generation of Text-Based stubs for dynamic libraries and frameworks.

HEADERMAP_INCLUDES_FLAT_ENTRIES_FOR_TARGET_BEING_BUILT

Specifies whether the header map contains a name/path entry for every header in the target being built.

HEADERMAP_INCLUDES_FRAMEWORK_ENTRIES_FOR_ALL_PRODUCT_TYPES

Specifies whether the header map contains a framework-name/path entry for every header in the target being built, including targets that do not build frameworks.

HEADERMAP_INCLUDES_PROJECT_HEADERS

Specifies whether the header map contains a name/path entry for every header in the project, regardless of the headers' target membership.

Header Search Paths (HEADER_SEARCH_PATHS)

This is a list of paths to folders to be searched by the compiler for included or imported header files when compiling C, Objective-C, C++, or Objective-C++. Paths are delimited by whitespace, so any paths with spaces in them need to be properly quoted.

Auto-Activate Custom Fonts (IBC_COMPILER_AUTO_ACTIVATE_CUSTOM_FONTS)

Instructs the XIB compiler to add custom fonts to the application's `Info.plist`, which will cause the fonts to activate upon application launch.

Show Errors (IBC_ERRORS)

Show errors encountered during the compilation of XIB files.

Flatten Compiled XIB Files (IBC_FLATTEN_NIBS)

If enabled, compile XIB files into flattened (non-wrapper) NIB files. After flattening, the resulting NIB is more compact but no longer editable by Interface Builder. When this option is disabled, the resulting NIB file remains editable in Interface Builder.

Default Module (IBC_MODULE)

Defines the module name for Swift classes referenced without a specific module name.

Show Notices (IBC_NOTICES)

Show notices encountered during the compilation of XIB files.

Other Interface Builder Compiler Flags (IBC_OTHER_FLAGS)

A list of additional flags to pass to the Interface Builder Compiler. Use this setting if Xcode does not already provide UI for a particular Interface Builder Compiler flag.

Overriding Plug-In and Framework Directory (IBC_OVERRIDING_PLUGINS_AND_FRAMEWORKS_DIR)

Instructs Interface Builder to load frameworks and Interface Builder plugins from the specified directory. Setting this value to `$(BUILD_DIR)/$(CONFIGURATION)$(EFFECTIVE_PLATFORM_NAME)` will ensure that Interface Builder will load frameworks and plug-ins from the built products directory of the current build configuration.

Plug-Ins (IBC_PLUGINS)

A list of paths to Interface Builder plugins to load when compiling XIB files.

Plug-In Search Paths (IBC_PLUGIN_SEARCH_PATHS)

A list of paths to be searched for Interface Builder plug-ins to load when compiling XIB files.

Strip NIB Files (IBC_STRIP_NIBS)

Strips an Interface Builder NIB to reduce its size for deployment. The resulting NIB is more compact but no longer editable by Interface Builder. When this option is disabled, the resulting NIB file remains editable by Interface Builder.

Show Warnings (IBC_WARNINGS)

Show warnings encountered during the compilation of XIB files.

Auto-Activate Custom Fonts (IBSC_COMPILER_AUTO_ACTIVATE_CUSTOM_FONTS)

Instructs the Storyboard compiler to add custom fonts to the application's `Info.plist` that will cause the fonts to activate upon application launch.

Show Errors (IBSC_ERRORS)

Show errors encountered during the compilation of Storyboard files.

Flatten Compiled Storyboard Files (IBSC_FLATTEN_NIBS)

Compiles a Storyboard file into flattened (non-wrapper) Storyboard file. After flattening, the resulting Storyboard is more compact but no longer editable by Interface Builder. When this option is disabled, the resulting Storyboard file remains editable in Interface Builder.

Default Module (IBSC_MODULE)

Defines the module name for Swift classes referenced without a specific module name.

Show Notices (IBSC_NOTICES)

Show notices encountered during the compilation of Storyboard files.

Other Storyboard Compiler Flags (IBSC_OTHER_FLAGS)

A list of additional flags to pass to the Interface Builder Compiler. Use this setting if Xcode does not already provide UI for a particular Interface Builder Compiler flag.

Strip Storyboardc Files (IBSC_STRIP_NIBS)

Strips an editable Interface Builder storyboardc file to reduce its size for deployment. The resulting storyboardc is more compact but no longer editable by Interface Builder. When this option is disabled, the resulting storyboardc file remains editable by Interface Builder.

Show Warnings (IBSC_WARNINGS)

Show warnings encountered during the compilation of Storyboard files.

Sub-Directories to Include in Recursive Searches (INCLUDED_RECURSIVE_SEARCH_PATH_SUBDIRECTORIES)

This is a list of `fnmatch()`-style patterns of file or directory names to include when performing a recursive search. By default, this is empty and is only customized when you want to provide exceptions to the list of filename patterns provided in Sub-Directories to Exclude in Recursive Searches (EXCLUDED_RECURSIVE_SEARCH_PATH_SUBDIRECTORIES).

Included Source File Names (INCLUDED_SOURCE_FILE_NAMES)

A list of patterns (as defined by `fnmatch(3)`) specifying the names of source files to explicitly *include* when processing the files in the target's build phases. This setting is only useful when combined with Excluded Source File Names (EXCLUDED_SOURCE_FILE_NAMES), and can be used to define complex filters for which files from the phase should be built in response to other build settings.

Expand Build Settings in Info.plist File (INFOPLIST_EXPAND_BUILD_SETTINGS)

Expand build settings in the `Info.plist` file.

Info.plist File (INFOPLIST_FILE)

The project-relative path to the property list file that contains the `Info.plist` information used by bundles. For details on information property list files, see Information Property List Files in Runtime Configuration Guidelines.

Info.plist Other Preprocessor Flags (INFOPLIST_OTHER_PREPROCESSOR_FLAGS)

Other flags to pass to the C preprocessor when preprocessing the `Info.plist` file.

Info.plist Output Encoding (INFOPLIST_OUTPUT_FORMAT)

Specifies the output encoding for the output `Info.plist`. The output encodings can be `binary` or `XML`. By default, the output encoding will be unchanged from the input.

INFOPLIST_PATH

Specifies the path to the bundle's information property list file.

Info.plist Preprocessor Prefix File (INFOPLIST_PREFIX_HEADER)

Implicitly include the given file when preprocessing the `Info.plist` file. The path given should either be a project relative path or an absolute path.

Preprocess Info.plist File (INFOPLIST_PREPROCESS)

Preprocess the `Info.plist` file using the C Preprocessor.

