Xcode Build System Guide

(Legacy)

Developer

Contents

Figures, Tables, and Listings

Objective-CSwift

Introduction

Important This document describes the build system of Xcode 3. For information about using the Xcode 4 build system, see *Project Editor Help* and *Xcode 4 User Guide*.

The *build system* is the part of the Xcode that is responsible for transforming the components of a project into one or more finished products. The build system takes a number of inputs and performs operations such as compiling, linking, copying files and so forth to produce an output—usually an application or other type software.

Xcode includes a powerful build system that can create a wide variety of Mac OS X and iOS products, such as frameworks, libraries, applications, command-line tools and more. Using Xcode's predefined project and target templates you can build these products right out of the box. However, the Xcode build system is also flexible enough to allow you to customize the build process—to tailor it to meet the special needs of your projects or support your preferred workflow.

You should read this document if you need to have a deep understanding of the Xcode build system to customize the building of your product or to use advanced features of the build system to speed up the build process.

Software requirements: The contents of this document apply to Xcode 3.2.3 on Mac OS X v10.6 (unless otherwise noted).

The following chapters show how to build a product in Xcode, describe targets and the build system inputs that they organize, and show how to take advantage of build phases, build settings, and build configurations to customize the build process. These chapters also describe a number of Xcode features that you can take advantage of to shorten the edit-build-debug cycle. These features include distributed builds, precompiled prefix headers, and predictive compilation.

Organization of This Document

This document contains the following chapters:

- Targets (page 10) describes Xcode targets and how to manage them.

- Build Phases (page 33) provides an overview of build phases, explains how Xcode determines the order in which build phases are processed to build a product.
- Build Settings (page 54) explains how build settings are implemented and how you can take advantage of them to communicate with the build system.
- Build Configurations (page 70) provides a general explanation of build configurations, describes the predefined build configurations you get when you create a target or project in Xcode, and explains how to modify and define your own build configurations.
- Linking (page 82) provides details about how Xcode links code together into images and how you can customize this task.
- Reducing Build Times (page 90) describes the Xcode features that help reduce build times: precompiled headers, predictive compilation, and distributed builds.

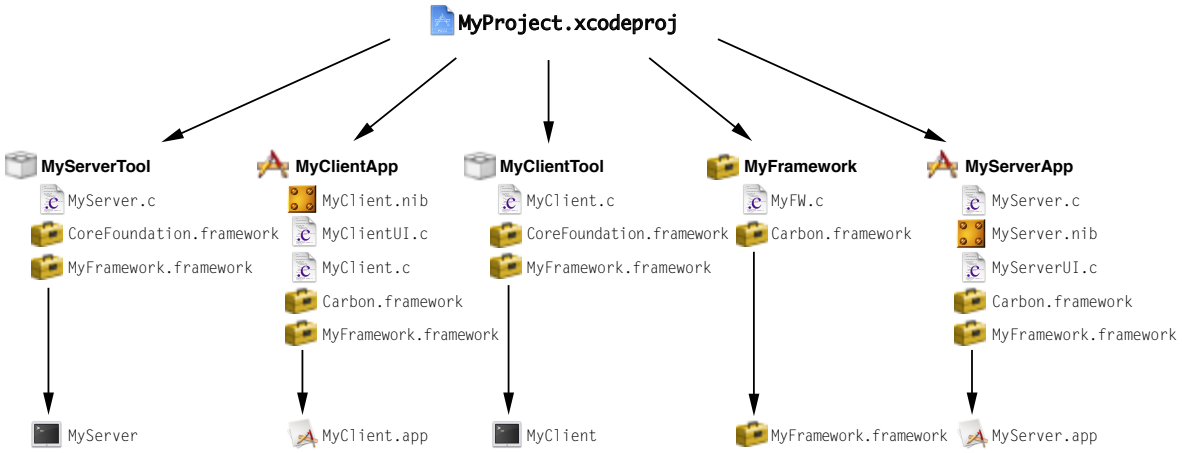
This document also has a revision history.

Targets

The organizing principle of the Xcode build system is the *target*. A *target* contains the instructions for building a finished product from a set of files in your project—for example, a framework, library, application, or command-line tool. Each target builds a single product. A simple Xcode project has just one target, which produces one product from the project's files. A larger development effort with multiple products may require a more complex project containing several related targets. For example, a project for a client-server software package may contain targets that create a client application, a server application, command-line tool versions of the client and server functionality, and a private framework that all the other targets use.

Figure 1-1 shows the targets you may have in a project such as the one described earlier, and the products that those targets create. The following list shows the project's main components.

Figure 1-1 Targets and products



- **Project**

  `MyProject.xcodeproj`: The file package containing the information about the project and its targets.

- **Targets**

  MyServerTool: Builds a server-side command-line tool.

  MyClientApp: Builds a client-side application.

  MyClientTool: Builds a client-side command-line tool.

  MyFramework: Builds a framework.

  MyServerApp: Builds a server-side application.

- **Products**

  `MyServer`: A server-side command-line tool

  `MyClient.app`: A client-side application.

  `MyClient`: A client-side command-line tool.

  `MyFramework.framework`: A framework.

  `MyServer.app`: A server-side application.

When you initiate a build, Xcode builds the product specified by the *active target* and any targets on which the active target *depends* (see Defining Target Dependencies (page 24) to learn about dependent targets). In the Groups & Files list, the *active target* is marked by a checkmark in a green circle. You can also see which target is active, as well as change the active target, in the Active Target pop-up menu in the project and Build Results windows. When you execute the Build command, Xcode builds the product of the active target.

This chapter describes Xcode targets and how to manage them.

Target Overview

A *target* is a blueprint for creating a product. To build a product, the build system takes a set of inputs—source files and the instructions for processing them—and produces an output (such as an application or a framework). The inputs to the build system are:

- **Build settings.** A build setting contains information on how to perform the operations required to build a product. For example, a build setting can specify command-line options for Xcode to pass to the compiler. Build settings can contain tool-specific options, paths to build files and product directories, and other information used by Xcode to determine how to perform build operations.

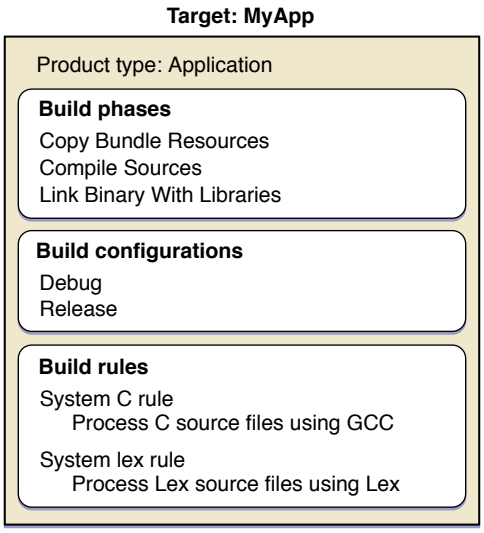
  Because build settings represent variable aspects of the build process, they are the most flexible means of customizing the build process. See Build Settings (page 54) for more information on using build settings.

- **Files.** For each product, the build system takes a set of files as inputs and performs various operations on them—such as compiling, linking, copying, and so forth—to arrive at the final output. Examples of source files are header files, implementation files, resource files, and so forth.

- **Per-file compiler flags.** Per-file compiler flags are additional options Xcode passes to the compiler as it compiles each source file. See Per-File Compiler Flags to learn more.

A target, illustrated in Figure 1-2, organizes the inputs required to create a single product.

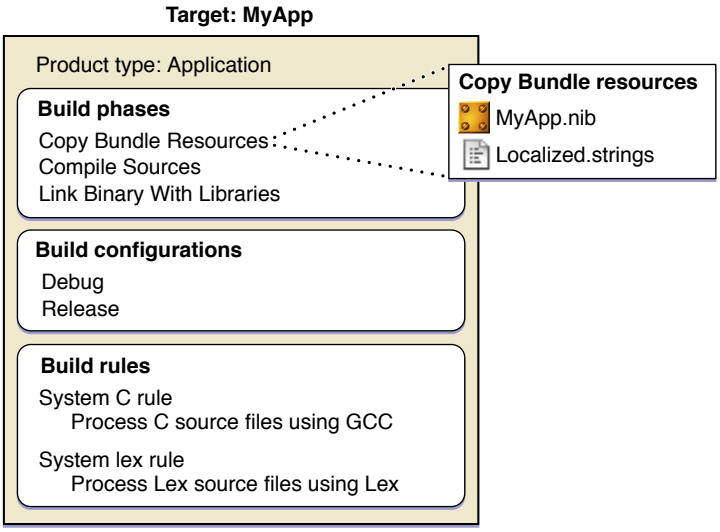
Figure 1-2 A target



A target contains:

- **Build phases.** Build phases organize the build files of a target according to the operations required to build the target's product. Each build phase consists of a list of input files and a task to be performed on each of those files. Common build phases include compiling files, linking object files, and copying resource files. Figure 1-3 shows the files that may be in a typical build phase.

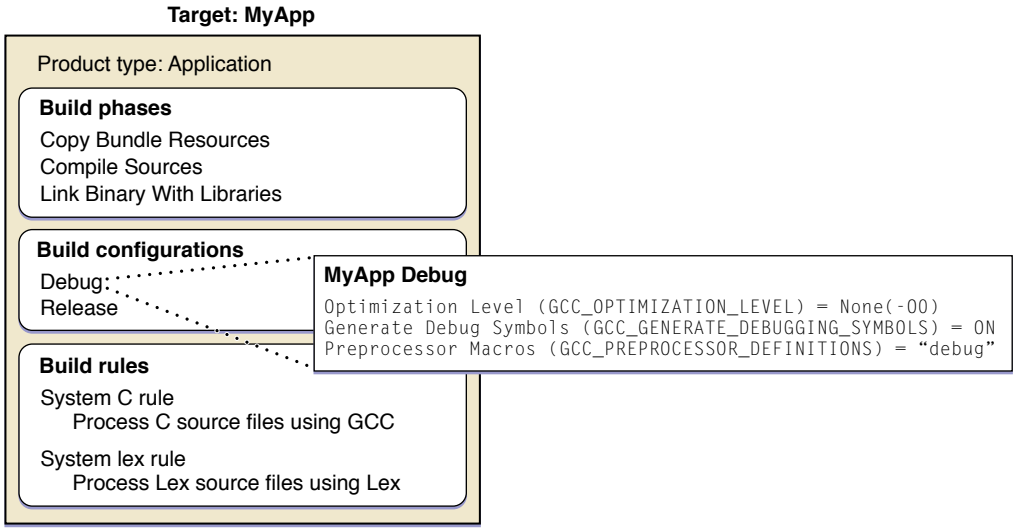
Figure 1-3 Files in a Copy Bundle Resources build phase



Xcode populates each new target with a predetermined set of build phases. You can add or remove build phases to change the operations Xcode performs when it builds the target. For further details on build phases, see Build Phases (page 33).

- **Build configurations.** Each target contains one or more named collections of build settings, called build configurations. A build configuration specifies a set of build settings that are used to build a target's product in a particular way. By defining multiple configurations for a target, you can quickly build different variations on the target's product. You can also modify all of a target's build configurations at once to define build settings that are shared by all configurations. Figure 1-4 shows some of the build settings that might be defined by one build configuration of a target.

Figure 1-4 Build settings in a build configuration



- **Build rules.** A target's build rules determine how each source file in the target is processed. Each build rule consists of a *condition*—such as files matching the type `.c`—and an *action*. Typically, this action specifies the tool Xcode invokes to process files that meet the condition; this is how Xcode determines the compiler to use for compiling source files, for example. Xcode maintains a list of build rules for targets that use the native build system. For targets that use an external build tool, you maintain the instructions for processing the build files with a makefile or other method.

> **Important:** Build rules apply only to the Compile Sources and Build ResourceManager Resources build phases.

Xcode defines a default set of build rules, but you can define custom build rules for a target. For more information on using build rules, see Build Rules (page 47).

A target and the product that it creates are closely related; every target has an associated *product type*. When you create a target from a target or project template, you choose the target's product type, as described in Creating Targets (page 14).

Based on the product type, Xcode specifies initial values for certain product-specific build settings. For example, when you create a target that builds an application, Xcode assigns it the build setting specification `INSTALL_PATH = "/Applications"` based on the product type. Any subsequent changes you make to the target after creating it may override these default values. Note that the project and target templates contain additional configuration information that Xcode uses when it creates targets.

Managing Targets

When you need to work on a software package with multiple products—for example, an application, a command-line tool, and a framework—an easy way to group all the pieces into one project is to use multiple targets. This section describes the tasks you may need to perform on a project that requires multiple targets, including creating, editing, duplicating, and removing targets. It also teaches how to create target dependencies so that, for example, Xcode builds your framework before building the application that uses it.

Creating Targets

When you create a project from one of the Xcode project templates, Xcode automatically creates a target for you. If, however, your project needs to contain more than one target—usually because you are creating more than one product—you can also add targets to an existing project.

If you are adding targets to your project, chances are you've already made a number of decisions about product type, programming language, and framework. Xcode provides a number of *target templates* to support your choices. The selection of target templates is similar to the selection of project templates. The target specifies the target's product type, a list of default build phases, and default definitions for some build settings. A target template typically includes all build settings and build phases required to build an instance of the specified product. Unlike the project templates provided by Xcode, the target templates do not specify any default files; you must add files to the target yourself, as described in Managing Target Files (page 28).

> **Note:** Targets created with target templates have no framework references. Any frameworks or libraries that the target is configured to link with are added to the Other Linker Flags build setting. You can add framework references to the target yourself, as described in Figure 1-10 (page 31).

To create a target and add it to an existing project:

1. Choose **Project** > **New Target**.

Xcode displays the New Target assistant, which lets you choose from a number of possible target templates. Each target template corresponds to a particular type of product, such as an application or loadable bundle.

2. Select one of the templates and click Next.
3. Enter the name of the target.

   If more than one project is open, you can choose which project to add the target to from the Add to Project pop-up menu.
4. Click Finish.

   Xcode creates a target configured for the specified product type. Xcode also creates a reference to the target's product and places it in your project, although the product does not exist on the file system until you build the target.

Table 1-2 lists templates that create targets using the native build system. Because it performs all target and file-level dependency analysis for targets using the native build system, Xcode can offer detailed feedback about the build process and integration with the user interface for these targets.

Table 1-1 iOS target templates

Target template	Creates
Cocoa Touch	
Application	An application based on the Cocoa Touch framework.
Static Library	A static library.
Unit Test Bundle	A target that compiles test code into a bundle, links it with the Unit Test framework and an executable to be tested, and runs a series of unit tests.

Table 1-2 Mac OS X target templates

Target template	Creates
Cocoa	
Application	An application, written in Objective-C or Objective-C++, that links against the Cocoa framework.
Dynamic Library	A dynamic library that links against the Cocoa framework.
Framework	A framework based on the Cocoa framework.
Loadable Bundle	A bundle, such as a plug-in, that can be loaded into a running program.

Target template	Creates
Shell Tool	A command-line utility based on the Cocoa framework.
Static Library	A static library, written in Objective-C or Objective-C++, based on the Cocoa framework.
Unit Test Bundle	A target that compiles test code into a bundle, links it with the Unit Test framework and an executable to be tested, and runs a series of unit tests.
Application Plug-in	
Automator Action	A target that builds an Automator action.
BSD	
Dynamic Library	A dynamic library, written in C, that makes use of BSD.
Object File	A single-module object file using BSD API.
Shell Tool	A command-line utility, written in C.
Static Library	A static library, written in C, that makes use of BSD.
System Plug-in	
Generic Kernel Extension	A kernel extension
IOKit Driver	A device driver that uses the I/O Kit.

In addition to these target templates, Xcode defines a handful of target templates that do not necessarily correspond to a particular product type. These targets are known as *special targets* and can be used to:

- Build a group of targets together
- Copy files to a specific file system location
- Build a product using an external build system
- Run a shell script

These are the special target types:

- **Aggregate Target**

Xcode defines a special type of target that lets you build a group of targets at once, even if those targets do not depend on each other. An aggregate target has no associated product and no build rules. Instead, an aggregate target depends on each of the targets you want to build together. For example, you may have a group of products that you want to build together. You would create an aggregate target and make it depend on each of the product targets. To build all the products, just build the aggregate target.

An aggregate target may contain a custom Run Script build phase or a Copy Files build phase, but it cannot contain any other build phases. Any build settings that the aggregate target contains are not interpreted but are passed to the build phases that the target contains. For more information on aggregate targets, see Defining Target Dependencies (page 24).