Info.plist Preprocessor Definitions (INFOPLIST_PREPROCESSOR_DEFINITIONS)

Space-separated list of preprocessor macros of the form `foo` or `foo=bar`. These macros are used when preprocessing the `Info.plist` file.

INFOSTRINGS_PATH

Specifies the file that contains the bundle's localized strings file.

Initialization Routine (INIT_ROUTINE)

This is the name of the routine to use for initialization.

Enable Text-Based Stubs Inlining (INLINE_PRIVATE_FRAMEWORKS)

Enables private framework inlining for Text-Based Stubs.

Perform Copy Files Phases During `installhdrs` (INSTALLHDRS_COPY_PHASE)

Specifies whether the target's Copy Files build phases are executed in `installhdr` builds.

Perform Shell Script Phases During `installhdrs` (INSTALLHDRS_SCRIPT_PHASE)

Specifies whether the target's Run Script build phases are executed in `installhdr` builds. See Active Build Action (ACTION) for details on `installhdr` builds.

INSTALL_DIR

Identifies the directory in the developer's filesystem into which the *installed* product is placed.

Install Group (INSTALL_GROUP)

The group name or `gid` for installed products.

Install Permissions (INSTALL_MODE_FLAG)

Permissions used for installed product files.

Install Owner (INSTALL_OWNER)

The owner name or `uid` for installed products.

Installation Directory (INSTALL_PATH)

The directory in which to install the build products. This path is prepended by the Installation Build Products Location (DSTROOT).

Intent Class Generation Language (INTENTS_CODEGEN_LANGUAGE)

The Source-code language to use for generated Intent class. By default "Automatic" will analyze your project to determine the correct language. Adjust this setting to explicitly select "Swift" or "Objective-C".

Building for Mac Catalyst (IS_MACCATALYST)

Indicates whether the target is building for Mac Catalyst. This build setting is intended for use in shell scripts and build setting composition and should be considered read-only.

Preserve Private External Symbols (KEEP_PRIVATE_EXTERNS)

Activating this setting will preserve private external symbols, rather than turning them into static symbols. This setting is also respected when performing a single-object prelink.

Path to Linker Dependency Info File (LD_DEPENDENCY_INFO_FILE)

This setting defines the path to which the linker should emit information about what files it used as inputs and generated. Xcode uses this information for its dependency tracking. Setting the value of this setting to empty will disable passing this option to the linker.

Dynamic Library Allowable Clients (LD_DYLIB_ALLOWABLE_CLIENTS)

This setting restricts the clients allowed to link a dylib by passing `–allowable_client` to the linker for each supplied value.

Dynamic Library Install Name (LD_DYLIB_INSTALL_NAME)

Sets an internal `install path` (`LC_ID_DYLIB`) in a dynamic library. Any clients linked against the library will record that path as the way `dyld` should locate this library. If this option is not specified, then the `–o` path will be used. This setting is ignored when building any product other than a dynamic library. See Dynamic Library Programming Topics.

Write Link Map File (LD_GENERATE_MAP_FILE)

Activating this setting will cause the linker to write a map file to disk, which details all symbols and their addresses in the output image. The path to the map file is defined by the Path to Link Map File (LD_MAP_FILE_PATH) setting.

Path to Link Map File (LD_MAP_FILE_PATH)

This setting defines the path to the map file written by the linker when the Write Link Map File (LD_GENERATE_MAP_FILE) setting is activated. By default, a separate file will be written for each architecture and build variant, and these will be generated in the Intermediates directory for the target whose product is

being linked.

Generate Position-Dependent Executable (LD_NO_PIE)

Activating this setting will prevent Xcode from building a main executable that is position independent (PIE). When targeting macOS 10.7 or later, PIE is the default for main executables, so activating this setting will change that behavior. When targeting OS X 10.6 or earlier, or when building for i386, PIE is not the default, so activating this setting does nothing.

You cannot create a PIE from `.o` files compiled with `-mdynamic-no-pic`. Using PIE means the codegen is less optimal, but the address randomization adds some security.

Quote Linker Arguments (LD_QUOTE_LINKER_ARGUMENTS_FOR_COMPILER_DRIVER)

This setting controls whether arguments to the linker should be quoted using `-Xlinker`. By default, Xcode invokes the linker by invoking the driver of the compiler used to build the source files in the target, and passing `-Xlinker` to quote arguments will cause the compiler driver to pass them through to the linker (rather than trying to evaluate them within the driver). By default, this setting is enabled. Disabling it will cause Xcode to not use `-Xlinker` to pass arguments to the linker. Disabling this setting is useful if the target has instructed Xcode to use an alternate linker (for example, by setting the `LD` setting to the path to another linker) and that alternate linker does not recognize `-Xlinker`.

Runpath Search Paths (LD_RUNPATH_SEARCH_PATHS)

This is a list of paths to be added to the `runpath` search path list for the image being created. At runtime, `dyld` uses the `runpath` when searching for dylibs whose load path begins with `@rpath/`. See Dynamic Library Programming Topics.

Other Lex Flags (LEXFLAGS)

Space-separated list of additional flags to pass to `lex`. Be sure to backslash-escape any arguments that contain spaces or special characters, such as path names that may contain spaces. Use this setting if Xcode does not already provide UI for a `lex` flag.

Generate Case-Insensitive Scanner (LEX_CASE_INSENSITIVE_SCANNER)

Enabling this option causes `lex` to generate a case-insensitive scanner. The case of letters given in the `lex` input patterns will be ignored, and tokens in the input will be matched regardless of case. The matched text given in `yytext` will have the preserved case (for example, it will not be folded).

Insert #line Directives (LEX_INSERT_LINE_DIRECTIVES)

Enabling this option instructs `lex` to insert `#line` directives so error messages in the actions will be correctly located with respect to either the original `lex` input file (if the errors are due to code in the input file), or `lex.yy.c` (if the errors are `lex`'s fault). This option is enabled by default; disabling it passes a flag to `lex` to not insert `#line` directives.

Suppress Default Rule (LEX_SUPPRESS_DEFAULT_RULE)

Enabling this option causes the default rule (that unmatched scanner input is echoed to `stdout`) to be suppressed. If the scanner encounters input that does not match any of its rules, it aborts with an error. This option is useful for finding holes in a scanner's rule set.

Suppress Warning Messages (LEX_SUPPRESS_WARNINGS)

Enabling this option causes `lex` to suppress its warning messages.

Library Search Paths (LIBRARY_SEARCH_PATHS)

This is a list of paths to folders to be searched by the linker for libraries used by the product. Paths are delimited by whitespace, so any paths with spaces in them need to be properly quoted.

Display Mangled Names (LINKER_DISPLAYS_MANGLED_NAMES)

Activating this setting causes the linker to display mangled names for C++ symbols. Normally, this is not recommended, but turning it on can help to diagnose and solve C++ link errors.

Link With Standard Libraries (LINK_WITH_STANDARD_LIBRARIES)

When this setting is enabled, the compiler driver will automatically pass its standard libraries to the linker to use during linking. If desired, this flag can be used to disable linking with the standard libraries, and then individual libraries can be passed as Other Linker Flags (OTHER_LDFLAGS).

Link-Time Optimization (LLVM_LTO)

Enabling this setting allows optimization across file boundaries during linking.

- *No:* Disabled. Do not use link-time optimization.
- *Monolithic Link-Time Optimization:* This mode performs monolithic link-time optimization of binaries, combining all executable code into a single unit and running aggressive compiler optimizations.
- *Incremental Link-Time Optimization:* This mode performs partitioned link-time optimization of binaries, inlining between compilation units and running aggressive compiler optimizations on each unit in parallel. This enables fast incremental builds and uses less memory than Monolithic LTO.

Localized String Macro Names (LOCALIZED_STRING_MACRO_NAMES)

The base names for NSLocalizedString-like macros or functions used to produce localized strings in source code. The default base names of NSLocalizedString and CFCopyLocalizedString are always considered, even if this setting is empty.

Localized String Swift UI Support (LOCALIZED_STRING_SWIFTUI_SUPPORT)

When enabled, literal strings passed to the Text() initializer from Swift UI will be extracted during localization export.

Mach-O Type (MACH_O_TYPE)

This setting determines the format of the produced binary and how it can be linked when building other binaries. For information on binary types, see Building Mach-O Files in Mach-O Programming Topics.

- *Executable:* Executables and standalone binaries and cannot be linked. `mh_execute`
- *Dynamic Library:* Dynamic libraries are linked at build time and loaded automatically when needed. `mh_dylib`
- *Bundle:* Bundle libraries are loaded explicitly at run time. `mh_bundle`
- *Static Library:* Static libraries are linked at build time and loaded at execution time. `staticlib`
- *Relocatable Object File:* Object files are single-module files that are linked at build time. `mh_object`

Suppress all mapc warnings (MAPC_NO_WARNINGS)

Compile `.xcmappingmodel` files into `.cdm` without reporting warnings.

Marketing Version (MARKETING_VERSION)

This setting defines the user-visible version of the project. The value corresponds to the `CFBundleShortVersionString` key in your app's Info.plist.

Module Map File (MODULEMAP_FILE)

This is the project-relative path to the LLVM module map file that defines the module structure for the compiler. If empty, it will be automatically generated for appropriate products when Defines Module (DEFINES_MODULE) is enabled.

Private Module Map File (MODULEMAP_PRIVATE_FILE)

This is the project-relative path to the LLVM module map file that defines the module structure for private headers.

MODULES_FOLDER_PATH

Specifies the directory that contains the product's Clang module maps and Swift module content.

MODULE_CACHE_DIR

Absolute path of folder in which compiler stores its cached modules—this cache is a performance improvement.

Module Identifier (MODULE_NAME)

This is the identifier of the kernel module listed in the generated stub. This is only used when building kernel extensions.

Module Start Routine (MODULE_START)

This defines the name of the kernel module start routine. This is only used when building kernel extensions.

Module Stop Routine (MODULE_STOP)

This defines the name of the kernel module stop routine. This is only used when building kernel extensions.

Module Version (MODULE_VERSION)

This is the version of the kernel module listed in the generated stub. This is only used when building kernel extensions.

Suppress momc warnings for delete rules (MOMC_NO_DELETE_RULE_WARNINGS)

Suppress managed object model compiler (`momc`) warnings for delete rules during the compilation of `.xcdatamodel(d)` files.

Suppress momc warnings on missing inverse relationships (MOMC_NO_INVERSE_RELATIONSHIP_WARNINGS)

Suppress managed object model compiler (`momc`) warnings from output on missing inverse relationships during the compilation of `.xcdatamodel(d)` files

Suppress momc warnings for entities with more than 100 properties (MOMC_NO_MAX_PROPERTY_COUNT_WARNINGS)

Suppress managed object model compiler (`momc`) warnings from output on entities with more than 100 properties during the compilation of `.xcdatamodel(d)` files.

Suppress all momc warnings (MOMC_NO_WARNINGS)

Suppress managed object model compiler (`momc`) warnings from output during the compilation of `.xcdatamodel(d)` files

Suppress momc error on transient inverse relationships (MOMC_SUPPRESS_INVERSE_TRANSIENT_ERROR)

Suppress managed object model compiler (`momc`) warnings from output on transient inverse relationships during the compilation of `.xcdatamodel(d)` files. This is only intended to be used on 10.4.x created models that compiled properly in 10.4.x before the error was introduced in 10.5

Other Metal Linker Flags (MTLLINKER_FLAGS)

Space-separated list of metal linker flags

Other Metal Compiler Flags (MTL_COMPILER_FLAGS)

Space-separated list of compiler flags

Produce Debugging Information (MTL_ENABLE_DEBUG_INFO)

Debugging information is required for shader debugging and profiling.

Enable Index-While-Building Functionality (Metal) (MTL_ENABLE_INDEX_STORE)

Control whether the compiler should emit index data while building.

Enable Modules (Metal) (MTL_ENABLE_MODULES)

Enables the use of modules. System headers are imported as semantic modules instead of raw headers. This can result in faster builds and project indexing.

Enable Fast Math (MTL_FAST_MATH)

Enable optimizations for floating-point arithmetic that may violate the IEEE 754 standard and disable the high precision variant of math functions for single and half precision floating-point.

Header Search Paths (MTL_HEADER_SEARCH_PATHS)

This is a list of paths to folders to be searched by the compiler for included or imported header files when compiling Metal. Paths are delimited by whitespace, so any paths with spaces in them need to be properly quoted. `MTL_HEADER_SEARCH_PATHS, -I`

Ignore Warnings (MTL_IGNORE_WARNINGS)

Enabling this option causes all warnings to be ignored. `MTL_IGNORE_WARNINGS, -W`

Metal Language Revision (MTL_LANGUAGE_REVISION)

Determine the language revision to use. A value for this option must be provided.

Preprocessor Definitions (MTL_PREPROCESSOR_DEFINITIONS)

Space-separated list of preprocessor macros of the form "foo" or "foo=bar".