- **Copy Files Target**

  A Copy Files target is an aggregate target that contains only one build phase, a Copy Files build phase. Building a Copy Files target simply copies the associated files to the specified destination in the file system. Copy Files targets are useful if you have custom build steps that require files that are not specific to any other targets to be copied. While Copy Files build phases allow you to add a step to the build process for a single target that copies files in that target, a Copy Files target lets you copy files that are not specific to any one target. For example, if your project has several targets that require the same files to be installed at a particular location, you can use a Copy Files target to copy the files, and make each of the other targets depend upon the Copy Files target. For more information on the Copy Files build phase, see Copy Files Build Phase (page 41).

- **External Target**

  Xcode allows you to create targets that do not use the native Xcode build system but instead use an external build tool that you specify. For example, if you have an existing project with a makefile, you can use an external target to run `make` and build the product.

  An external target creates a product but does not contain build phases. Instead, it calls a build tool in a directory. With an external target, you can take full advantage of the Xcode text editor, class browser, and source-level debugger. However, many Xcode features—such as Fix and Continue—rely on the build information maintained by Xcode for targets using the native build system. As a result, these are not available to an external target. Furthermore, you must maintain your custom build system yourself. For instance, if you need to add files to an external target built using `make`, you must edit the makefile yourself. To learn how to work with non–Xcode-based projects in a more straightforward way, see Using the Organizer.

- **Shell Script Target**

  A Shell Script target is an aggregate target that contains only one build phase, a Run Script build phase. Building a Shell Script target simply runs the associated shell script. Shell Script targets are useful if you need to perform custom build steps. While Run Script build phases allow you to add custom steps to the build process for a single target, a Shell Script target lets you define a custom build operation that you

can use with many targets. For example, if your project has several targets that use the files generated by a Shell Script target, you can make each of those targets depend upon the Shell Script target. For more information on using shell scripts as part of the build process, see Run Script Build Phase (page 43).

Editing Targets

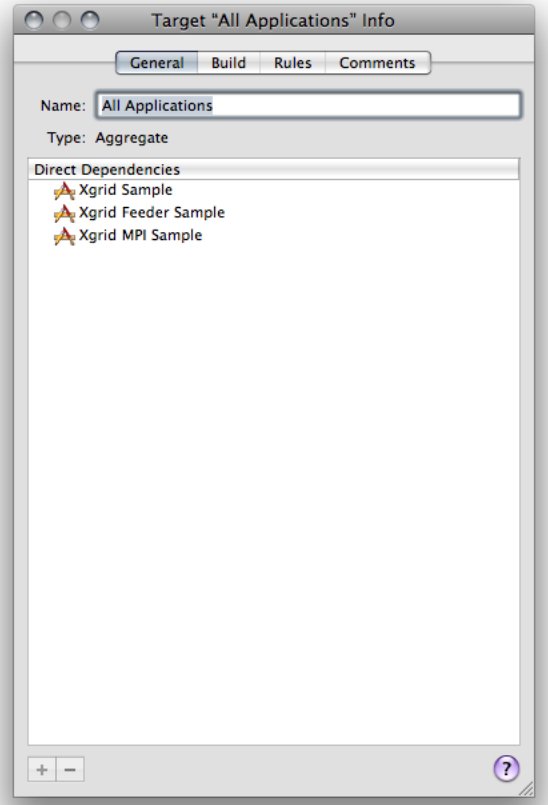
To edit a target you use the *target editor*, which allows you to view and modify target settings. As described in Target Overview (page 11), a target defines the instructions necessary to create a product. Each target has an associated set of files, tasks, and settings that together constitute these instructions. You can edit and view many of these settings in the target editor.

The information available in the target editor varies, depending on the type of target. The target editor for native targets lets you view and edit all target settings and build information associated with that target. The target editor for external targets contains only general information. To edit build information for either of these types of targets, you must use the target editor.

To learn how to modify the files and build phases associated with a target, see Managing Target Files (page 28) and Build Phases (page 33), respectively.

The Target Info window is the primary mechanism for viewing and editing target information. Figure 1-5 shows the target editor.

Figure 1-5 General pane in the Target Info window



The target editor contains the following panes:

- **General.** The General pane contains global information about a target, such as its name, the name of the associated product, and target dependencies. To learn more, see Editing General Target Settings (page 20).
- **Build.** The Build pane lets you view and edit build settings for the target. It is described in Editing Build Settings.
- **Rules.** The Rules pane displays the current system build rules, as well as any custom build rules defined for the current target. For more information, see Build Rules (page 47).
- **Properties.** The Properties pane lets you edit information property list entries for targets that create products requiring `Info.plist` files, such as applications and other bundles.

> **Note:** If the target does not have an `Info.plist` file, the Properties pane is not visible in the target editor. This pane is described in Editing Information Property List Entries (page 20).

- **Comments.** The Comments pane lets you associate notes or other documentation with the target. See Adding Comments to Project Items for more information.

Editing General Target Settings

In the General pane of the target editor, you can edit basic target settings for any target in an Xcode project. The General pane contains the following settings:

- **Name.** The name Xcode uses to refer to the target. It may also specify the product name (see *Xcode Build Setting Reference* for more information).
- **Type.** The type of product created by the target. The product type is determined when you create the target; you cannot change it.
- **Direct Dependencies list.** The Direct Dependencies list identifies the targets upon which the current target depends. See Adding Target Dependencies (page 25) to learn more about target dependencies.

Editing Information Property List Entries

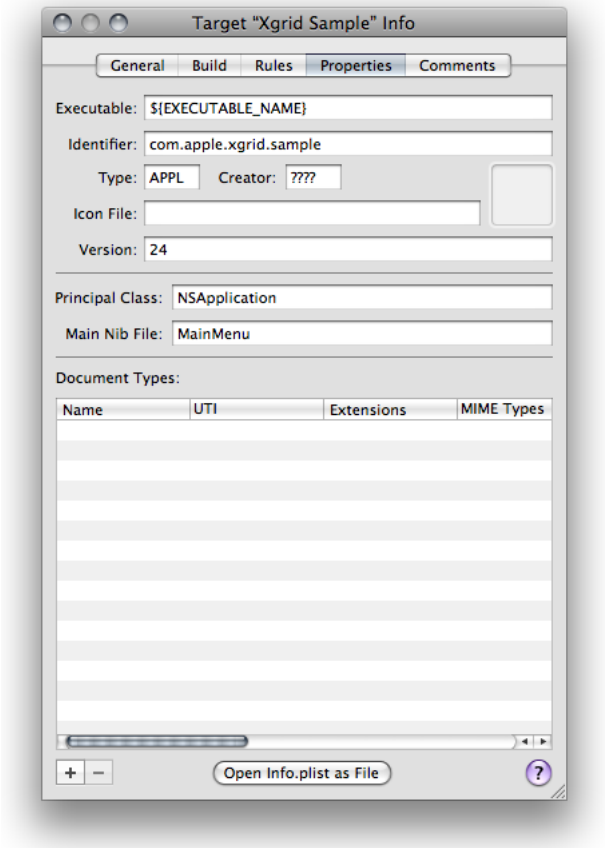
Information property list entries contain information used by the Finder and other system software. This information ends up in a file, called `Info.plist` that is contained within the product bundle. If a product does not come in the form of a bundle, it has no `Info.plist` file.

The `Info.plist` file defines for the Finder such things as the bundle's icon, the documents it can open, the URLs it can handle, and so on. Unlike build settings, property list entries do not affect the build process; they are copied into the bundle's `Info.plist` file at the end of the build process. Note, however, that Xcode evaluates build settings referenced in the `Info.plist` file, and you can have Xcode preprocess the `Info.plist` file with the GCC preprocessor. For a list of entries used by the system, see *Runtime Configuration Guidelines*.

Information Property List Entries for Native Targets

The Properties pane of the target editor, shown in Figure 1-6, allows you to edit information property list entries for native targets. This pane is visible only for targets that create products with `Info.plist` files.

Figure 1-6 Target editor: Properties pane



The top section of the Properties pane allows you to edit essential information about the product, such as the name of the associated executable, bundle identifier, type and creator, version information, and the icon to associate with the finished product. Note that the name of the icon here must match the name of an icon file that is copied into the `Resources` directory of the product bundle.

The Principal Class and Main Nib File settings are specific to Cocoa applications and bundles, and Automator actions. Principal Class corresponds to the `NSPrincipalClass` property list key. Main Nib File specifies the nib file that's automatically loaded when the application launches. It corresponds to the information property list key `NSMainNibFile`.

The Document Types list allows you to specify which documents your finished product can handle. These are the aspects each entry in the list specifies:

- **Name.** The name of the document type. For example, “Apple Sketch Document.”
- **UTI.** A list of Uniform Type Identifiers (UTIs) for the document. UTIs are strings that uniquely identify abstract types. They can be used to describe a file format or data type but can also be used to describe type information for other sorts of entities, such as directories, volumes, or packages. For more information on UTIs, see *Uniform Type Identifiers Overview*.
- **Extensions.** A list of the filename extensions for this document type. Don’t include the period in the extension. For example, `sketch` and `draw2`.
- **MIME Types.** A list of the MIME types for the document.
- **OS Types.** A list of four-letter codes for the document. These codes are stored in the documents’ resource or information property list files. For example, `sktc`.
- **Class.** The subclass of `NSDocument` that this document uses. Use this field only if you’re writing a document-based Cocoa application.
- **Icon File.** The name of the file that contains the document type’s icon.
- **Store Type.** The store type is the type of backing store to use to serialize the document. For more information on store types, see *Core Data Programming Guide*.
- **Role.** A description of how the application uses the documents of this type. You can choose from three values:
  - Editor. The application can display, edit, and save documents of this type.
  - Viewer. The application can display, but not edit, documents of this type.
  - None. The application can neither display nor edit documents of this type but instead uses them in some other way. For example, the Finder could declare an icon for font documents.
- **Package.** Specifies whether the document is a single file or a file package.

> **Note:** Instead of the Document Types table, the Properties pane for Automator action targets provides a different interface for editing Automator-specific property list keys. These property list keys are discussed in *Automator Programming Guide*.

You may have additional keys that you need to include in your `Info.plist` file; for example, applications that include Apple Help help books need two additional `Info.plist` entries. To add these additional keys, you can edit the `Info.plist` file directly by clicking the Open Info.plist as File button at the bottom of the Properties pane. Note that you can refer to build settings in the `Info.plist` file. For example, `$(PRODUCT_NAME)` expands to become the base name of the product built by the target.

You can set properties on multiple targets; simply select the targets in the project window and open the target editor. In the Properties pane, you can edit the values of properties that apply to more than one target.

Preprocessing Info.plist Files

You can specify that Xcode process the `Info.plist` file using the GCC preprocessor. This allows you to include headers, use `#if`-style conditional statements, and take advantage of macro expansion in the `Info.plist` file.

To preprocess the `Info.plist` file in a target, turn on the Preprocess Info.plist File (INFOPLIST_PREPROCESS) build setting.

You can also use the following build settings to control preprocessing of `Info.plist` files:

- Info.plist Preprocessor Definitions (INFOPLIST_PREPROCESSOR_DEFINITIONS). A list of preprocessor macros to define when preprocessing the `Info.plist` file. For example, `DEBUG=1`.
- Info.plist Preprocessor Prefix File (INFOPLIST_PREFIX_HEADER). The path to a prefix file to include when preprocessing the `Info.plist` file. Use a project-relative or an absolute path.
- Info.plist Other Preprocessor Flags (INFOPLIST_OTHER_PREPROCESSOR_FLAGS). A list of additional flags to pass to GCC when preprocessing the `Info.plist` file. For more on the flags you can pass here, see *GNU C 4.0 Preprocessor User Guide* .

Duplicating Targets

There are two main reasons you might need to duplicate a target: You require two targets that are very similar but contain slight differences in the files or build phases that they include, or you have a complicated set of options that you build with and would prefer to simply start with a copy of a target that already contains those build settings.

Xcode allows you to duplicate a target, creating a copy that contains the same files, build phases, dependencies and build configuration definitions of the original.

To create a copy of a target:

1. Select the target you want to copy in the Groups & Files list.
2. Choose one of these:
    - Edit > Duplicate
    - Groups & Files list shortcut menu > Duplicate

Removing Targets

When your project contains targets that are no longer in use, you may want to remove them to reduce clutter.

To remove a target from a project:

1. Select the target to delete in the Groups & Files list.

2. Press the Delete key or choose:
    - Edit > Delete.

When you delete a target, Xcode also deletes the product reference for the product created by that target and removes any dependencies on the deleted target.

Defining Target Dependencies

In a complex project, you may have several targets that create a number of related products. Frequently, these targets need to be built in a specific order. Returning to the example of the client-server software package created by the project shown in Figure 1-1 (page 10), you see that the client application, server application, and command-line tool targets each link to the private framework created by another target in the same project.

Before the application and command-line tool targets can be built, the framework target must be built. Because they require the private framework in order to build, each of the application and command-line tool targets is said to depend upon the target that creates the framework. You can use a *target dependency* to ensure that Xcode builds targets in the proper order; in this example, you would add a dependency upon the framework target to each of the application and command-line tool targets.

However, the applications and command-line tool in the client-server package must still be built individually. None of these targets requires the product created by any target other than the framework target. Xcode provides another mechanism for grouping targets that you want to build together but that are otherwise unrelated; this is an *aggregate target* (see Creating Targets (page 14) for details). This section shows how to add a target dependency and create an aggregate target, and gives an example of how you can use these tools to organize a software development effort with multiple products and projects.

Creating Aggregate Targets

To build several targets together, even if they aren't dependent on each other, create an *aggregate target*. As described in Creating Targets (page 14), an *aggregate target* does not produce a product itself and it does not contain build rules or information property list entries. Instead it exists so that you can make it dependent on other targets. When you build the aggregate target, the build system builds the targets it depends on sequentially or in parallel. To learn about building targets concurrently, see Building in Parallel.

To create an aggregate target:

1. Choose Project > New Target.
2. Select Aggregate from the New Target Assistant.

For each target you want to build with this aggregate target, add a target dependency to the aggregate target, as described in the Adding Target Dependencies (page 25).

Note that, although it does not contain any other build phases, an aggregate target can include a Run Script or Copy Files build phase. Any build setting defined for the aggregate target will not be interpreted, but will be passed to any build phases that the target contains.

Adding Target Dependencies

When you build a target (target A) with a dependency upon another target (target B), Xcode makes sure that target B is built and up to date before building target A. That way, you can be sure that when target A needs the product created by target B, target B is built before target A. In addition, if there are errors building target B, Xcode doesn't build target A.

You can view and modify a target's dependencies in two ways:

- In the General pane of the target editor. The Direct Dependencies list shows the targets upon which the current target depends.

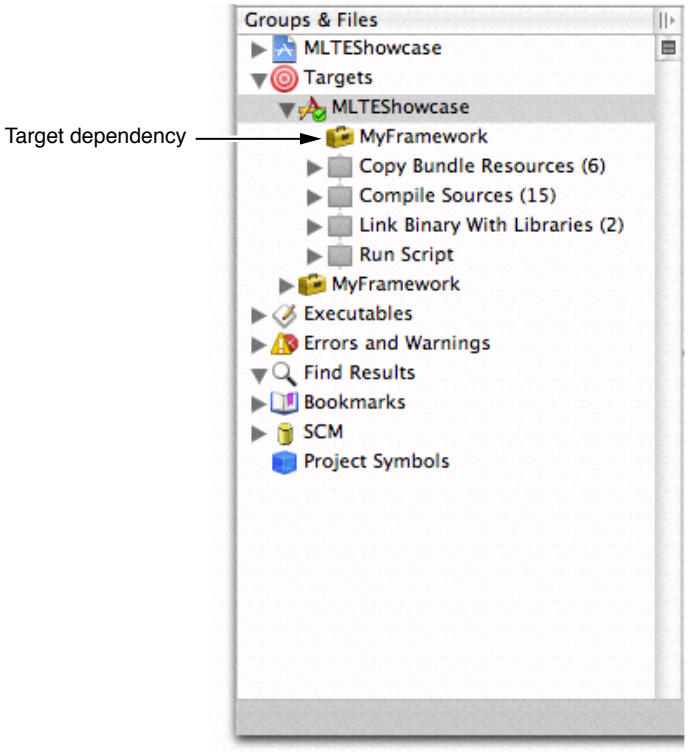
  To add a target dependency, click the plus-sign button. (For the plus sign button to be available, the project must contain or reference more than one target.) The list of targets shown includes all the other targets in the current project, as well as the targets in any referenced projects. Targets in referenced projects are grouped according to the project to which they belong. For more information on referencing other projects, see Referencing Other Projects. You may also drag a target from the Targets group to the Direct Dependencies list.

  To remove a target dependency, select it in the list and click the minus button.

- In the Groups & Files list. Open the Targets group; the targets that the current target depends on are listed before the build phases in the target. You can make the current target depend on another target by dragging that target to the current target in the Groups & Files list.

Figure 1-7 shows a target dependency in the Groups & Files list.