Treat Warnings as Errors (MTL_TREAT_WARNINGS_AS_ERRORS)

Enabling this option causes all warnings to be treated as errors. `MTL_TREAT_WARNINGS_AS_ERRORS, -Werror`

Other Nasm Flags (NASM_OTHER_FLAGS)

Space-separated list of additional flags to pass to the `nasm` assembler. Be sure to backslash-escape any arguments that contain spaces or special characters, such as path names that may contain spaces. Use this setting if Xcode does not already provide UI for a particular `nasm` flag.

Nasm Preinclude File (NASM_PREINCLUDE_FILE)

Specifies a file to be preincluded, before the main source file starts to be processed.

NATIVE_ARCH

Identifies the architecture on which the build is being performed.

OBJECT_FILE_DIR

Partially identifies the directory into which variant object files are placed. The complete specification is computed using the variants of this build setting.

Intermediate Build Files Path (OBJROOT)

The path where intermediate files will be placed during a build. Intermediate files include generated sources, object files, etc. Shell script build phases can place and access files here, as well. Typically this path is not set per target, but is set per project or per user. By default, this is set to `$(PROJECT_DIR)/build`.

Build Active Architecture Only (ONLY_ACTIVE_ARCH)

If enabled, only the active architecture is built. This setting will be ignored when building with a run destination which does not define a specific architecture, such as a 'Generic Device' run destination.

On Demand Resources Initial Install Tags (ON_DEMAND_RESOURCES_INITIAL_INSTALL_TAGS)

Defined a set of initial On Demand Resources tags to be downloaded and installed with your application.

On Demand Resources Prefetch Order (ON_DEMAND_RESOURCES_PREFETCH_ORDER)

Once your app is installed, this defined a set of On Demand Resources tags that should be downloaded. These tags are downloaded after the initial installation of your application, and will be downloaded in the order the tags provided in the list from first to last.

OpenCL Architectures (OPENCL_ARCHS)

A list of the architectures for which the product will be built. This is usually set to a predefined build setting provided by the platform.

Auto-vectorizer (OPENCL_AUTO_VECTORIZE_ENABLE)

Auto-vectorizes the `OpenCL` kernels for the CPU. This setting takes effect only for the CPU. This makes it possible to write a single kernel that is portable and performant across CPUs and GPUs.

OpenCL Compiler Version (OPENCL_COMPILER_VERSION)

The `OpenCL` C compiler version supported by the platform.

Flush denorms to zero (OPENCL_DENORMS_ARE_ZERO)

This option controls how single precision and double precision denormalized numbers are handled. If specified as a build option, the single precision denormalized numbers may be flushed to zero; double precision denormalized numbers may also be flushed to zero if the optional extension for double precision is supported. This is intended to be a performance hint and the `OpenCL` compiler can choose not to flush denorms to zero if the device supports single precision (or double precision) denormalized numbers.

This option is ignored for single precision numbers if the device does not support single precision denormalized numbers, for example, `CL_FP_DENORM` bit is not set in `CL_DEVICE_SINGLE_FP_CONFIG`.

This option is ignored for double precision numbers if the device does not support double precision or if it does support double precision but not double precision denormalized numbers, for example, `CL_FP_DENORM` bit is not set in `CL_DEVICE_DOUBLE_FP_CONFIG`.

This flag only applies for scalar and vector single precision floating-point variables and computations on these floating-point variables inside a program. It does not apply to reading from or writing to image objects.

Double as single (OPENCL_DOUBLE_AS_SINGLE)

Treat double precision floating-point expression as a single precision floating-point expression. This option is available for GPUs only.

Relax IEEE Compliance (OPENCL_FAST_RELAXED_MATH)

This allows optimizations for floating-point arithmetic that may violate the IEEE 754 standard and the `OpenCL` numerical compliance requirements defined in in section 7.4 for single-precision floating-point, section 9.3.9 for double-precision floating-point, and edge case behavior in section 7.5 of the `OpenCL` 1.1 specification.

This is intended to be a performance optimization.

This option causes the preprocessor macro `__FAST_RELAXED_MATH__` to be defined in the `OpenCL` program.

Use MAD (OPENCL_MAD_ENABLE)

Allow `a * b + c` to be replaced by a `mad` instruction. The `mad` computes `a * b + c` with reduced accuracy. For example, some `OpenCL` devices implement `mad` as truncate the result of `a * b` before adding it to `c`.

This is intended to be a performance optimization.

Optimization Level (OPENCL_OPTIMIZATION_LEVEL)

- *None:* Do not optimize. `-O0` With this setting, the compiler's goal is to reduce the cost of compilation and to make debugging produce the expected results. Statements are independent: if you stop the program with a breakpoint between statements, you can then assign a new value to any variable or change the program counter to any other statement in the function and get exactly the results you would expect from the source code.
- *Fast:* Optimizing compilation takes somewhat more time, and a lot more memory for a large function. `-O, -O1` With this setting, the compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time. In Apple's compiler, strict aliasing, block reordering, and inter-block scheduling are disabled by default when optimizing.
- *Faster:* The compiler performs nearly all supported optimizations that do not involve a space-speed tradeoff. `-O2` With this setting, the compiler does not perform loop unrolling or function inlining, or register renaming. As compared to the `Fast` setting, this setting increases both compilation time and the performance of the generated code.
- *Fastest:* Turns on all optimizations specified by the `Faster` setting and also turns on function inlining and register renaming options. This setting may result in a larger binary. `-O3`
- *Fastest, smallest:* Optimize for size. This setting enables all `Faster` optimizations that do not typically increase code size. It also performs further optimizations designed to reduce code size. `-Os`

OpenCL Other Flags (OPENCL_OTHER_BC_FLAGS)

Space-separated list of additional flags to pass to the compiler. Be sure to backslash-escape any arguments that contain spaces or special characters, such as path names that may contain spaces. Use this setting if Xcode does not already provide UI for a particular compiler flag.

OpenCL Preprocessor Macros (OPENCL_PREPROCESSOR_DEFINITIONS)

Space-separated list of preprocessor macros of the form `foo` or `foo=bar`.

Order File (ORDER_FILE)

The path to a file that alters the order in which functions and data are laid out.

For each section in the output file, any symbol in that section that are specified in the order file is moved to the start of its section and laid out in the same order as in the order file. Order files are text files with one symbol name per line. Lines starting with a # are comments. A symbol name may be optionally preceded with its object file leafname and a colon (for example, `foo.o:_foo`). This is useful for static functions/data that occur in multiple files. A symbol name may also be optionally preceded with the architecture (for example, `ppc:_foo` or `ppc:foo.o:_foo`). This enables you to have one order file that works for multiple architectures. Literal C-strings may be ordered by quoting the string in the order file (for example, `"Hello, world\n"`).

Generally you should not specify an order file in Debug or Development configurations, as this will make the linked binary less readable to the debugger. Use them only in Release or Deployment configurations.

Save as Execute-Only (OSACOMPILE_EXECUTE_ONLY)

Saves the output script in execute-only form; the script can be run, but cannot be opened in Script Editor or Xcode. With this option turned off, a user may see the original script source by opening the script.

Other C Flags (OTHER_CFLAGS)

Space-separated list of additional flags to pass to the compiler for C and Objective-C files. Be sure to backslash-escape any arguments that contain spaces or special characters, such as path names that may contain spaces. Use this setting if Xcode does not already provide UI for a particular C or Objective-C compiler flag.

Other Code Signing Flags (OTHER_CODE_SIGN_FLAGS)

A list of additional options to pass to `codesign(1)`.

Other C++ Flags (OTHER_CPLUSPLUSFLAGS)

Space-separated list of additional flags to pass to the compiler for C++ and Objective-C++ files. Be sure to backslash-escape any arguments that contain spaces or special characters, such as path names that may contain spaces. Use this setting if Xcode does not already provide UI for a C++ or Objective-C++ compiler flag.

Other IIG C Flags (OTHER_IIG_CFLAGS)

Space-separated list of additional flags to pass to the `iig` invocation of clang. Be sure to backslash-escape any arguments that contain spaces or special characters, such as path names that may contain spaces. Use this setting if Xcode does not already provide UI for a particular `iig` flag

Other IIG Flags (OTHER_IIG_FLAGS)

Space-separated list of additional flags to pass to the `iig` compiler. Be sure to backslash-escape any arguments that contain spaces or special characters, such as path names that may contain spaces. Use this setting if Xcode does not already provide UI for a particular `iig` flag

Other Linker Flags (OTHER_LDFLAGS)

Options defined in this setting are passed to invocations of the linker.

Other Librarian Flags (OTHER_LIBTOOLFLAGS)

Options defined in this setting are passed to all invocations of the archive librarian, which is used to generate static libraries.

Other MiG Flags (OTHER_MIGFLAGS)

Space-separated list of additional flags to pass to `mig`. Be sure to backslash-escape any arguments that contain spaces or special characters, such as path names that may contain spaces. Use this setting if Xcode does not already provide UI for a `mig` flag.

Other OSACompile Flags (OTHER_OSACOMPILEFLAGS)

Space-separated list of additional flags to pass to `osacompile`. Be sure to backslash-escape any arguments that contain spaces or special characters, such as path names that may contain spaces. Use this setting if Xcode does not already provide UI for a particular `osacompile` flag.

Other Rez Flags (OTHER_REZFLAGS)

Space-separated list of additional flags to pass to the `Rez` compiler. Be sure to backslash-escape any arguments that contain spaces or special characters, such as path names that may contain spaces. Use this setting if Xcode does not already provide UI for a particular `Rez` flag.

Other Swift Flags (OTHER_SWIFT_FLAGS)

A list of additional flags to pass to the Swift compiler.

Other Text-Based InstallAPI Flags (OTHER_TAPI_FLAGS)

Options defined in this setting are passed to invocations of the `Text-Based InstallAPI` tool.

PACKAGE_TYPE

Uniform type identifier. Identifies the type of the product the target builds. Some products may be made up of a single binary or archive. Others may comprise several files, which are grouped under a single directory. These container directories are known as *bundles*.

Property List Output Encoding (PLIST_FILE_OUTPUT_FORMAT)

Specifies the output encoding for property list files (`.plist`). The output encodings can be `binary` or `XML`. By default, the output encoding will be unchanged from the input.

PLUGINS_FOLDER_PATH

Specifies the directory that contains the product's plugins.

Precompiled Header Uses Files From Build Directory (PRECOMPS_INCLUDE_HEADERS_FROM_BUILT_PRODUCTS_DIR)

This setting allows for better control of sharing precompiled prefix header files between projects. By default, Xcode assumes that the prefix header file may include header files from the build directory if the build directory is outside of the project directory. Xcode cannot determine this ahead of time since other projects may not have been built into the shared build directory at the time the information is needed.

If your prefix file never includes files from the build directory you may set this to `NO` to improve sharing of precompiled headers. If the prefix does use files from a build directory that is inside your project directory, you may set this to `YES` to avoid unintended sharing that may result in build failures.

Single-Object Prelink Flags (PRELINK_FLAGS)

Additional flags to pass when performing a single-object prelink.

Prelink libraries (PRELINK_LIBS)

Additional libraries to pass when performing a single-object prelink.

Don't Dead-Strip Inits and Terms (PRESERVE_DEAD_CODE_INITS_AND_TERMS)

Activating this setting, in combination with the Dead Code Stripping (DEAD_CODE_STRIPPING) (`-dead_strip`) option, causes the `-no_dead_strip_inits_and_terms` flag to be passed to `ld(1)` via `cc(1)` to disable dead code stripping for initialization and termination routines. This option should not be used without the aforementioned Dead Code Stripping (DEAD_CODE_STRIPPING) option.

Private Headers Folder Path (PRIVATE_HEADERS_FOLDER_PATH)

The location to copy the private headers to during building, relative to the built products folder.

Product Bundle Identifier (PRODUCT_BUNDLE_IDENTIFIER)

A string that uniquely identifies the bundle. The string should be in reverse DNS format using only alphanumeric characters (`A–Z`, `a–z`, `0–9`), the dot (`.`), and the hyphen (`–`). This value is used as the `CFBundleIdentifier` in the `Info.plist` of the built bundle.

PRODUCT_DEFINITION_PLIST

Path to a file specifying additional requirements for a product archive.

Product Module Name (PRODUCT_MODULE_NAME)

The name to use for the source code module constructed for this target, and which will be used to import the module in implementation source files. Must be a valid identifier.

Product Name (PRODUCT_NAME)

This is the basename of the product generated by the target.

Project Name (PROJECT_NAME)

The name of the current project.

PROJECT_TEMP_DIR

Identifies the directory in which the project's intermediate build files are placed. This directory is shared between all the targets defined by the project. Run Script build phases should generate intermediate build files in the directory identified by DERIVED_FILE_DIR, not the location this build setting specifies.

Provisioning Profile (PROVISIONING_PROFILE_SPECIFIER)

Must contain a profile name (or UUID). A missing or invalid profile will cause a build error. Use in conjunction with Development Team (DEVELOPMENT_TEAM) to fully specify provisioning profile.