Figure 1-7 Groups & Files list: Target dependency



Removing Target Dependencies

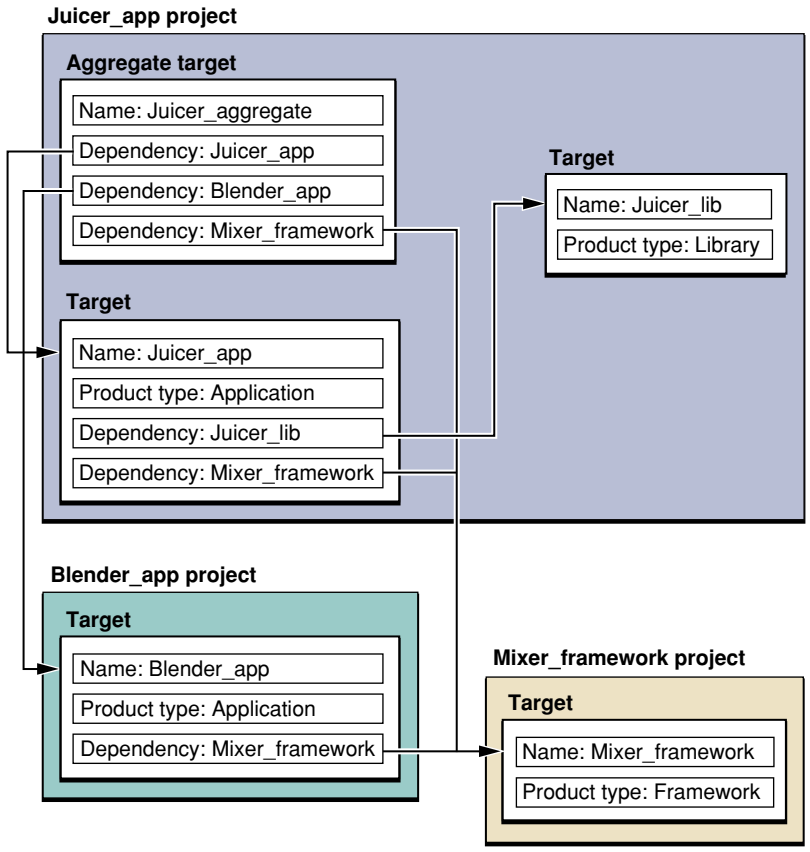
To remove a target dependency:

1. Select the target to modify in the Groups & Files list.
2. Open the target editor and display the General pane.
3. In the Direct Dependencies list, select the target dependency to delete.
4. Click the minus (-) button.

Multiple Target Example

Suppose that your organization has two teams working on separate applications, that one application includes an internal library, and that each application relies on a framework supplied by a third team. Figure 1-8 shows one way to set up your software development with three Xcode projects, using project references, target dependencies, and aggregate targets to relate the various products.

Figure 1-8 Three projects with dependencies



In Figure 1-8, the `Juicer_app` project contains the `Juicer_app` target for building the Juicer application, and the `Juicer_lib` target for building an internal library. The application target depends on the library target, and also has a cross-project dependency on the `Mixer_framework` target in the `Mixer_framework` project. Finally, the `Juicer_app` project contains the `Juicer_aggregate` target as a convenience for building the entire suite of projects.

> **Note:** Aggregate targets are typically used to build targets that don't otherwise depend on each other.

In Figure 1-8, the `Blender_app` project contains a target for building the Blender application. The Blender target also has a cross-project dependency on the Mixer framework.

Finally, the `Mixer_framework` project contains a target for building the Mixer framework, used by both the Juicer and Blender applications.

Given this combination of projects, targets, and dependencies, the following statements are true:

- Building the Juicer target builds the Juicer library if it needs updating, and also builds the Mixer framework, if it needs updating.
- Building the Juicer aggregate target builds the Juicer application, which builds the Juicer library if it needs updating. The aggregate target also builds the Blender application and the Mixer framework.
- Building the Juicer library does not cause any other targets to be built.
- Building the Blender target builds the Mixer framework if it needs updating.

Managing Target Files

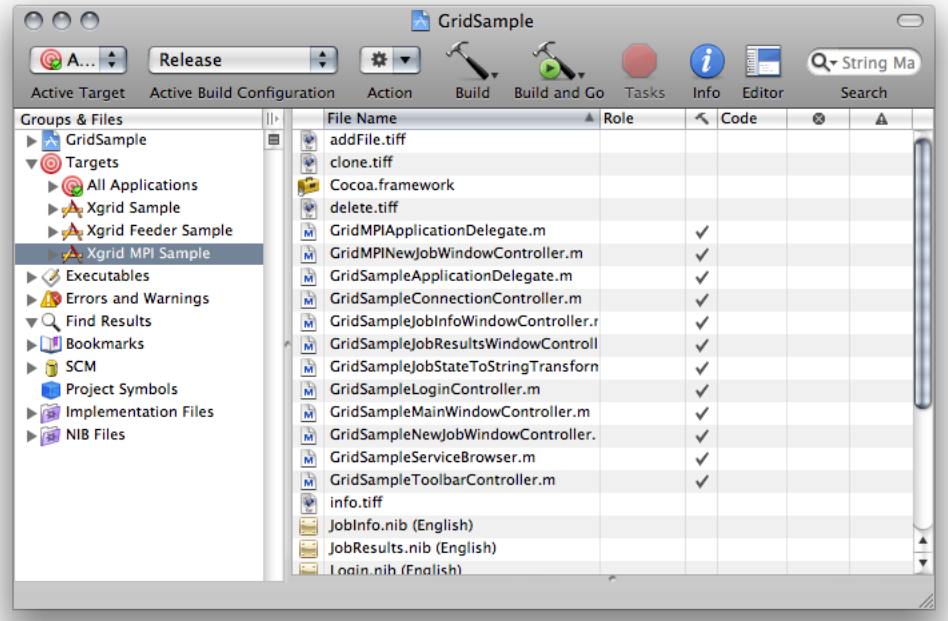
When you add files to a project, you can have Xcode also add those files to one or more targets. This is the easiest way to add files to a target. However, you may have existing files in your project that you wish to add to a target, or find that you no longer need a target file. This section describes how to view the files in a target and shows you how to add and remove target files.

Viewing the Files in a Target

To view all of the files included in a target, select the target in the Groups & Files list; all targets are grouped under the Targets group. The detail view shows all the files and folders in the target, similar to what you see in Figure 1-9. If one is not already open in the current window, open a detail view by choosing:

- View > Detail

Figure 1-9 Target files in the project window



You can also see a target's files grouped according to the operation performed on those files during the build process. If you open a target in the Groups & Files list (by clicking the disclosure triangle next to it), you see the *build phases* for that target. Build phases, described in further detail in Build Phases (page 33), represent a task performed when the target is built. To see the files in a particular build phase, do one of the following in the Groups & Files list:

- Select the build phase. The build phase files appear in the detail view.
- Open the build phase. The build phase files appears under the build phase name.

Adding and Removing Target Files

Xcode provides several ways to modify a set of target files. In the project window, you can see all the files in a target and you can add files to, or remove files from, the target. You can also use file-info editors to see which targets include them and change their target membership.

To add files to a target, you can:

- Drag the file reference or references to the appropriate build phase of the given target in the Groups & Files list. For native targets, Xcode does not let you drag a file to a build phase that does not accept that type of file as an input. For example, you cannot drag a nib file to the Compile Sources build phase. See Build Phases (page 33) for more information on the available build phases and their files.
- Specify that a file be included in a target when you add that file to your project, as described in Managing Files and Folders in a Project.
- Add a file to the active target. Find the file in the detail view or in the Groups & Files list and select the checkbox in the Target column for that file. If the Target column is not visible, select either these:
  - Groups & Files list header shortcut menu > Target Membership
  - Detail view header shortcut menu > Target

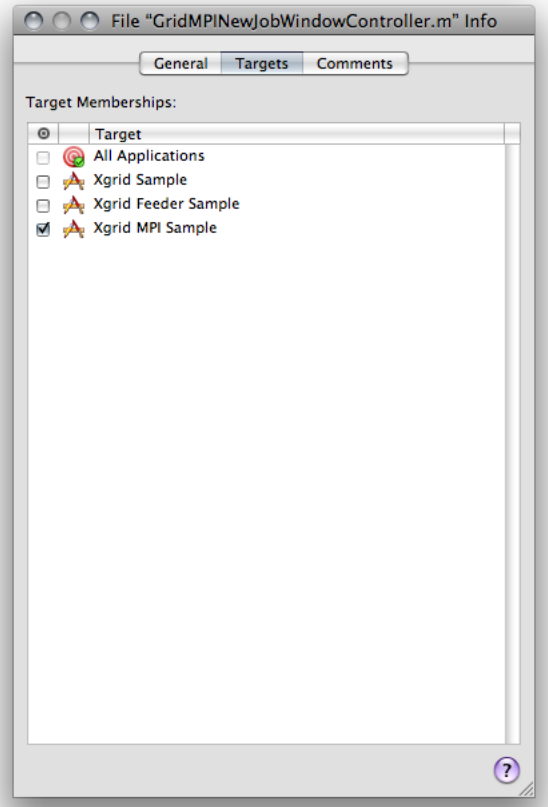
To remove a file from a target, in the Groups & Files list:

1. Open the target to reveal its build phases.
2. Open the build phase to which the file belongs.
3. Select the file to remove from the target, and either:
   - Choose Delete from the shortcut menu
   - Choose Edit > Delete

To remove a file from the active target, find the file in the detail view or in the Groups & Files list and deselect the checkbox in the Target column for that file. Xcode removes the file from the target but does not remove the file reference from the project.

To see which targets a file belongs to and modify the file's target membership, select the file reference in the project window, open the file-info editor, and display the Targets pane, shown in Figure 1-10.

Figure 1-10 File-info editor: Targets pane



The *Targets Membership table* shows all the targets in the current project. The checkbox next to each target indicates whether the file is included in that target.

If you have multiple files selected, the Targets pane may show a dash in the checkbox next to a target. This indicates that some of the selected files are included in that target, while others are not. If the checkbox next to a target is dimmed, that target does not contain a build phase appropriate for processing the selected file or files.

Setting the Role of Header Files

Header files have a special purpose in a target: They publish the programming interface to symbols defined or implemented in the target's implementation files. Implementation files in a target use header files to gain access to symbols defined in other implementation files. But implementation files may also require access to the programming interface to frameworks, libraries, or plug-ins that are not part of the project.

When you define a target, you may implement symbols you want to make public to clients of the target's product. But, to make your product easy to use (and to keep implementation details hidden), you may want to keep many of its symbols inaccessible to clients of your product. Also, if you develop products to be used both by the development team and by end users, you may need to publish interfaces to symbols that must be used only by other team members but that end users must not use. You use a header file's *role* within its target to specify the purpose of the header file.

To set the role of header files in a target:

1. Select the target containing the header files whose role you want to set.
2. Select the target's Copy Headers build phase.
3. Choose the desired role for each header file from the Role column in the detail view.

There are three roles available: `public`, `private`, and `project`. Table 1-3 describes these roles.

Table 1-3 Header file roles

Role	Header file disposition
`public`	The header file is included as part of the build product. Its location is specified by the Public Headers Folder Path (PUBLIC_HEADERS_FOLDER_PATH) build setting. You should publish as `public` only interfaces that are finalized and meant to be used by your product's end users.
`private`	The header file is included as part of the build product. Its location is specified by the Private Headers Folder Path (PRIVATE_HEADERS_FOLDER_PATH) build setting. You should publish as `private` interfaces that you don't intend to be used by end users or that are in early stages of development.
`project`	The header file is available only for use by implementation files in the current project. This header file is not included as part of the built product.

Build Phases

A *build phase* represents a task to be performed on a set of files. Each build phase specializes in a specific kind of file, such as header files, source files, resource files, and frameworks and libraries. Each build phase also performs specific operations on the files associated with it. That way, changes in the way files of the same kind are processed can be made by customizing the operations that the build phase performs.

This chapter provides an overview of build phases, explains how Xcode determines the order in which build phases are processed to build a product, describes in detail some of the build phases available, and explains how build rules allow you to customize the Compile Sources build phase and the Build ResourceManager Resources build phase in a target.

Build Phase Overview

A build phase performs a set of operations on a group of files as part of building a product. To illustrate this process, think of the tasks you would perform to prepare for and deliver a presentation; many of the required tasks contain a set of inputs and outputs:

1. Specify the purpose of the presentation and identify the audience.

   This task has no prerequisites but produces a set of criteria that serve as the guiding principles for the presentation.

2. Gather the appropriate data, such as articles, reports, and surveys.

   Information gathering requires a set of criteria that specify what to look for and where to look for it. The outcome is a set of documents containing the data gathered.

3. Purchase a business suit.

   This task has no inputs or outputs that directly relate to the presentation.

4. Analyze the data by collating facts, extrapolating trends, and summarizing representative opinions.

   This task uses the data gathered in task 2 and produces one document containing the information on which the presentation is based.

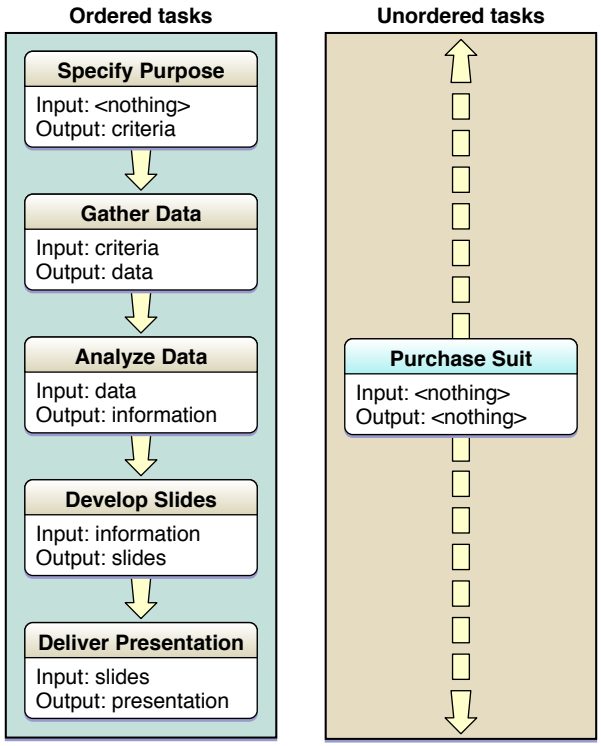
5. Develop a set of slides, including compelling illustrations, from the information produced by your analysis.

   This task requires the information produced in task 4 and produces a set of slides and illustrations, which could be presented using Keynote.

6. Deliver the presentation by showing the slides and illustrations in sequence and by engaging the audience with a loud, clear voice while maintaining eye contact with them.

The order in which build phases are executed in a target depends on their inputs and outputs. Most of the tasks shown earlier contain a set of inputs and outputs that associate them with each other in an anterior/posterior relationship. Tasks that contain inputs that are the outputs of other tasks or outputs that are the inputs of other tasks are *ordered tasks*. These are tasks that are executed in the order determined by their inputs and outputs. *Unordered tasks*, on the other hand, are tasks with no inputs and outputs, or tasks whose inputs are not the outputs of other tasks and whose outputs are not the inputs of other tasks. In Figure 2-1 the presentation tasks shown earlier are categorized as ordered and unordered.

Figure 2-1 Presentation tasks

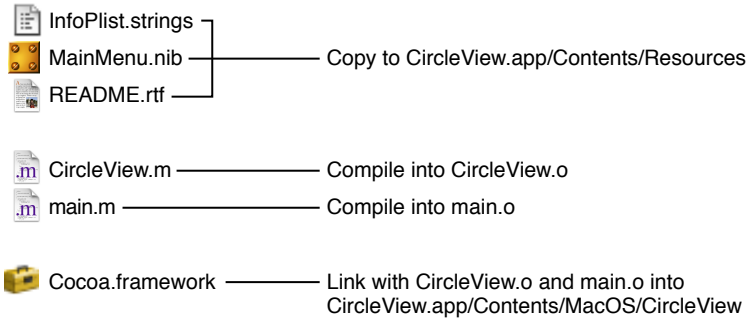


Most of the tasks illustrated depend on another task's outputs. The Specify Purpose task is the only task among the ordered tasks that doesn't depend on the output of any other task; therefore, it's performed first. The Purchase Suit task is the only unordered task; that is, it's neither the anterior nor the posterior of other tasks. Therefore, it can be executed at any time during the preparation of the presentation.

Using Build Phases

To build an application, you typically compile source files and link them to system frameworks and libraries. Figure 2-2 shows some of the operations needed to build the CircleView example application.