Public Headers Folder Path (PUBLIC_HEADERS_FOLDER_PATH)

The location to copy the public headers to during building, relative to the built products folder.

RC Generation Language (RCPROJECT_CODEGEN_LANGUAGE)

The Source-code language to use for generated RC objects. By default Xcode will generate Swift types that represent the objects in your scene. Adjust this setting to explicitly select "Swift", or select "None" to disable RC object generation.

Re-Exported Framework Names (REEXPORTED_FRAMEWORK_NAMES)

List of framework names that should have their symbols be reexported from the built library.

Re-Exported Library Names (REEXPORTED_LIBRARY_NAMES)

List of library names that should have their symbols be reexported from the built library.

Re-Exported Library Paths (REEXPORTED_LIBRARY_PATHS)

List of library paths that should have their symbols be reexported from the built library.

REMOVE_CVS_FROM_RESOURCES

Specifies whether to remove `CVS` directories from bundle resources when they are copied.

REMOVE_GIT_FROM_RESOURCES

Specifies whether to remove `.git` directories from bundle resources when they are copied.

REMOVE_HG_FROM_RESOURCES

Specifies whether to remove `.hg` directories from bundle resources when they are copied.

REMOVE_SVN_FROM_RESOURCES

Specifies whether to remove `SVN` directories from bundle resources when they are copied.

File Fork of Binary Sources (RESMERGER_SOURCES_FORK)

Determines whether `ResMerger` treats binary input files as data-fork hosted or resource-fork hosted, or whether it automatically examines each input file.

Resources Targeted Device Family (RESOURCES_TARGETED_DEVICE_FAMILY)

Overrides Targeted Device Family (TARGETED_DEVICE_FAMILY) when the resource copying needs to differ from the default targeted device.

RETAIN_RAW_BINARIES

Specifies whether to keep copies of unstripped binaries available.

REZ_COLLECTOR_DIR

Specifies the directory in which the collected Resource Manager resources generated by `ResMerger` are stored before they are added to the product.

REZ_OBJECTS_DIR

Specifies the directory in which compiled Resource Manager resources generated by `Rez` are stored before they are collected using `ResMerger`.

Rez Prefix File (REZ_PREFIX_FILE)

Implicitly include the named file on the command line for each `Rez` file compiled. The path given should either be a project relative path or an absolute path.

Preprocessor Defines (REZ_PREPROCESSOR_DEFINITIONS)

These strings will be defined when compiling resource manager resources.

Preprocessor Undefines (REZ_PREPROCESSOR_UNDEFINITIONS)

These strings will be undefined when compiling resource manager resources.

Resolve Aliases (REZ_RESOLVE_ALIASES)

Enables aliases to be unresolved or conditionally resolved. The default is to resolve aliases always.

Read-only Resource Map (REZ_RESOURCE_MAP_READ_ONLY)

Enabling this option causes the resource map output to be read-only.

Rez Script Type (REZ_SCRIPT_TYPE)

Enables the recognition of a specific 2-byte character script identifier to use when compiling resource manager resources. This allows for 2-byte characters in strings to be handled as indivisible entities. The default language is Roman, which specifies 1-byte character sets.

Rez Search Paths (REZ_SEARCH_PATHS)

This is a list of paths to search for files with resource manager resources. Paths are delimited by whitespace, so any paths with spaces in them need to be properly quoted.

Show Diagnostic Output (REZ_SHOW_DEBUG_OUTPUT)

Enabling this option causes version and progress information to be written when compiling resource manager resources.

Suppress Type Redeclaration Warnings (REZ_SUPPRESS_REDECLARED_RESOURCE_TYPE_WARNINGS)

Enabling this option causes warnings about redeclared resource types to be suppressed.

Analyze During 'Build' (RUN_CLANG_STATIC_ANALYZER)

Activating this setting will cause Xcode to run the `Clang` static analysis tool on qualifying source files during every build.

Scan All Source Files for Includes (SCAN_ALL_SOURCE_FILES_FOR_INCLUDES)

Activating this setting will cause all source files to be scanned for includes (for example, of header files) when computing the dependency graph, in which case if an included file is changed then the including file will be rebuilt next time a target containing it is built. Normally only certain types of files, such as C-language source files, are scanned.

This setting is useful if your project contains files of unusual types, which are compiled using a custom build rule.

SCRIPTS_FOLDER_PATH

Specifies the directory that contains the product's scripts.

Base SDK (SDKROOT)

The name or path of the base SDK being used during the build. The product will be built against the headers and libraries located inside the indicated SDK. This path will be prepended to all search paths, and will be passed through the environment to the compiler and linker. Additional SDKs can be specified in the Additional SDKs (ADDITIONAL_SDKS) setting.

Symbol Ordering Flags (SECTORDER_FLAGS)

These flags are typically used to specify options for ordering symbols within segments, for example the `–sectorder` option to `ld`.

Generally you should not specify symbol ordering options in Debug or Development configurations, as this will make the linked binary less readable to the debugger. Use them only in Release or Deployment configurations.

Separately Edit Symbols (SEPARATE_SYMBOL_EDIT)

Activating this setting when the linked product's symbols are to be edited will cause editing to occur via a separate invocation of `nmedit(1)`. Otherwise editing will occur during linking, if possible.

SHARED_FRAMEWORKS_FOLDER_PATH

Specifies the directory that contains the product's shared frameworks.

Precompiled Headers Cache Path (SHARED_PRECOMPS_DIR)

The path where precompiled prefix header files are placed during a build. Defaults to `$(OBJROOT)/SharedPrecompiledHeaders`. Using a common location allows precompiled headers to be shared between multiple projects.

Skip Install (SKIP_INSTALL)

If enabled, don't install built products even if deployment locations are active.

SRCROOT

Identifies the directory containing the target's source files.

Strings File Output Encoding (STRINGS_FILE_OUTPUT_ENCODING)

Specify the encoding to be used for Strings files (by default, the output encoding will be 16-bit Unicode). The value can be either an `NSStringEncoding`, such as one of the numeric values recognized by `NSString`, or it can be an IANA character set name as understood by `CFString`. The operation will fail if the file cannot be converted to the specified encoding.

Additional Strip Flags (STRIPFLAGS)

Additional flags to be passed when stripping the linked product of the build.

Strip Linked Product (STRIP_INSTALLED_PRODUCT)

If enabled, the linked product of the build will be stripped of symbols when performing deployment postprocessing.

Remove Text Metadata From PNG Files (STRIP_PNG_TEXT)

Metadata in the form of text chunks in PNG files will be removed to reduce their footprint on disk.

Strip Style (STRIP_STYLE)

The level of symbol stripping to be performed on the linked product of the build. The default value is defined by the target's product type.

- *All Symbols:* Completely strips the binary, removing the symbol table and relocation information. `all, -s`
- *Non-Global Symbols:* Strips non-global symbols, but saves external symbols. `non-global, -x`
- *Debugging Symbols:* Strips debugging symbols, but saves local and global symbols. `debugging, -S`

Strip Swift Symbols (STRIP_SWIFT_SYMBOLS)

Adjust the level of symbol stripping specified by the STRIP_STYLE setting so that when the linked product of the build is stripped, all Swift symbols will be removed.

Supported Platforms (SUPPORTED_PLATFORMS)

The list of supported platforms from which a base SDK can be used. This setting is used if the product can be built for multiple platforms using different SDKs.

Supports Mac Catalyst (SUPPORTS_MACCATALYST)

Support building this target for Mac Catalyst.

Supports Text-Based InstallAPI (SUPPORTS_TEXT_BASED_API)

Enable to indicate that the target supports `Text-Based InstallAPI`, which will enable its generation during `install` builds.

Active Compilation Conditions (SWIFT_ACTIVE_COMPILATION_CONDITIONS)

A list of compilation conditions to enable for conditional compilation expressions.

Compilation Mode (SWIFT_COMPILATION_MODE)

This setting controls the way the Swift files in a module are rebuilt.

- *Incremental*: Only rebuild the Swift source files in the module that are out of date, running multiple compiler processes as needed.
- *Whole Module*: Always rebuild all Swift source files in the module, in a single compiler process.

Disable Safety Checks (SWIFT_DISABLE_SAFETY_CHECKS)

Disable runtime safety checks when optimizing.

Exclusive Access to Memory (SWIFT_ENFORCE_EXCLUSIVE_ACCESS)

Enforce exclusive access at run-time.

Import Paths (SWIFT_INCLUDE_PATHS)

A list of paths to be searched by the Swift compiler for additional Swift modules.

Install Objective-C Compatibility Header (SWIFT_INSTALL_OBJC_HEADER)

For frameworks, install the Objective-C compatibility header describing bridged Swift classes into the Public Headers Folder Path (PUBLIC_HEADERS_FOLDER_PATH) so they may be accessed from Objective-C code using the framework. Defaults to `YES`.

Objective-C Bridging Header (SWIFT_OBJC_BRIDGING_HEADER)

Path to the header defining the Objective-C interfaces to be exposed in Swift.

Objective-C Generated Interface Header Name (SWIFT_OBJC_INTERFACE_HEADER_NAME)

Name to use for the header that is generated by the Swift compiler for use in `#import` statements in Objective-C.

Optimization Level (SWIFT_OPTIMIZATION_LEVEL)

- *None:* Compile without any optimization. `-Onone`
- *Optimize for Speed:* `-O`
- *Optimize for Size:* `-Osize`
- *Whole Module Optimization:* `-O -whole-module-optimization`

Precompile Bridging Header (SWIFT_PRECOMPILE_BRIDGING_HEADER)

Generate a precompiled header for the Objective-C bridging header, if used, in order to reduce overall build times.

Reflection Metadata Level (SWIFT_REFLECTION_METADATA_LEVEL)

This setting controls the level of reflection metadata the Swift compiler emits.

- *All:* Type information about stored properties of Swift structs and classes, Swift enum cases, and their names, are emitted into the binary for reflection and analysis in the Memory Graph Debugger.
- *Without Names:* Only type information about stored properties and cases are emitted into the binary, with their names omitted. `-disable-reflection-names`
- *None:* No reflection metadata is emitted into the binary. Accuracy of detecting memory issues involving Swift types in the Memory Graph Debugger will be degraded and reflection in Swift code may not be able to discover children of types, such as properties and enum cases. `-disable-reflection-metadata`

Suppress Warnings (SWIFT_SUPPRESS_WARNINGS)

Don't emit any warnings.

Swift 3 @objc Inference (SWIFT_SWIFT3_OBJC_INFERENCE)

Control how the Swift compiler infers @objc for declarations.

Treat Warnings as Errors (SWIFT_TREAT_WARNINGS_AS_ERRORS)

Treat all warnings as errors.

Build Products Path (SYMROOT)

The path at which all products will be placed when performing a build. Typically this path is not set per target, but is set per-project or per-user. By default, this is set to `$(PROJECT_DIR)/build`.

System Framework Search Paths (SYSTEM_FRAMEWORK_SEARCH_PATHS)

This is a list of paths to folders containing system frameworks to be searched by the compiler for both included or imported header files when compiling C, Objective-C, C++, or Objective-C++, and by the linker for frameworks used by the product. The order is from highest to lowest precedence. Paths are delimited by whitespace, so any paths with spaces in them need to be properly quoted. This setting is very similar to "Framework Search Paths", except that the search paths are passed to the compiler in a way that suppresses most warnings for headers found in system search paths. If the compiler doesn't support the concept of system framework search paths, then the search paths are appended to any existing framework search paths defined in "Framework Search Paths".

System Header Search Paths (SYSTEM_HEADER_SEARCH_PATHS)

This is a list of paths to folders to be searched by the compiler for included or imported system header files when compiling C, Objective-C, C++, or Objective-C++. The order is from highest to lowest precedence. Paths are delimited by whitespace, so any paths with spaces in them need to be properly quoted. This setting is very similar to "Header Search Paths", except that headers are passed to the compiler in a way that suppresses most warnings for headers found in system search paths. If the compiler doesn't support the concept of system header search paths, then the search paths are appended to any existing header search paths defined in "Header Search Paths".

Text-Based InstallAPI Verification Mode (TAPI_VERIFY_MODE)

Selects the level of warnings and errors to report when building `Text-Based InstallAPI`.

Targeted Device Family (TARGETED_DEVICE_FAMILY)

The build system uses the selected device to set the correct value for the `UIDeviceFamily` key it adds to the target's `Info.plist` file. This also drives the --target-device flag to actool, which determines the idioms selected during catalog compilation for iOS platforms.

TARGET_BUILD_DIR

Identifies the root of the directory hierarchy that contains the product's files (no intermediate build files). Run Script build phases that operate on product files of the target that defines them should use the value of this build setting, but Run Script build phases that operate on product files of other targets should use BUILT_PRODUCTS_DIR instead.

Target Name (TARGET_NAME)

The name of the current target.

TARGET_TEMP_DIR

Identifies the directory containing the target's intermediate build files. Run Script build phases should place intermediate files at the location indicated by DERIVED_FILE_DIR, not the directory identified by this build setting.