Figure 2-2 Building an application



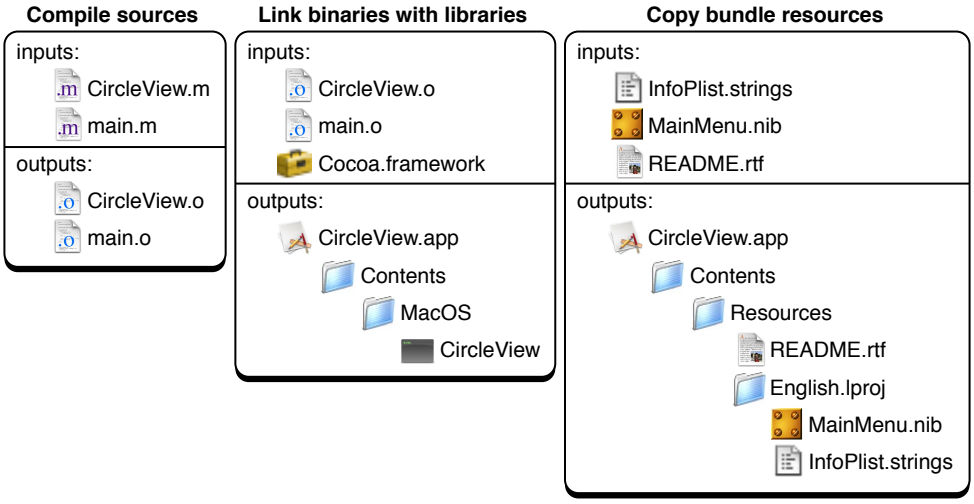
This simple project shows that there are many things to take into account when building a product. You have to copy resource files to the appropriate places, compile implementation files into object files, and link object files with the appropriate frameworks to produce a binary file. If you modify an implementation file, you must compile it, and link the generated object file with the other object files and the necessary frameworks.

If you analyze the files and the operations illustrated in Figure 2-2 (page 35), you notice that the project contains three types of files: Files that are copied or installed, files that are processed or compiled to produce intermediate files (such as object files), and frameworks that are linked with the object files to produce a binary file. To build a product, appropriate operations are performed on the files, depending on their type.

Build phases associate groups of files with operations to be performed on them in order to build a product; for example, installing header files within frameworks, compiling source files, linking object files with system libraries or frameworks, and so on. As you add files to a project, Xcode associates them with the appropriate build phase, based on the files' type (specified by each file's extension). If you want to compile implementation

files with a different compiler, you make the change once, at the build phase level (see Compile Sources Build Phase (page 41) and Build Rules (page 47) for details). Build phases help make the process of building an application, plug-in, library, or framework, understandable and easy to customize, as illustrated in Figure 2-3.

Figure 2-3 Building an application using build phases



A build phase operates both on its inputs, which are the files associated with it (either by you or intrinsically by Xcode), and on its outputs, which are the files produced after the build phase is executed. Each build phase executes its task by invoking tools to perform the operations needed to accomplish the task.

Xcode offers several build phases, which are shown in Table 2-1. The name of the build phase reflects the task performed by that build phase.

Table 2-1 Build phases available in Xcode

Build phase	Description
Copy Headers	Installs header files with `public` or `private` roles in the appropriate locations in the product. See Setting the Role of Header Files (page 31) for details.
Copy Bundle Resources, Build Java Resources	Installs files by copying the associated files from the project to the `Resources` directory in the product.
Copy Files	Installs files by copying the associated files from the project to a location in the product or to a specific location in the target file system, such as `/Library/Frameworks` or `/Library/Application Support`.
Compile Sources	Compiles source files into object files using a predefined tool, a tool you specify, or a build-rule script.

Build phase	Description
Run Script	Executes a shell script. You can use any scripting language whose scripts you can execute from the command line, such as AppleScript, Perl, Python, Ruby, and so on.
Link Binary With Libraries	Links object files with frameworks and libraries to produce a binary file.
Build ResourceManager Resources	Compiles `.r` files into resources that go into a separate resource file or into the executable's resource fork.
Compile AppleScripts	Compiles `.applescript` files and places the resulting `.scpt` files in the product's `Resources/Scripts` directory.

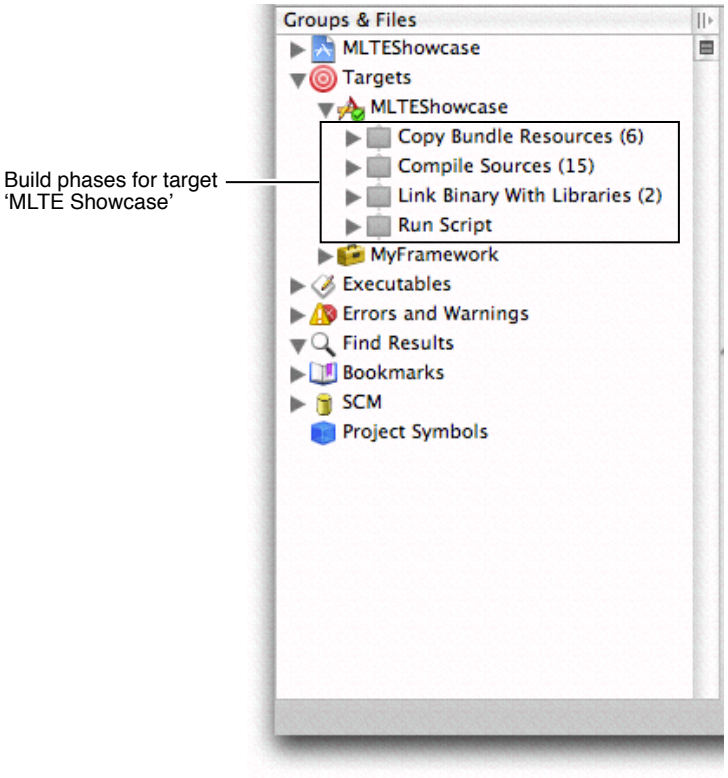
Table 2-2 lists the build phases and their possible inputs and outputs.

Table 2-2 Input files and output files of build phases

Build phase	Inputs	Outputs
Copy Headers	`.h` files	Copies in appropriate locations in the product.
Compile Sources	`.c`, `.m`, `.y`, `.l`, among other types of source files	`.o` files in target's `build` directory.
Link Binary With Libraries	`.framework`, `.dylib`, and the `.o` files in the target's build directory	Binary file, framework, or library in product destination directory.
Copy Bundle Resources	`.nib` files, `.strings` files, image files, and others	Copies in the product's `Resources` directory.
Build ResourceManager Resources	`.r`, and `.rsrc` files	Resource (`.rsrc`) files in the product's `Resources` directory.
Compile AppleScripts	`.applescript` files	AppleScript script (`.scpt`) files in the product's `Resources/Scripts` directory.

Each target has a set of build phases, separate from the build phases in other targets in your project. To view the build phases in a target, click the disclosure triangle next to the target in the Groups & Files list.

Figure 2-4 Viewing build phases



Build Phase Processing Order

The order in which the build system executes build phases is important because some build phases produce files that are part of the inputs of other build phases. Therefore, the former must be executed before the latter. In native targets, dependencies between build phases determine the order of execution. In Jam-based targets, you must ensure that the build phases within a target are ordered appropriately.

Build Phase Processing Order in Native Targets

In some cases, a build phase inherently includes the outputs of other build phases as its inputs. For example, the outputs (`.o` files) of the Compile Sources build phase are part of the inputs of the Link Binary With Libraries build phase. This makes the Compile Sources build phase an antecedent of the Link Binary With Libraries build phase. Therefore, the order in which build phases are executed in a target depends on their inputs and outputs.

Most build phases have their inputs and outputs defined implicitly. However, Run Script build phases may have neither. Here, the build system tries to run the associated script in the order specified within the target, but the actual point in the build process at which the script is run is undetermined. If you assign either input or output files to a Run Script build phase, the script's point of execution in the build process is determined by other targets having the build phase's outputs as their inputs, or the inputs of the build phase being the outputs of other build phases.

Figure 2-3 (page 36) illustrates this. Regardless of the order of the Compile Sources and the Link Binary With Libraries build phases in the target, the Compile Sources build phase is executed before the Link Binary With Libraries build phase. This is because the inputs to the Link Binary With Libraries build phase rely on output from the Compile Sources build phase.

In the Compile Sources build phase, the build system determines the order in which files are processed through the inputs and outputs of the target's build rules, in a similar way in which the order of build phases is determined.

Build Phase Processing Order in Jam-Based Targets

The order in which the build system executes the build phases in Jam-based targets is determined by the order of the build phases within a target. The order of the input files within the build phase determines the order in which the build system processes each file in that build phase.

You must make sure that build phases that produce files required by other build phases are listed first within the target. For example, the Compile Sources build phase must always be listed before the Link Binary With Libraries build phase.

Within the Compile Sources build phase, you must ensure that source files that generate derived files are placed above dependent files. For example, if a target has Yacc (`.y`) and Lex files (`.l`) files and processing the Lex files requires the C (`.c`) files generated from the Yacc files, within the build phase the Yacc files must be listed before the Lex files.

Reordering Build Phases

Typically, there is no need to change the order of build phases in a target; the default arrangement works for the majority of cases. If, however, you find that you need to change the position of a build phase—for example, when adding custom build phases to a Jam-based target—you can reorder a build phase by dragging the icon for the build phase to its new location in the target.

Adding and Deleting Build Phases

New targets—whether created through the New Target Assistant or as part of creating a project—already include a set of predefined build phases. The predefined build phases for a target vary, depending on the type of product created by the target. For example, a target that builds an application typically includes a Copy Bundle Resources build phase to copy resources into the application bundle; however, a target that builds a shell tool does not include this build phase. The predefined build phases work for most simple targets; however, you can accommodate more complex products and build steps by adding your own build phases.

To add a build phase:

1. (Optional) Select the target to which you want to add the build phase. Otherwise, Xcode adds the build phase to the active target.
2. Choose a build phase from:
    - Project > New Build Phase

    Table 2-1 (page 36) shows the available build phases.

Xcode adds the new build phase after the currently selected build phase, or, if no build phase is selected, adds it as the last build phase in the target. Several types of build phases can appear only once in a target; for example, there can be only one Compile Sources build phase. However, a target can contain multiple instances of the Copy Files and Run Script build phases.

To delete a build phase, select it in the Groups & Files list and press Delete, or choose Edit > Delete. Deleting a build phase does not delete the files in the build phase from the project or from the file system.

Adding Files to a Build Phase

When you add a file to a project, Xcode lets you choose whether to also add the file to any targets in the project. When you add files to a target this way, Xcode automatically assigns the files to build phases, based on each file's type.

To view the inputs to a particular build phase, you can:

- Select the build phase in the Groups & Files list. The files assigned to the build phase are displayed in the detail view.
- Open the build phase in the Groups & Files list.

Intermediate files—files generated by Xcode in other build phases—are not listed. Xcode handles these intermediate files automatically.

If the default file placement is not sufficient for your needs, you can drag the file or files from their current build phase to the new build phase. You can also drag existing files from the project group to the appropriate build phase in the Groups & Files list. To remove a file from a build phase, select the file and press Delete.

Compile Sources Build Phase

The Compile Sources build phase is one of the most easily customized build phases (the other one being the Run Script build phase). The reason is that this build phase must handle a wide variety of file types. Xcode is preconfigured to process several types of source files, but you may have to compile source files that Xcode doesn't know about.

The feature that makes the Compile Sources build phase so flexible is its support of *build rules*. They specify the tool or script the build system invokes to process files in Compile Sources and Build ResourceManager Resources build phases when building a product. For details on build rules, see Build Rules (page 47).

Copy Files Build Phase

A Copy Files build phase allows you to copy files and resources of any type to specific locations as part of the build process. It complements the build phases that copy specific types of files. An example is the Copy Headers build phase, which deals only with header files. You can have as many Copy Files build phases as you need in a target.

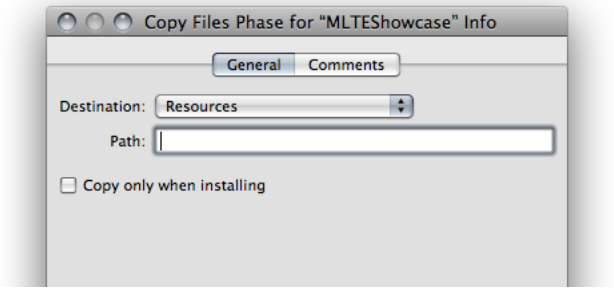
For example, using a Copy Files build phase, you can copy fonts to `/Library/Fonts`. Or, if you're developing a plug-in, a Copy Files build phase can copy the generated plug-in to the appropriate location. You can have as many Copy Files build phases in a target as you need.

To create a Copy Files build phase:

1. In the project window, open the target to which you want to add the build phase, and select the build phase after which to add the new build phase.
2. Choose Project > New Build Phase > New Copy Files Build Phase. Xcode adds the new Copy Files build phase after the build phase selected in the Groups & Files list.
3. Drag the files you want to copy from the Groups & Files list to the Copy Files build phase.

To configure the new Copy Files build phase, select it and open the Copy Files build phase editor, shown in Figure 2-5.

Figure 2-5 Copy Files build phase editor



Together, the Destination pop-up menu and the Path field specify the location to which Xcode copies the files in the Copy Files build phase. The Destination pop-up menu lets you choose from a number of standard locations. Table 2-3 shows the destination-location names you can choose in a Copy Files build phase for an application called MyApp and the resulting destination path. All the options, except Absolute Path and Products Directory, specify paths inside the generated bundle.

Table 2-3 Destination names and example destination paths of the Copy Files build phase

Destination name	Destination path
Absolute Path	Anywhere.
Wrapper	`MyApp.app`
Executables	`MyApp.app/Contents/MacOS`
Resources	`MyApp.app/Contents/Resources`
Java Resources	`MyApp.app/Contents/Resources/Java`
Frameworks	`MyApp.app/Contents/Frameworks`
Shared Frameworks	`MyApp.app/Contents/SharedFrameworks`
Shared Support	`MyApp.app/Contents/SharedSupport`
Plug-ins	`MyApp.app/Contents/PlugIns`
Products Directory	The directory in which Xcode places the project's built products. See Build Locations.

The Path field specifies the path, relative to the location specified in the Destination pop-up menu, to the target directory. If you choose Absolute Path from the menu, the Path field must contain the complete path to the destination directory for the files.

The "Copy only when installing" option lets you specify whether the build phase copies the files only in *install builds* of the product. That is, when using the `install` option of `xcodebuild` or when the Deployment Location (DEPLOYMENT_LOCATION) build setting is turned on. For more on `xcodebuild`, see Building with xcodebuild.

Run Script Build Phase

A Run Script build phase lets you execute any commands you need to perform a task. You can archive files using `tar`, send mail, write messages to a log file, execute AppleScript scripts, and so on. You can use any of the shell languages available in your system. And you can have any number of Run Script build phases in a target.

Before executing your script, Xcode assigns the values of most build settings to environment variables. In particular, it sets the environment variables listed in Table 2-4 (page 44), as well as any build settings that are defined for the active target. This includes build settings defined at the target layer, as well as any build settings inherited from lower layers. If you are building using `xcodebuild`, Xcode also passes any build settings defined on the command line to the environment.

Listing 2-1 shows how to print the value of a build setting by printing the value of the corresponding environment variable.

Listing 2-1 Example Run Script–build-phase script

```
echo "Building ${PRODUCT_NAME}"
```

This is the corresponding entry in the build transcript in the build results window:

```
Building Sketch
```


> **Note:** If you change the value of environment variables in one script, the change is not visible in another script. This is because Xcode does not recognize changes made to those environment variables during the script's execution and because shell scripts are independent of each other. For details on the build settings that are reflected in the script's environment variables, see Using Build Settings in Run Script Build Phases (page 46).

To perform operations on intermediate files, you can use several build settings, whose value—which Xcode sets before executing your script—you access through environment variables. A few of these build settings are listed in Table 2-4; the complete list of build settings that Xcode sets is much larger; see *Xcode Build Setting Reference* for the complete list of build settings.

Table 2-4 Environment variables accessible from a Run Script build phase

Environment variable	Description
`ACTION`	The action being performed on the current target, such as `build` or `clean`.
`BUILD_VARIANTS`	The variations—`debug`, `profile` or `normal`—that Xcode is creating for the product being built.
`PROJECT_NAME`	The name of the project containing the target that is being built.
`PRODUCT_NAME`	The name of the product being built, without any extension or suffix.
`TARGET_NAME`	The name of the target being built.
`TARGET_BUILD_DIR`	The location of the target being built.
`BUILT_PRODUCTS_DIR`	The directory that holds the products created by building the targets in a project.
`TEMP_FILES_DIR`	The directory that holds intermediate files for a specific target.
`DERIVED_FILES_DIR`	The directory that holds intermediate source files generated by the Compile Sources build phase.
`INSTALL_DIR`	The location of the installed product.