Test Host (TEST_HOST)

Path to the executable into which a bundle of tests is injected. Only specify this setting if testing an application or other executable.

Treat missing baselines as test failures (TREAT_MISSING_BASELINES_AS_TEST_FAILURES)

When running tests that measure performance via `XCTestCase`, report missing baselines as test failures.

Unexported Symbols File (UNEXPORTED_SYMBOLS_FILE)

A project-relative path to a file that lists the symbols not to export. See `ld -exported_symbols_list` for details on exporting symbols.

UNLOCALIZED_RESOURCES_FOLDER_PATH

Specifies the directory that contains the product's unlocalized resources.

User Header Search Paths (USER_HEADER_SEARCH_PATHS)

This is a list of paths to folders to be searched by the compiler for included or imported user header files (those headers listed in quotes) when compiling C, Objective-C, C++, or Objective-C++. Paths are delimited by whitespace, so any paths with spaces in them need to be properly quoted. See Always Search User Paths (Deprecated) (ALWAYS_SEARCH_USER_PATHS) for more details on how this setting is used. If the compiler doesn't support the concept of user headers, then the search paths are prepended to the any existing header search paths defined in Header Search Paths (HEADER_SEARCH_PATHS).

Use Header Maps (USE_HEADERMAP)

Enable the use of *Header Maps*, which provide the compiler with a mapping from textual header names to their locations, bypassing the normal compiler header search path mechanisms. This allows source code to include headers from various locations in the file system without needing to update the header search path build settings.

Validate Built Product (VALIDATE_PRODUCT)

If enabled, perform validation checks on the product as part of the build process.

Validate Workspace (VALIDATE_WORKSPACE)

If enabled, perform validation checks on the workspace configuration as part of the build process.

Validate Workspace - Ignored Frameworks (VALIDATE_WORKSPACE_SKIPPED_SDK_FRAMEWORKS)

List of framework names for which to suppress deprecation warnings and missing framework errors in the workspace validator.

Valid Architectures (VALID_ARCHS)

A space-separated list of architectures for which the target should actually be built. For each target, this is intersected with the list specified in Architectures (ARCHS), and the resulting set is built. This allows individual targets to opt out of building for particular architectures. If the resulting set of architectures is empty, no executable will be produced.

VERBOSE_PBXCP

Specifies whether the target's Copy Files build phases generate additional information when copying files.

Versioning System (VERSIONING_SYSTEM)

Selects the process used for version-stamping generated files.

- *None:* Use no versioning system.
- *Apple Generic:* Use the current project version setting. `apple-generic`

Versioning Username (VERSION_INFO_BUILDER)

This defines a reference to the user performing a build to be included in the generated Apple Generic Versioning stub. Defaults to the value of the `USER` environment variable.

Generated Versioning Variables (VERSION_INFO_EXPORT_DECL)

This defines a prefix string for the version info symbol declaration in the generated Apple Generic Versioning stub. This can be used, for example, to add an optional `export` keyword to the version symbol declaration. This should rarely be changed.

Generated Versioning Source Filename (VERSION_INFO_FILE)

Used to specify a name for the source file that will be generated by Apple Generic Versioning and compiled into your product. By default, this is set to `$(PRODUCT_NAME)_vers.c`.

Versioning Name Prefix (VERSION_INFO_PREFIX)

Used as a prefix for the name of the version info symbol in the generated versioning source file. If you prefix your exported symbols you will probably want to set this to the same prefix.

Versioning Name Suffix (VERSION_INFO_SUFFIX)

Used as a suffix for the name of the version info symbol in the generated versioning source file. This is rarely used.

Other Warning Flags (WARNING_CFLAGS)

Space-separated list of additional warning flags to pass to the compiler. Use this setting if Xcode does not already provide UI for a particular compiler warning flag.

Warning Linker Flags (WARNING_LDFLAGS)

These flags are passed with linker invocations, and by default give the `–no_arch_warnings` flag to the linker to avoid many warnings being generated during multi-architecture builds.

Wrapper Extension (WRAPPER_EXTENSION)

The extension used for product wrappers, which has a default value based on the product type.

WRAPPER_NAME

Specifies the filename, including the appropriate extension, of the product bundle.

WRAPPER_SUFFIX

Specifies the suffix of the product bundle name, including the character that separates the extension from the rest of the bundle name.

Append Plug-in Data (XCODE_PLUGINCOMPILER_APPEND)

Append compiled plug-in data to existing plug-in data, instead of overwriting it.

Copy Original Plug-in Data (XCODE_PLUGINCOMPILER_COPY_ORIGINAL)

Copy original (uncompiled) plug-in data to output alongside compiled plug-in data.

Include Extension XML (XCODE_PLUGINCOMPILER_INCLUDE_EXTENSION_XML)

Include extension XML data in the resulting xcplugindata files.

Plug-in Data Format (XCODE_PLUGINCOMPILER_OUTPUT_FORMAT)

The property list format, binary or XML, to use for the resulting xcplugindata file.

Plug-in Maximum Developer Tools Version (XCODE_PLUGINCOMPILER_TOOLS_VERSION_MAX)

Latest version of Developer Tools on which this plug-in will be used.

Plug-in Minimum Developer Tools Version (XCODE_PLUGINCOMPILER_TOOLS_VERSION_MIN)

Earliest version of Developer Tools on which this plug-in will be used.

Other Yacc Flags (YACCFLAGS)

Space-separated list of additional flags to pass to `yacc`. Be sure to backslash-escape any arguments that contain spaces or special characters, such as path names that may contain spaces. Use this setting if Xcode does not already provide UI for a `yacc` flag.

Generated File Stem (YACC_GENERATED_FILE_STEM)

The file stem to use for the files generated by `yacc`. The files will be named `<stem>.tab.c` and `<stem>.tab.h` based on the value of this setting. The Standard (y) option will cause all `yacc` source files in the same target to produce the same output file, and it is not recommended for targets containing multiple `yacc` source files.

Generate Debugging Directives (YACC_GENERATE_DEBUGGING_DIRECTIVES)

Enabling this option changes the preprocessor directives generated by `yacc` so that debugging statements will be incorporated in the compiled code.

Insert #line Directives (YACC_INSERT_LINE_DIRECTIVES)

Enabling this option causes `yacc` to insert the `#line` directives in the generated code. The `#line` directives let the C compiler relate errors in the generated code to the user's original code. If this option is disabled, `#line` directives specified by the user in the source file will still be retained.

See also

Configure build settings

Copyright © 2020 Apple Inc. All rights reserved.