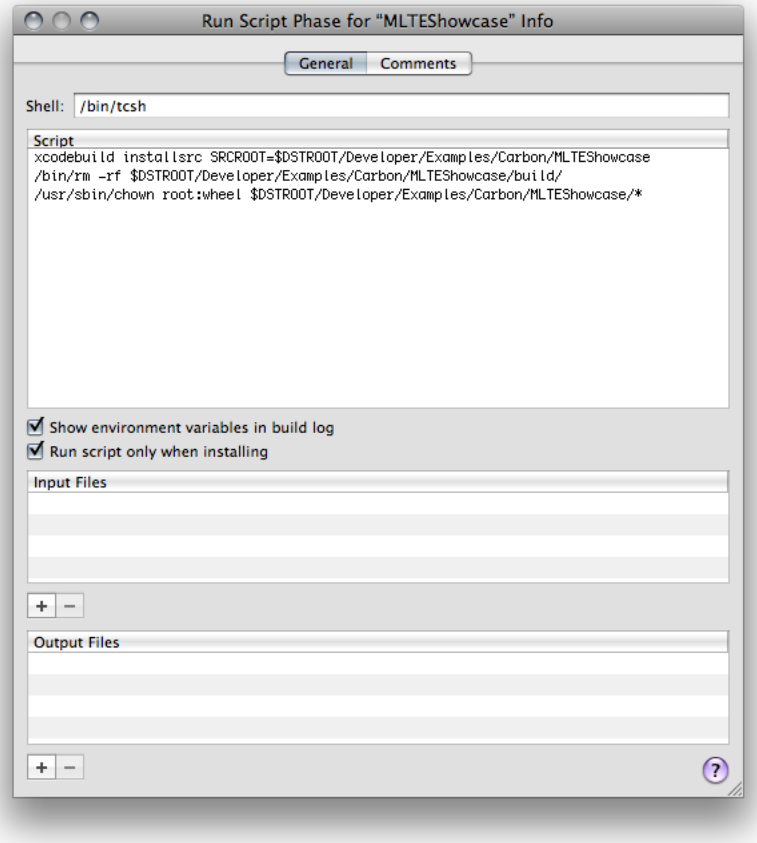
The script executes using the permissions of the logged-in user. Xcode runs the script with the initial working directory set to the project directory.

To create a Run Script build phase:

1. In the Groups & Files list, open the target to which you want to add the build phase, and select an existing build phase.
2. Choose Project > New Build Phase > New Shell Script Build Phase.

To configure the new Run Script build phase, open the Run Script build phase Info window, shown in Figure 2-6.

Figure 2-6 Run Script build phase Info window



The Run Script build phase editor contains these fields:

- **Shell.** Specifies the path to the appropriate shell.
- **Script.** Specifies the script itself as shown in Figure 2-6 (page 45). You can also enter a script that invokes a script stored elsewhere to, for example, avoid storing an often-changed script in the project file.

  A Run Script build phase with the following entries for Shell and Script executes a script called `MyScript.sh` stored in a directory called `Build_Scripts` in the project directory.

Shell:	`/bin/sh`

Script:	`$(SRCROOT)/Build_Scripts/MyScript.sh`

- **Run script only when installing.** Runs the script only during install builds, that is, when using the `install` option of `xcodebuild` or when the build settings Deployment Location (DEPLOYMENT_LOCATION) and Deployment Postprocessing (DEPLOYMENT_POSTPROCESSING) are on.
- **Show environment variables in build log.** Lists the build system's environment variables in the build log.
- **Input Files.** Specifies the names of the input files the script operates on.
- **Output Files.** Specifies the files that the script produces.

  In these two tables, file paths are interpreted relative to the project directory.

Xcode uses the input and output files to determine whether to run the script and to determine the order in which the script is executed. Specifying input and output files ensures that Xcode runs the script only when the modification date of any of the input files is later than the modification date of any of the output files (reducing the time it takes to build your product), and that the files the script produces are included in the dependency analysis the build system performs before building your product. If you provide no outputs, Xcode runs the script every time you build the target.

After the script terminates, Xcode uses the script's exit value to determine whether to continue the build. When the script exits with a nonzero exit value, Xcode stops the build. If you want to perform other actions—such as writing to a log file—in the event of a build failure, you must perform the appropriate action in the script.

Using Build Settings in Run Script Build Phases

As Xcode constructs the command-line invocations for the various tools it uses to build a product, it accesses the build settings available at the target level. Also, when it sets the environment variables for shell scripts in Run Script build phases, it resolves most build settings and sets environment variables with their values.

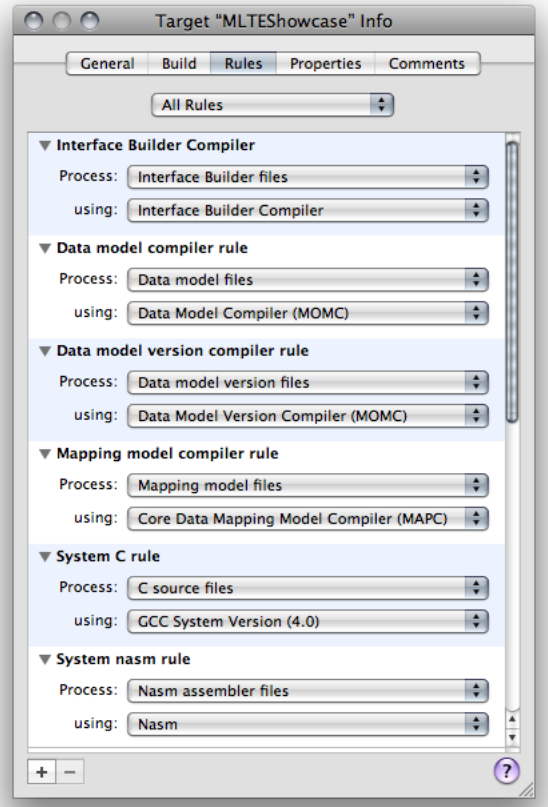
Note: Modifying the values of the environment variables that the build system sets for shell scripts in Run Script build phases has no effect on the value of the corresponding build setting. That is, shell scripts executed as part of the same build process after one in which you modify the environment variables get the unmodified values. This is the expected UNIX scoping behavior for environment variables.

Build Rules

Build rules specify how particular types of files are processed in Compile Sources build phases and specify which tool is used to process files in Build ResourceManager Resources build phases. For example, a build rule may indicate that all C source files be processed with the GCC compiler. Each build rule consists of a condition and an action. The condition determines whether a source file is processed with the associated action. Usually, the condition specifies a file type.

Xcode provides default build rules that process C-based files, assembly files, Rez files, and so on. You can add rules to process other types of file to each target. You can see the build rules in effect for a target in the Rules pane of the Target Info window, shown in Figure 2-7. Build rules are processed in the order they appear in the Rules pane.

Figure 2-7 Target editor: Rules pane



There are two types of build rules:

- **System build rules.** System build rules are predefined and unmodifiable, although you can override them. They include rules for processing C-based, Assembler, and Rez source files. See System Build Rules (page 49) for details. System rules are the same for all targets in a project.
- **Target-specific build rules.** Target-specific build rules are custom rules that you have defined for a particular target.

Target-specific build rules can specify files that the system build rules do not directly address, and can override the existing system build rules. For example, there's a system build rule for the processing of C-based source files, which means that `.c` and `.m` files are processed by the same build rule. You can, however, add a

target-specific build rule indicating that `.m` files be processed by a different compiler. In addition, instead of specifying a particular type of file, you can set the build rule's condition to a pattern that matches a set of files. See Creating a Build Rule (page 50).

You can view all build rules for a target, including available system build rules, by choosing All Rules from the pop-up menu at the top of the Rules pane in the target editor. To see only those build rules defined for the target, choose Rules Specific to Target.

A build rule's action typically specifies the tool or compiler to use when processing files that meet the given condition. But you can also specify a build-rule script. The default interpreter is `/bin/sh`. However, you can specify any script interpreter by entering `#!<interpreter_path>` as the first line of the script. When you use a build-rule script, you must specify the files the script produces as the build rule's output files. See Creating a Build Rule Script (page 50).

When processing a source file, Xcode evaluates the build rules from top to bottom and chooses the first one whose condition matches the source file being processed. Because custom build rules appear above the built-in system rules, the custom build rules can override the system build rules.

After a file is processed by a build rule, the output produced may be processed by another build rule when the second build rule takes as input files of the type the first build rule produces. For example, say you have a command-line tool that generates a Rez (`.r`) file from another type of file. You create a custom build rule that takes the specialized file as input an processes it using your command-line tool. Because the product of your custom build rule is a Rez file, the Rez file is then processed by the Rez system rule, which produces a `.rsrc` file.

System Build Rules

System build rules are predefined rules in all targets and are used to process several well-known file types. Table 2-5 lists the system build rules that Xcode provides.

Table 2-5 System build rules

Rule	Inputs	Outputs
Data Model Compiler	`.xcdatamodel`	`.mom`
C	`.c`, `.m`, `.cpp`, `.mm`	`.o`
Assembler	`.s`	`.o`
Yacc	`.y`	`.h`, `.c`
Lex	`.l`	`.c`

Rule	Inputs	Outputs
Rez	`.r`	`.rsrc`
MiG	`.defs`	`.c`

Creating a Build Rule

In addition to having system build rules, Xcode lets you define custom build rules on a per-target basis. Custom build rules allow you to change the way files of a particular type are processed or add support for types of files not directly addressed by the system rules.

To define a build rule's condition, choose a file type from the Process pop-up menu. You can also define rules that match arbitrary file names by choosing "Source files with names matching:" from the Process pop-up menu, and specifying a filename pattern in the text field that appears. This field accepts file name substitution (or globbing), accepting the same kinds of file patterns that you can use in a command shell editor. For example, to match all files whose names start with a capital letter and end with a `.def` suffix, you specify `[A–Z]*.def` as the pattern.

You define the build rule's action by choosing one of the available compilers from the "using" pop-up menu. Xcode provides a number of built-in compilers to choose from. Alternatively, you can define a custom script for processing files that meet the build rule's condition, as described in Creating a Build Rule Script (page 50).

When processing a source file in the target, Xcode evaluates the build rules from top to bottom and chooses the first one whose condition matches the source file being processed. For this reason, you should put the most specific build rules above the more general ones. For example, a rule that matches only C++ files should appear above a rule that matches all C-like files—that is, C, C++, Objective-C, and Objective-C++ files.

To reorder a custom rule, drag it from its background.

> **Note:** System rules cannot be reordered.

Creating a Build Rule Script

Instead of choosing one of Xcode's built-in compilers to process the files specified by a build rule's condition, you can create a custom script to process those files.

To create a custom script for a build rule, choose:

- "using" > "Custom script"

You can enter your script in the text field that appears, or you can store your script as a separate file in your project and invoke it from the text field using its project-relative path. In this case, you are actually defining a one-line build rule script that calls the script in your project.

In addition to defining the script, you also need to tell Xcode the paths of any output files that the script produces. Enter the path to each separate output file that is produced by the script in the "with output files" table below the script text field. For each file, create a row by clicking the plus (+) button. In this row, specify either the full (absolute) path or the project-relative path to the file. You can use any of the environment variables listed in Table 2-6 (page 51).

For example, suppose that you need to define a build rule to process files with the extension `.abcdef`. Also suppose that the build rule script produces a `.c` file for each `.abcdef` file and that the generated `.c` files are placed in the default directory for intermediate files. In this case, the output-files specification could look like this:

```
$(DERIVED_FILES_DIR)/$(INPUT_FILE_BASE).c
```

Note the use of parentheses in this example. The output-files specification uses the standard Xcode syntax for accessing build settings. Your custom script, however, must use standard shell syntax to access any environment variables corresponding to Xcode build settings, as described in Execution Environment for Build Rule Scripts (page 51).

The generated files are automatically fed back to the rule-processing engine. For example, continuing the `.abcdef` example rule, Xcode processes the `.c` files the script produces using the rule that processes `.c` files.

Execution Environment for Build Rule Scripts

Before executing a build-rule script, Xcode sets environment variables to reflect the values of some build settings used to build the product. See Using Build Settings in Run Script Build Phases (page 46) for details. In addition, Xcode sets a few additional environment variables (shown in Table 2-6) to values the script may need to access.

Table 2-6 Environment variables for build rule scripts

Variable	Description	Example
`DERIVED_FILES_DIR`	Path of the directory for derived (intermediate) files.	`/Users/me/project/build/Project.build/Target.build/DerivedSources`

Variable	Description	Example
INPUT_FILE_DIR	Directory containing the source file.	/Users/me/Project
INPUT_FILE_BASE	Base filename of the source file being processed.	source
INPUT_FILE_NAME	Filename of the source file being processed, including its extension.	source.cpp
INPUT_FILE_PATH	Path to the source file being processed.	/Users/me/Project/source.cpp
INPUT_FILE_REGION_-PATH_COMPONENT	Directory containing the source file being processed when it's localized. Otherwise, an empty string "".	en_US.lproj/
INPUT_FILE_SUFFIX	Suffix of the source file being processed, including the leading period.	.cpp
TARGET_BUILD_DIR	Path to the directory into which the target's products are being built.	/Users/me/Project/build/Development

For example, you can use these environment variables to access the current input file, as in the following shell script:

```
cat ${INPUT_FILE_NAME} > ${DERIVED_FILES_DIR}/${INPUT_FILE_BASE}.c
```

Note that the build rule script uses standard shell syntax to access the environment variables, rather than the Xcode syntax for referencing build settings.

Xcode runs build rule scripts with the current working directory set to the project directory. This behavior lets you access specific source files by using their project-relative paths.

Build Settings

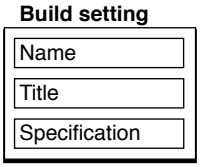
A *build setting* is a variable that contains information used to build a product. For each operation performed in the build process—such as compiling Objective-C source files—build settings control how that operation is performed. For example, the information in a build setting can specify which options Xcode passes to the tool—in this case, the compiler—used to perform that operation.

Build settings constitute the main method of customizing the build process. They represent variable aspects of the build process that Xcode consults as it builds a product.

This chapter explains how build settings are implemented and how you can take advantage of them to communicate with the build system.

Build Setting Overview

A build setting in Xcode has two parts: The name and the specification. The *build setting name* identifies the build setting and can be used within other settings; in that sense, it is similar to the names of environment variables in a command shell. The *build setting specification* is the information Xcode uses to determine the value of the build setting at build time. A build setting may also have a display name, or *build setting title*, which is used to display the build setting in the Xcode user interface.



The build system consults the value of build settings as it generates tool invocations. For example, to generate profiling code for all the source files compiled with GCC, you turn on the Generate Debug Symbols (`GCC_GENERATE_DEBUGGING_SYMBOLS`) build setting. When Xcode creates the `gcc` command-line invocation to compile a source file, it evaluates compiler build settings (such as Generate Debug Symbols) to construct the argument list.

Build setting values may come from a number of sources at build time. Xcode stores build setting specifications in dictionaries spread across several layers. Typically, you are most interested in the *target layer*; however, it is important to understand the various layers from which build setting values can be derived. See Build Setting Evaluation (page 58) for details.

Xcode has many built-in build settings that you can use to customize the build process. Furthermore, you can define your own build settings, which you can use to configure standard build settings across several targets or access within scripts in Run Script build phases to perform special tasks. To learn how to use each build setting, see *Xcode Build Setting Reference*.

In addition to build settings, you can set *per-file compiler flags* that the build system uses when creating tool invocations. With them you can change how a file is compiled without affecting any other files in the project. See Per-File Compiler Flags for details.

Build Setting Syntax

Build setting names start with a letter or underscore character; the remaining characters can be letters, underscore characters, or numbers. The Xcode application doesn't allow you to define build settings whose names don't follow this convention. Because build setting names are case sensitive, however, `project_name` is irrelevant to the build system and doesn't override the value of the `PROJECT_NAME` build setting.

Xcode displays titles for most of its predefined build settings in the Build pane of project and target editors. For example, the title of the `PRODUCT_NAME` build setting is Product Name. The `xcodebuild` tool doesn't use build setting titles.

The value of a build setting is determined by evaluating the build setting specification. A build setting specification can be a value such as a string, number, and so forth, or it can reference the value of other build settings.

To reference a build setting value in a build setting specification, use the name of the build setting surrounded by parentheses and prefixed by the dollar-sign character ($). For example, the specification of a build setting that refers to the value of the Product Name build setting could be similar to `The name of this target's product is $(PRODUCT_NAME)`.

> **Note:** When referring to build setting values in build setting specifications, you must use the build setting name, such as `PRODUCT_NAME`, instead of its title.

In addition to the build settings provided by Xcode, you can add *user-defined build settings* to a project. A user-defined build setting is one that is not defined by the build system. You can add them at the command-line, target, project and environment levels. You can reference user-defined build settings in build setting specifications, and scripts in Run Script build phases. User-defined build settings don't have a title.

The following list describes some circumstances in which you may need to add user-defined build settings to a project:

- To modify the value of build settings that are not displayed in the Xcode application

- To specify a value that can be used by several build settings in different layers (see Build Setting Evaluation (page 58) for details)
- To define a build setting you want to use in Run Script build phase scripts or a build-rule script

Conditional Build Settings

In general, build setting specifications apply to a target regardless of the SDK used to build it or the architectures for which the product is built. There are occasions, however, when you may need to specify a build setting value that must apply only when the product is built using a particular SDK, or when Xcode is generating a executable code for a particular architecture or for a particular variant of the product. *Conditional build setting definitions* allow you to add build setting values that apply only when one or more conditions are met (for example, the product is being built using the iOS Simulator SDK).

These are the conditions available in conditional build setting definitions:

- **Architecture**

  The architecture condition lets you create build setting definitions for particular architectures.

  This is the format of a build setting definition conditionalized on architecture:

```
<build_setting_name>[arch=<architecture_pattern>] =
<build_setting_specification>
```

  These are examples of architecture patterns:

```
armv6                   // ARM v6
i386                    // 32-bit Intel
x86_64                  // 64-bit Intel
*                       // Any architecture
```

  For example, to specify Other C Flags as `-dM` for the i386 architecture, you would use:

```
OTHER_CFLAGS[arch=i386] = -dM
```

- **SDK**

  The SDK condition lets you create build setting definitions for particular SDKs.

  This is the format of a build setting definition conditionalized on SDK:


```
<build_setting_name>[sdk=<sdk_pattern>] = <build_setting_specification>
```

You can set the SDK condition to match one or any release of an SDK. These are examples of SDK patterns:

```
iphoneos4.0           // The iPhone Device SDK 4.0
iphoneos*             // Any release of the iOS SDK
iphonesimulator4.0    // The iOS Simulator SDK 4.0
iphonesimulator*      // Any release of the iOS Simulator SDK
macosx*               // Any release of the Mac OS X SDK
*                     // Any SDK
```

- **Variant**

  Build variants specify the purpose of a product. These are the available build variants:

  - `normal`: The normal build variant produces a binaries that can be released to end users.
  - `profile`: The profile variant produces binaries to be used to profile program execution.
  - `debug`: The debug variant produces binaries you can use to debug your programs.

  You can use build setting variants in build setting definitions to tailor build aspects to particular binary variants.

  This is the format of a build setting definition conditionalized on variant:

```
<build_setting_name>[variant=<variant_name>] =
<build_setting_specification>
```

You can specify one or more conditions in a conditional build setting definition by concatenating the conditions enclosing each condition with square brackets or by separating the conditions with commas:

```
<build_setting_name>[sdk=<sdk_pattern>][arch=<arch_pattern>] =
<build_setting_specification>

<build_setting_name>[sdk=<sdk_pattern>,arch=<arch_pattern>] =
<build_setting_specification>
```

To define conditional build settings at the command line on in shell scripts, surround the definition with single or double quotation marks:

```
"OTHER_CFLAGS[arch=i386]=-dM"
```


The conditional build setting syntax provides a way to define build settings, not to access the conditions or values of the build settings at built time. At build time, referencing a build setting that has been conditionalized provides the conditional value of the build setting. For example, setting `OTHER_CFLAGS[arch=i386]` to `-dM`, results in `$(OTHER_CFLAGS)` returning `-dM` at build time when the active architecture is i386. You cannot use the condition patterns at build time to access the build setting conditions.

Note: You cannot use conditional build settings in Run Script build phases.

This section describes the literal syntax of conditional build setting definitions, used in build configuration files, environment variables, shell scripts, or `xcodebuild` invocations. To learn how to use the build settings editor to define conditional build setting definitions, see Editing Build Settings.

Note: The build settings editor doesn't show the variant condition, but you can specify build settings conditionalized on product variants in configuration files.

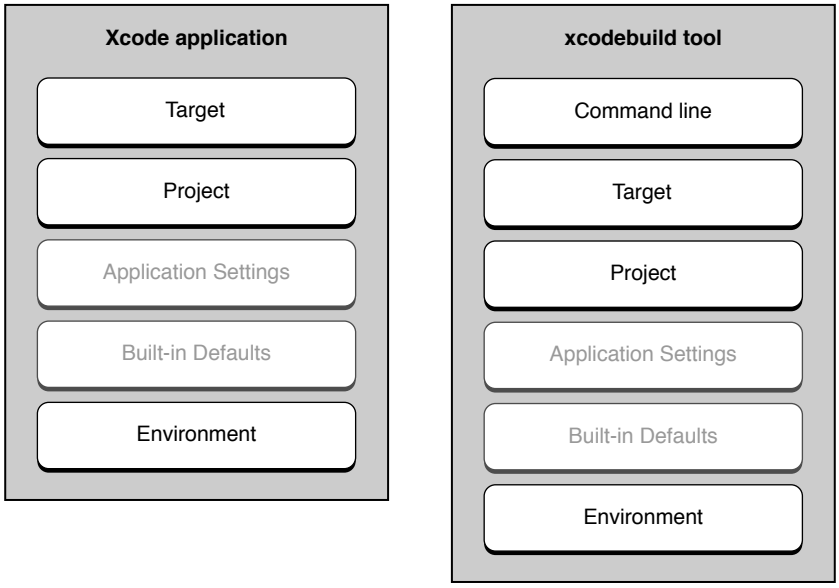
Build Setting Evaluation

To take advantage of build settings in your project, you must understand how they are evaluated when Xcode builds your product. By defining build settings at several build setting layers you can, for example, quickly change an aspect of a product when building from the command line, configure a product aspect only in one of a product's alternate "flavors" but not in all of them, or specify an aspect for all the projects that you work on.

At build time, Xcode evaluates each build setting individually. Figure 3-1 shows the layers in which build settings can be defined; the specifications of build settings defined in higher layers override the specifications of the same build settings at lower layers. Using this layered approach, you can define a build setting in terms of itself, in multiple layers. That is, you can include the value of a build setting as part of the specification of

the build setting in more than one layer. The value of the build setting at build time is a composite of the specifications of that build setting in all the layers where it's defined. For an example of how this process works, see Multilayer Build Setting Definitions (page 62).

Figure 3-1 Build setting layers



Note: The build setting layers may not all be available when you build a product. If you build from the Xcode application, the command-line layer is not used.

When the build system needs the value of a build setting, it starts at the highest build setting layer available to it and works down in the order shown in the following list:

1. **Command-line layer** (`xcodebuild` only). The command-line layer includes build settings defined in the `xcodebuild` invocation.

   The `xcodebuild` tool's command invocation represents the highest layer in which you can define build settings when building a product. The build settings defined in this layer are accessible only to `xcodebuild`.

   This layer gives you a way to customize builds performed from the command line without requiring access to the Xcode application.

> **Note:** In the *build settings editor* (the Build pane in project and target editors), Xcode provides data editors appropriate for particular build settings. For example, build settings with a Boolean value use a checkbox and those whose value is a list of paths use a list editor. In the command line, however, you don't have access to checkboxes or list editors when specifying the value of build settings. When specifying build settings at the command line, you should consult *Xcode Build Setting Reference* to find out the proper text-based format for the build settings' specifications.

2. **Target layer.** The target layer contains build settings defined in the active build configuration of the target being built.

   This is the main layer for specifying how a product is built. The build setting specifications in this layer override the ones defined at lower layers; this applies to both user-defined build settings and built-in build settings that have titles.

   Each target can define several named collections of build settings, called *build configurations*. This lets you produce different "flavors" of a target's product without having to create separate targets. When you build, you build a specific configuration of the target.

   > **Note:** A build configuration can inherit build setting definitions from a configuration file. For more information on how configuration files affect build setting precedence, see Build Configurations (page 70).

   Xcode evaluates build settings for a target independently from the build settings defined in other targets. This is true even for dependent and aggregate targets.

   For more information on targets, see Targets (page 10).

3. **Project layer.** The project layer contains build settings defined in the active build configuration of the current project—the project in which the current target is defined.

   Xcode and `xcodebuild` use the project layer to specify projectwide aspects, such as the location of the project itself. You can define build settings that you want all targets in the project to share at this layer. These build settings can be overridden by individual targets (at the target layer). When you have multiple targets with build settings that should be synchronized, the project layer can be a convenient way to do so.

   Projects, like targets, also define build configurations; for more information, see Build Configurations (page 70).

4. **Application settings layer.** The application build settings layer contains application-wide build settings defined for the current user.

The application settings layer contains per-user settings, such as any source trees defined by the current user. In Figure 3-1 (page 59), this layer appears dimmed because Xcode provides limited access to build settings at this layer. Xcode Preferences > Building provides an interface for changing the `OBJROOT` and `SYMROOT` build settings at this layer.

5. **Built-in defaults.** The built-in defaults layer contains a number of default values for build settings that are built into Xcode. Most of the build settings defined at this layer are build settings that specify required attributes of Mac OS X products.

6. **Environment layer.** Build settings defined in the Xcode application environment or the shell from which `xcodebuild` is launched.

   The environment layer is composed of environment variables that correspond to build setting names. You can use it to configure build settings that must apply to more than one project. This layer, however, cannot access build settings configured in any other layer. For example, defining an environment variable named `MY_PRODUCT_NAME` as `My Company $(PRODUCT_NAME)` results in an undefined-variable error, unless you also define an environment variable named `PRODUCT_NAME`.

   You must follow the syntax described in Build Setting Syntax (page 55) when defining the environment variables in your session.

   > **Note:** If you associate additional compiler flags with a file, as described in Per-File Compiler Flags, those flags are always used when the file is processed as part of a build. They cannot be overridden in any of the build setting layers.

As soon as the build system finds a definition for the build setting it's looking for, it stops traversing the build setting layers. However, if the build setting specification found includes references to other build settings, it resolves them. This starts the traversing process again, as many times as necessary to compute the value of the original build setting.

If a build setting refers to itself (that is, the build setting specification includes a reference to the build setting being evaluated), the build system resolves the reference starting at the subsequent build setting layer. The following sections provide examples of this process.

Note: Xcode uses *bold text* in the build setting editors for projects and targets to indicate build settings specified at the current level—that is, at the level that you are currently viewing, either the project or the target. Build settings that are not in bold text are specified at lower layers.

Multilayer Build Setting Definitions

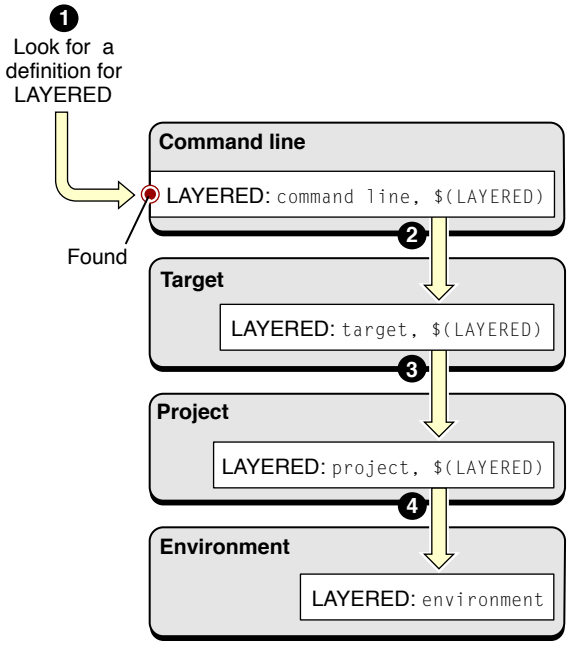
Imagine that, to build a product, you need to define a single build setting at every build setting layer. This is a very unlikely case to be sure, but one that illustrates how the process works. Table 3-1 shows an example configuration for the `LAYERED` build setting throughout the build setting layers. This example assumes that the product is built from the command-line using `xcodebuild`:

Table 3-1 Configuration of the `LAYERED` build setting

Build setting layer	Build setting specification
Command line	`command line, $(LAYERED)`
Target	`target, $(LAYERED)`
Project	`project, $(LAYERED)`
Environment	`environment`

Figure 3-2 shows how the build system would evaluate the LAYERED build setting when building using `xcodebuild`.

Figure 3-2 Evaluation of the LAYERED build setting



To evaluate the LAYERED build setting, the build system does the following:

1. Looks for a definition for LAYERED in the command-line layer (the highest available to `xcodebuild`). It finds the specification `command line, $(LAYERED)`.
2. Resolves `$(LAYERED)` starting at the next layer down, the target layer. At this layer, it obtains the specification `target, $(LAYERED)`. Because this specification also references the value of the LAYERED build setting, the build system continues to look in lower layers for the build setting specification.
3. Resolves `$(LAYERED)` starting at the project layer, obtaining `project, $(LAYERED)`.
4. Resolves `$(LAYERED)` starting at the environment layer, obtaining `environment`.

   The evaluation of LAYERED stops here because there are no build setting layers below the environment layer. When all the references are resolved, the final value of the build setting is computed as `command line, target, project, environment`.

This example uses `$(LAYERED)` to access the value of the same build setting at a lower layer. However, you can get the same result in the example by replacing `$(LAYERED)` with `$(value)` in any of the build setting specifications at the command-line, target, or project layers.

The process of evaluating a build setting specification that references itself repeats recursively until the build system reaches the environment layer or until the build system finds a build setting specification that does not reference its own value.

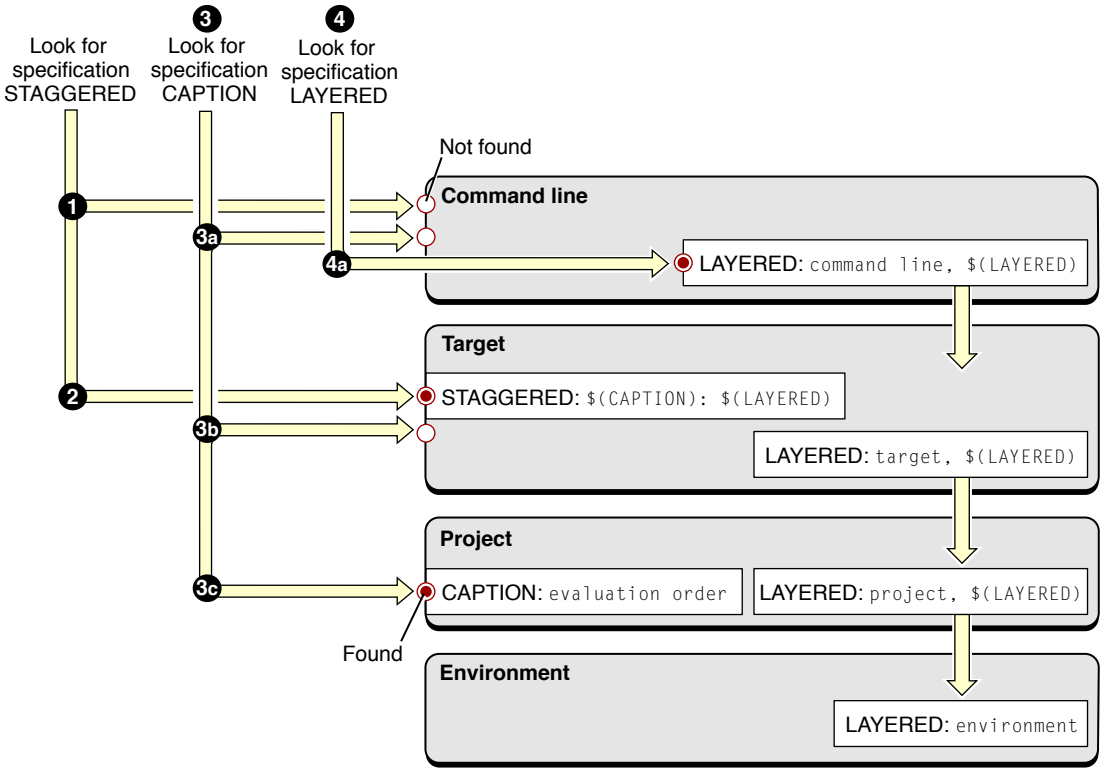
Build Setting References

The build system gives you a great deal of flexibility when defining build settings. The following example illustrates how the build system evaluates the STAGGERED build setting, which is defined in the target layer and references the values of several other build settings.

The value of the STAGGERED build setting is composed of data and a caption for the data, with the caption shown first and both elements separated by a colon (:) and a space. The specification for STAGGERED contains references to two other build settings: LAYERED (the data, explained in Multilayer Build Setting Definitions (page 62)) and CAPTION (the caption for the data). It also contains static elements (the colon and space characters).

Figure 3-3 shows how the build system evaluates the STAGGERED build setting.

Figure 3-3 Evaluation of the STAGGERED build setting



These are the steps the build system takes to evaluate the STAGGERED build setting:

1. Look for a definition of STAGGERED in the command-line layer. None is found.

2. Look for a definition of `STAGGERED` in the target layer.

   The build system finds a definition of `STAGGERED`: `STAGGERED = $(CAPTION): $(LAYERED)`. Here is where the evaluation of the `STAGGERED` build setting begins.

3. Resolve `$(CAPTION)` starting at the command-line layer:
   a. Look for a definition of `CAPTION` in the command-line layer. None is found.
   b. Look for a definition of `CAPTION` in the target layer. None is found.
   c. Look for a definition of `CAPTION` in the project layer. The build system finds the specification `evaluation order`.

      The evaluation of `CAPTION` stops here because there are no references to other build settings. The final value of the `CAPTION` build setting is `evaluation order`.

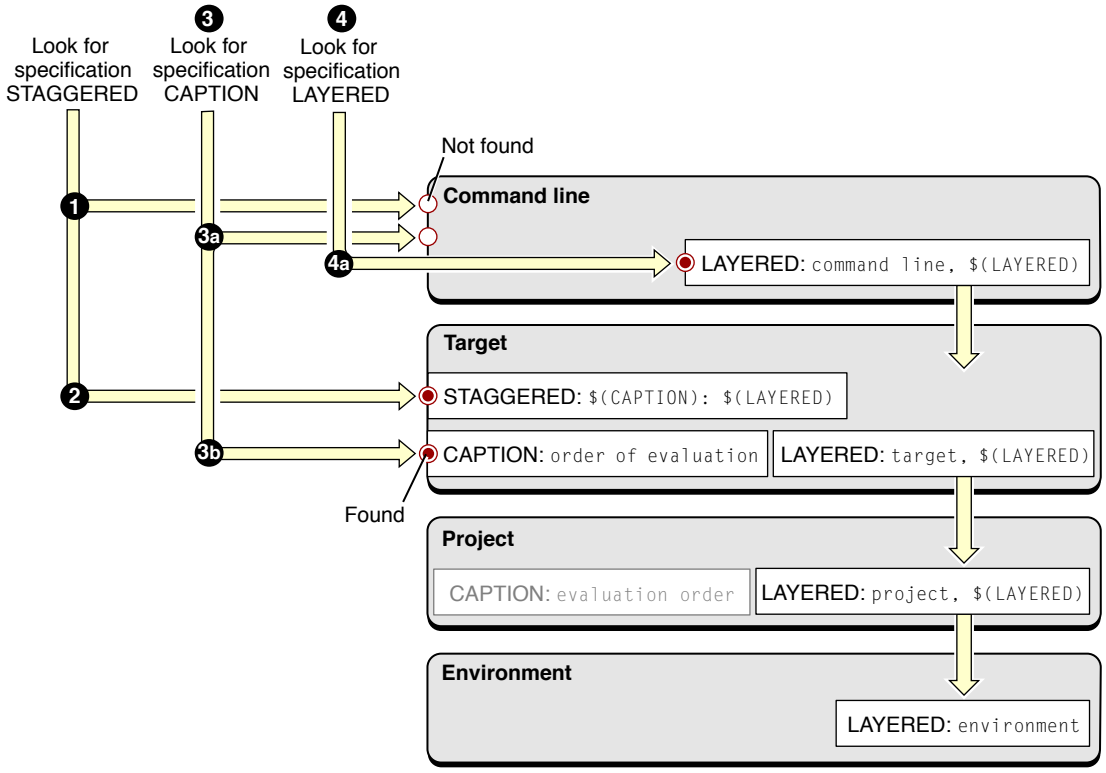
4. Resolve `$(LAYERED)` starting at the command-line layer.
   a. Look for a definition of `LAYERED` in the command-line layer. The build system finds the specification `command line, $(LAYERED)`. The build system resolves the specification for `LAYERED` at this build setting layer as described in the previous section. The final value of the `LAYERED` build setting is `command line, target, project, environment`.

5. Get the final value for the `STAGGERED` build setting by replacing the two references in the build setting's specification with their values: `evaluation order: command line, target, project, environment`.

Knowing the precedence that the build system uses when evaluating build settings makes it easy to determine where to configure build settings to tailor the build process for special situations. Following the `STAGGERED` example, imagine you want to override the value of the `CAPTION` build setting for a particular target. All you would have to do is configure the `CAPTION` build setting in that target with the appropriate value.

Figure 3-4 shows the effects of overriding CAPTION in the target layer.

Figure 3-4 Evaluation of the STAGGERED build setting with CAPTION overridden in the target layer



These are the steps the build system takes to evaluate the STAGGERED build setting after overriding CAPTION in the target build setting layer:

1. Look for a definition of STAGGERED in the command-line layer. None is found.
2. Look for a definition of STAGGERED in the target layer. The build system finds `STAGGERED = $(CAPTION): $(LAYERED)`.
3. Resolve `$(CAPTION)` starting at the command-line layer:
    a. Look for a definition of CAPTION in the command-line layer. None is found.
    b. Look for a definition of CAPTION in the target layer. The build system finds `order of evaluation`.

    The evaluation of CAPTION stops here because there are no references to other build settings in its specification. Even though CAPTION is configured in the project layer, that specification has been overridden in the target layer; therefore, it's ignored. The final value of CAPTION in this example is `order of evaluation`.
4. Look for a definition of LAYERED in the command-line layer. The build system finds `LAYERED = command line, $(LAYERED)`.

a. Resolve the specification for `LAYERED` at this build setting layer as described earlier in this section, obtaining `command line, target, project, environment`.

5. Get the final value for the `STAGGERED` build setting by replacing the two references in the build setting's specification with their values: `order of evaluation: command line, target, project, environment`.

Build Setting Troubleshooting

As you work on a project, you may need to determine where and how a build setting is defined. Because build settings can be identified by their name and their title (see Build Setting Syntax (page 55) for details). Depending on where a build setting is defined—in the `xcodebuild` invocation, in an environment variable, or in the Xcode application's user interface—you may need to map between a build setting name and its corresponding build setting title.

This section provides tips on how to troubleshoot build setting problems you may encounter.

Finding Build Setting Definitions

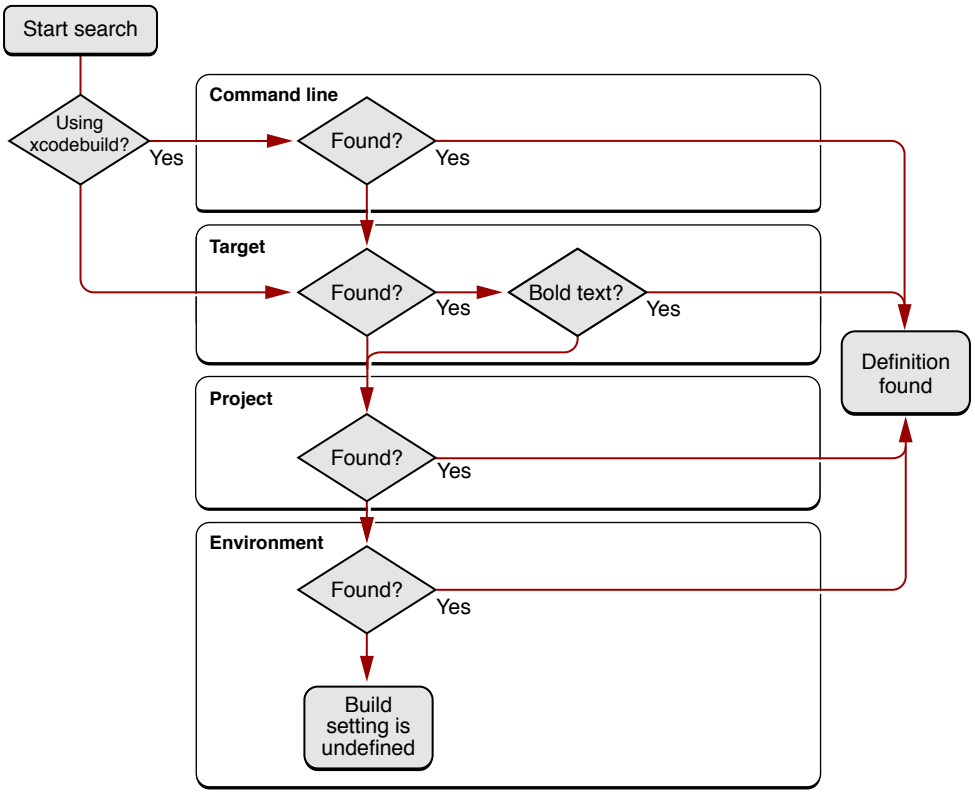
During the development process you may need to find the top definition of a build setting in order to change its value during a build. That is, you want to find the specification that overrides all other specifications throughout the build setting layers (see Build Setting Evaluation (page 58) for details). This section showcases a technique you can use to quickly locate the top definition of a build setting.

Look for the top definition of a build setting in these places:

- The `xcodebuild` invocation (when building using `xcodebuild`)
- The active configuration of the target you're interested in
- The active configuration of the project
- The output of the `env` command

Figure 3-5 shows the steps you would take to find the definition of a build setting when you know the build setting's name, title, or its specification.

Figure 3-5 Finding build setting definitions



1. If you're building using `xcodebuild`, look for the build setting name (or its specification) in the tool's invocation. If the build setting is part of the invocation, you found the definition.
2. Look for the build setting in the target you're interested in. You can locate a particular build setting by choosing All Settings from the Show pop-up menu of the build settings editor and entering the name, the title, or the specification (shown in the Value column) of the build setting you're looking for. You can look in all configurations of the target at once by choosing All Configurations from the Configuration pop-up menu:
    a. If the build setting is displayed in bold text, it means that it's defined in the target. This is the top definition of the build setting. If the build setting's definition is displayed as "Multiple values," the build setting has a different definition in the various configurations of the target. Select a specific configuration from the Configuration pop-up menu to see its definition of the build setting. Choose Active Configuration to see the definition in the active build configuration.
    b. If the build setting is displayed in nonbold text, the build setting is defined in the project layer or the environment layer (see Build Setting Evaluation (page 58) for details).

3. Look for the build setting in the active configuration at the project level.
4. Look for the build setting in the build environment (the definitions of all the environment variables the Xcode application or `xcodebuild` have access to during the build process).
    a. Add a Run Script build phase to the target you're interested in and make it the first build phase to make it easier to locate the script's output.
    b. Add an invocation to the `env` command to the build phase's shell script.
    c. If you're building using the Xcode application, open the Build Results window, reveal the build log pane, and build the product. If you're building using `xcodebuild`, invoke the tool from your shell as you normally would. See Building Products for information on how to build in Xcode.
    d. Look at the build log (the detailed log in the Build Results window if using the Xcode application). The build environment is listed after the group of `setenv` invocations that set environment variables that reflect most of the build setting values for the current build. Search that group for the name of the build setting you're interested in. If the build setting is not in that group, the build setting is not defined for the target you are investigating.

       If the build setting is defined in the `setenv` group and the `env` group, you can override its value only in the target layer or above. If the build setting is defined only in the `env` group, look for the build setting definition among the environment variables defined in the user-configuration files for the logged-in user. If you find the build setting there, change its specification, log out, and log in.

Build Configurations

Build configurations are named collections of build settings that allow you to build one or more products in a project in different ways for different situations. For example, you can define separate debug and release build configurations. You can then use these build configurations to build debug and release versions of your product, without creating separate targets. Build configurations are a flexible tool for quickly "tweaking" your product or for saving different groups of build settings to apply to a target depending on the current circumstances.

This chapter provides a general explanation of build configurations, describes the predefined build configurations you get when you create a target or project in Xcode, and explains how to modify and define your own build configurations.

Build Configuration Overview

Build configurations allow you to build two or more "flavors" of a product without having to create separate targets for each product flavor. When you build a target, Xcode uses the build settings defined by that target for the current build configuration.

A common use of build configurations is to build a given target differently to create debug and release versions of a product. There are a handful of build settings—for example, optimization settings for the compiler—whose values are different, depending upon whether you are building a product for debugging purposes or to release to customers. In each situation, the target you build is identical in every other way—files, build phases, and build rules—because the product you want to generate for each is essentially the same. The only difference is in how the source files of the target are processed to build that product.

If you were to create two targets—one for debugging and one for the final build—you would have to remember to keep both targets in sync. When you added a file to one target, you would have to remember to add it to the other, and so forth. With a build configuration, you can build using a different group of build settings without changing the target's build phases or build rules. It is much simpler to create build configurations containing the build settings whose values you want to change, and then build based on the appropriate configuration.

The project defines the list of build configurations; all targets in a project share the same namespace for build configurations. The actual definition of any given build configuration—the particular build setting definitions in that configuration—is specific to each target. You can also define project-level configurations. As described

in Build Setting Evaluation (page 58), build settings defined at the target level take precedence over any values assigned to those build settings at the project level. This gives you a great deal of flexibility. For example, you can use project-level build configurations to define build settings that apply to all or most targets in the project. You can then modify or override these build settings for individual targets at the target layer.

The *active build configuration* is the build configuration Xcode uses when building the active target and any targets it depends upon. When you initiate a build, Xcode builds the active target and any targets it depends upon. For each target, Xcode applies the build settings defined by that target for the active build configuration, as well as any build settings defined for the active build configuration at the project level. See Setting the Active Target and Build Configuration for a description of how to change the active build configuration.

For each target, Xcode evaluates the build settings defined by that target for the active build configuration, as well as any build settings defined for the active build configuration at the project level. As described in Managing Build Configurations (page 72), the list of build configurations is the same for all targets within a project. When building targets in a referenced project, Xcode uses the build settings defined in the corresponding build configuration of that project.

If your project includes cross-project references, it is possible that the referenced project defines a different set of build configurations from those of the current project; the active build configuration may not exist in the referenced project. In this case, Xcode falls back on the default configuration for the referenced project and its targets.

For each target and project involved in a build, here is how the build configuration is selected when building from the Xcode application.

1. Xcode checks for a build configuration with the same name as the active build configuration. If this configuration is found, Xcode uses it.
2. If no matching build configuration is found, Xcode then uses the default build configuration.

The process of selecting build configurations is similar when building from the command line with `xcodebuild`:

1. If a specific build configuration is passed as a command-line argument to `xcodebuild` , `xcodebuild` first looks for a build configuration with that name in the project. If that configuration is found, Xcode uses it.
2. If no build configuration is specified on the command line or if the project does not define the specified build configuration, `xcodebuild` uses the default configuration.

New Xcode projects contain two predefined build configurations, Debug and Release. You can edit those build configurations or define new build configurations of your own.

In addition to the predefined Debug and Release build configurations, you may need to define a special build configuration to satisfy particular needs, such as configuring an environment to measure the performance of your application. In such a build configuration you may define build settings that add a library or framework that gathers and logs performance-related information. You may also need to target your application to specific Mac OS X versions. In that case, you may need to build your product slightly differently for each version. For example, you can have build configurations named Mac OS X 10.3 and Mac OS X 10.4 that build a product tailored for Mac OS X v10.3 and Mac OS X v10.4, respectively, by setting Mac OS X Deployment target (`MACOSX_DEPLOYMENT_TARGET`), and any other build settings necessary, to the appropriate values.

Predefined Build Configurations

A debug build of a product may include debugging information to assist you in fixing bugs. However, this extra information can consume valuable space in a user's system. A debug build should contain only the code necessary to run the application.

Some build settings tell Xcode to add debugging information to an executable or specify whether to optimize its execution speed. Other build settings turn on features such as Fix and Continue, which are useful only during debugging.

All Xcode project and target templates include two build configurations, the Debug build configuration and the Release build configuration. By default, the Debug build configuration turns on Fix and Continue, and debug-symbol generation, among others, while turning off code optimization. The Release build configuration turns off Fix and Continue. The code-optimization level is set to its highest by default, through the Optimization Level (`GCC_OPTIMIZATION_LEVEL`) build setting. Note that the Release build configuration doesn't turn on Deployment Location (`DEPLOYMENT_LOCATION`); therefore, the product is not copied to the location where it would be installed on a user's system. It also doesn't turn on Deployment Postprocessing (`DEPLOYMENT_POSTPROCESSING`), which specifies whether to strip binaries and whether to set their permissions to standard values.

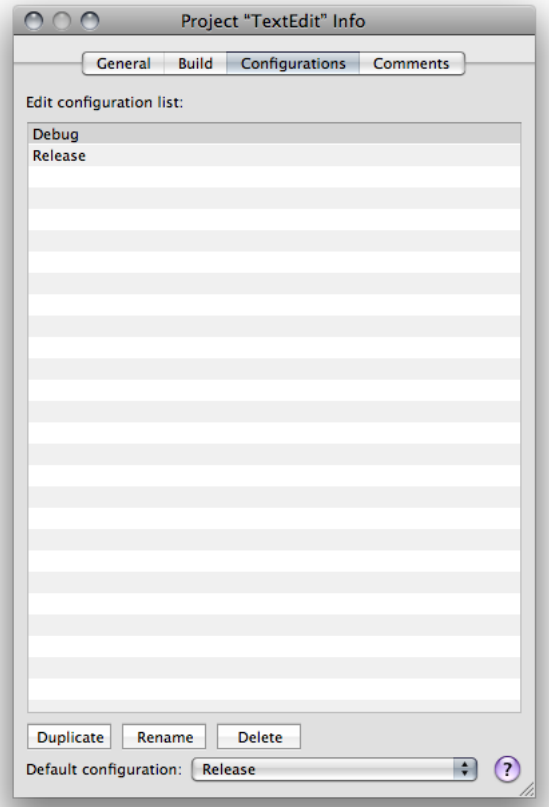
Managing Build Configurations

Generally, the predefined build configurations described in Predefined Build Configurations (page 72) are enough for most people. You can use them as they are or add additional build settings to them. For example, you can specify that the Debug build configuration for a target or project display more compiler warnings.

If you're thinking about creating a target, consider whether it might be best to create a build configuration instead. In general, create a target to create a product, and create a build configuration to modify how that product is built. If you're creating targets that differ only in their build settings, consider creating one target with several build configurations. If you need to create targets that differ in other ways—such as build phases—you need to create separate targets.

To see the list of build configurations defined for a project, open the project editor, and display the Configurations pane, shown in Figure 4-1. The configuration list displays all of the build configurations defined in the current project. You can edit the list of configurations as described in Managing Build Configurations (page 72).

Figure 4-1 Project editor: Configurations pane



These are the components of the Configuration pane of the project editor:

- **Configuration list.** List of the project's build configurations.
- **Duplicate button.** Duplicates the selected build configuration throughout the project. In this way, if you have a number of build settings common to all configurations in a target, you do not have to set them all up again.

- **Rename button.** Renames the selected build configuration name. throughout the project.
- **Delete button.** Deletes the selected build configuration from the project.
- **Default configuration pop-up menu.** Specifies which of the build configuration is the default build configuration.

  the *default build configuration* is used when the project being built does not have a definition for the active build configuration, as described in Build Configuration Overview (page 70).

Editing Build Configurations

As you've seen, you often want to vary the value of a build setting depending on the circumstances of the build. Xcode lets you edit individual build configurations to define build settings specific to that particular build variation. You typically also have some build settings that remain the same regardless of the current build configuration. Rather than editing each build configuration in the target or project separately, you can define the same build setting for all build configurations by editing all of a target or project's build configurations at once. Both approaches are described next.

- To edit a particular build configuration, choose that build configuration from the Configuration pop-up menu in the project or target build settings editor, and make your changes. See Editing Build Settings to learn how to use the build settings editor.
- To edit build settings for the active build configuration, choose Active Configuration from the Configuration pop-up menu.
- To edit build settings for all build configurations in a target or project, choose All Configurations from the Configuration pop-up menu. Build settings that have the same value for all configurations in the target or project display that value. Build settings that have different values for different build configurations display the string “Multiple values” or a checkbox with a dash. You can still edit the value for these build settings; doing so changes the definition of the build setting across all of the available build configurations to have the same value. To edit the definition of a build configuration in multiple targets at once, select those targets and open the target editor.

Build Configuration Files

When you work on large software endeavors with several projects and targets, many of which sharing a number of common build setting definitions, it can be time-consuming and repetitive to define the same build settings over and over again. For this reason, Xcode allows you to base a build configuration on *build configuration file*. A *build configuration file* (or *configuration file* for short) contains a list of build setting definitions. Using build configuration files, you can easily share a set of build setting definitions among all the individuals working on your team, or quickly configure targets and projects with common build setting definitions.

When you base a target or project's build configuration on a build configuration file, that build configuration automatically inherits the build setting definitions in that configuration file (and the configuration files it includes), as well as any subsequent changes to the configuration file (or the configuration files it includes). If you then modify the value of any of those build settings in the target or project, the new value is used instead of the value in the configuration file.

For example, if you have a standard list of debugging build settings that you use to build debug versions of all your products, you can define a configuration file—say `MyDebugSettings.xcconfig` —that contains these build settings. You can then base the Debug build configurations for all your targets on this file. Each Debug configuration inherits the build settings in the `MyDebugSettings.xcconfig` file. You can then modify any of these build settings for individual targets or projects, as necessary.

Creating a Configuration File

A configuration file is simply a plain text file with a list of build setting names and assignments, one per line. (You may also add comments using the `//` comment marker.) You can use build settings recognized by the Xcode build system or define your own custom build settings in a configuration file. For example, to specify that profiling code be generated for all build configurations based on a particular file, add the following to the file:

```
GENERATE_PROFILING_CODE = YES                // Turn on profiling code
```

Note that you must use the name of the build setting, not the build setting title. If the build setting can contain a list of values—such as `OTHER_CFLAGS`—separate each value with a space.

This format is the same format that Xcode copies to the clipboard when you copy and paste build settings from the build setting list. You can quickly create a configuration file by selecting the build settings you want to include, copying them, and pasting them into a new text file. You can then make any necessary modifications to the build setting definitions. Xcode also includes a configuration file template, which you can use to create a configuration file.

To create a configuration file:

1. Choose File > New File.
2. In the New File Assistant, select Xcode > Configuration Settings File.

Configuration files must have the extension `.xcconfig` to be recognized by Xcode.

> **Note:** In the build settings editor, Xcode provides data editors appropriate for particular build settings. For example, build settings with a Boolean value use a checkbox and those whose value is a list of paths use a list editor. In configuration files, however, you don't have access to checkboxes or list editors when specifying the value of build setting. As described earlier, you can specify the value of such build settings in the build settings editor and copy the build setting from the editor to the configuration file. When editing the values directly in the file, you should consult Build Setting Syntax (page 55) and *Xcode Build Setting Reference* to find out the proper text-based format for the build settings' specifications.

A configuration file may refer to one or more additional configuration files using the `#include` directive, as shown in Listing 4-1. You can use relative or absolute paths to specify the location of the configuration file to include. The group of configuration files joined together by `#include` directives is known as a *configuration unit*. A single configuration file that doesn't include other configuration files is also a configuration unit.

Listing 4-1 Referring to other configuration files

```
// File: MyConfig.xcconfig
#include "/Network/ProductDevelopmentSpecs/TeamConfig"
ARCHS = ppc i386
#include "/Network/ProductPackagingSpecs/ProductConfig"
```

Xcode analyses build setting definitions in the order specified in the configuration unit. For example, the MyConfig configuration file in Listing 4-1 includes two other configuration files: TeamConfig and ProductConfig. Xcode processes the build setting assignments in the configuration unit in this order:

Build setting definitions in TeamConfig

Build setting definitions in MyConfig

Build setting definitions in ProductConfig

When a configuration unit contains more than one definition for a particular build setting, Xcode uses the last definition in the unit. Keep in mind that configuration files do not have access to build setting definitions made in configuration files they include. That is, you cannot modify the definition made in an included configuration file; you can only replace it.

Basing Build Configurations on Configuration Files

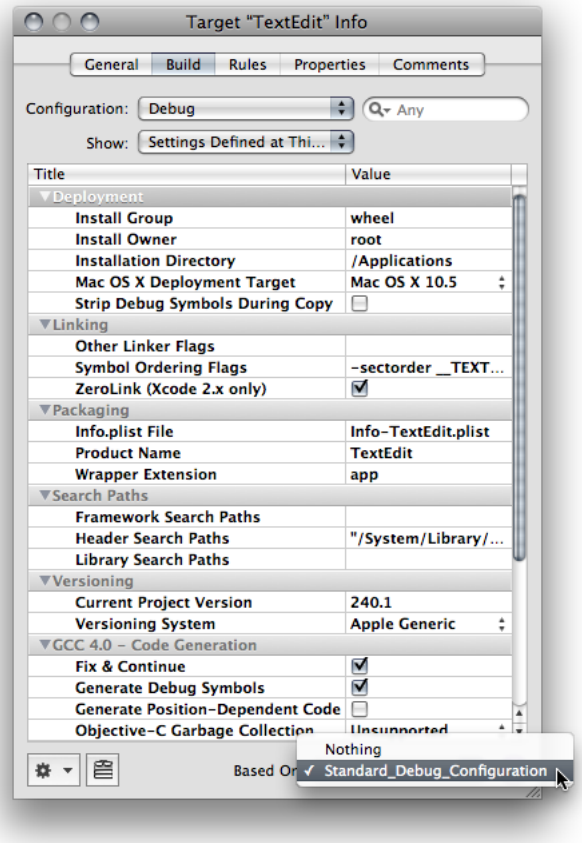
To base a build configuration on a configuration file:

1. Add the configuration file to your project, as described in Managing Files and Folders in a Project.

   Note: Do not add the configuration file to the target when you add it to the project.

2. In the build settings editor, choose the build configuration you want to modify from the Configuration menu.
3. Choose the configuration file from the Based On pop-up menu, below the build settings table, as shown in Figure 4-2. This menu displays all the build configuration files in the project.

Figure 4-2 Build settings editor: Choosing a configuration file



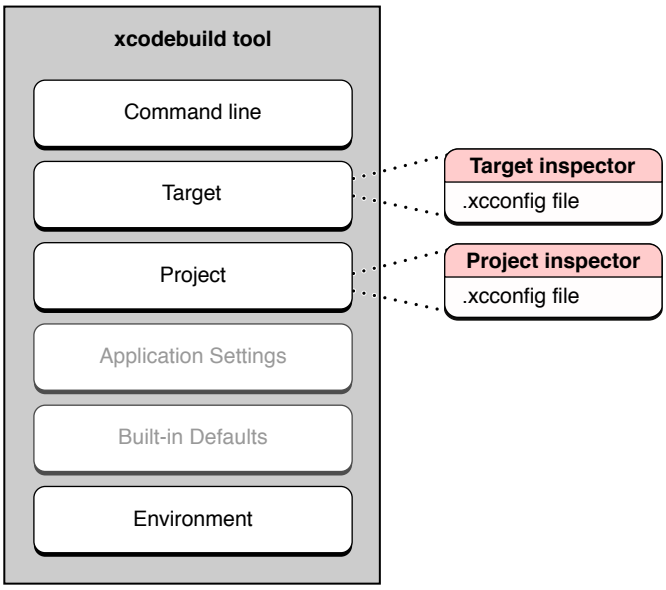
Your project can contain any number of configuration files; however, each individual build configuration can be based on only one configuration file.

Build Setting Evaluation and Configuration Files

Build settings are defined in a number of different layers, with definitions in higher layers overriding definitions in lower layers.

Of course, build settings at the target and project layers can be defined in the Xcode user interface or inherited from build configuration files. Figure 4-3 shows the build setting layers and their relationship to the build settings defined in configuration files.

Figure 4-3 Build setting layers and configuration files



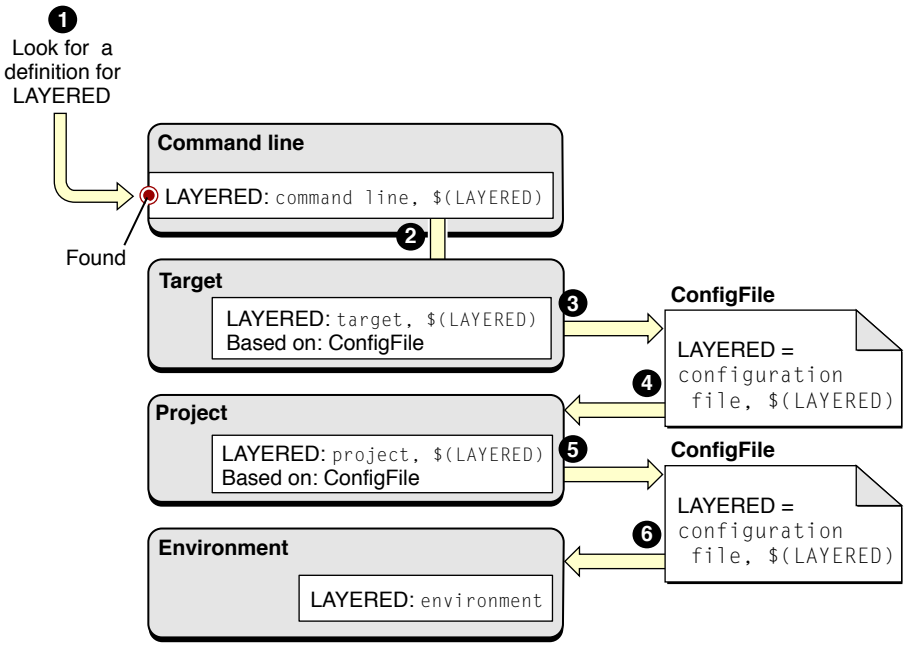
Note: Figure 4-3 shows the build setting layers available when building from the command-line with `xcodebuild`. If you are building from the Xcode application, the command-line layer is not available to you. However, all other layers are available.

When you use configuration files, the target and project layers themselves are each divided into two separate build setting layers. These are:

1. Build setting definitions listed in the build settings editor. Build settings configured in the Xcode user interface override build settings defined in the configuration file for the same build configuration of the target or project. This lets you customize the build for one or more targets or projects that otherwise share a common set of build settings.
2. Build setting definitions in the configuration file on which the active build configuration for the target or project is based. You can use configuration files to share build setting definitions across one or more build configurations in multiple targets or projects.

To understand how to use configuration files effectively, you must understand how configuration files affect the evaluation of build settings. The following example builds on the example used in Multilayer Build Setting Definitions (page 62) to show how Xcode evaluates build settings in a project that uses configuration files. Figure 4-4 shows how the build system would evaluate the `LAYERED` build setting when using `xcodebuild`.

Figure 4-4 Evaluation of the `LAYERED` build setting using configuration files



To evaluate the `LAYERED` build setting, Xcode does the following:

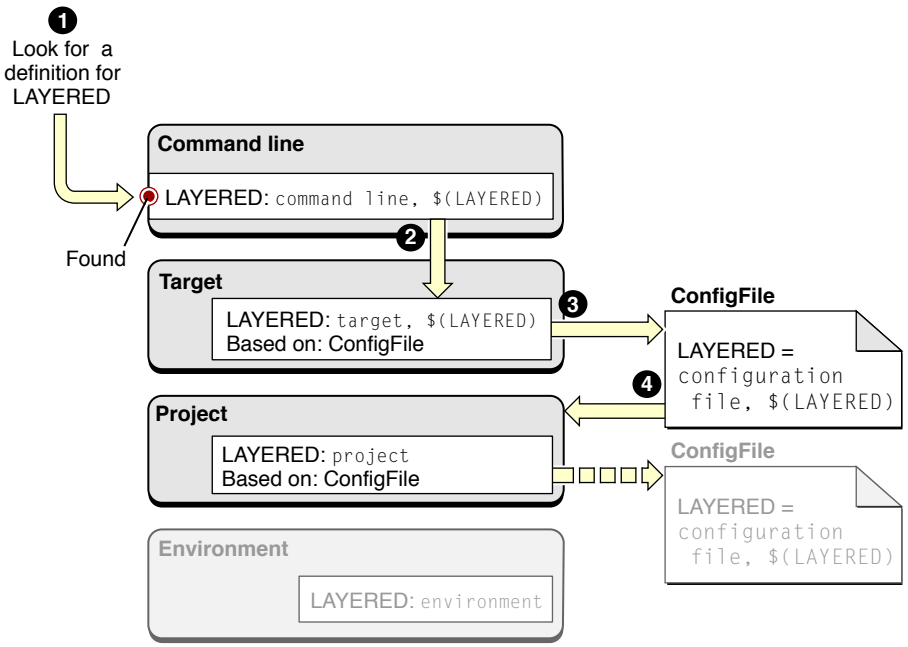
1. Looks for a definition for `LAYERED` in the command-line layer (the highest available to `xcodebuild`). It finds the specification `command line, $(LAYERED)`.
2. Resolves `$(LAYERED)` starting at the next layer down, the target layer. At this layer, it obtains the specification `target, $(LAYERED)`; this is the specification defined in the Xcode application for the active build configuration of the target. Because the specification also references the value of the `LAYERED` build setting, the build system continues to look for the build setting specification. In this case, the build configuration is based on a configuration file, so the build system continues its search there.
3. Looks in the `ConfigFile.xcconfig` file on which the active build configuration of the active target is based. In the configuration file, it obtains the specification `configuration file, $(LAYERED)`. Because this specification also references the value of the `LAYERED` build setting, the build system continues to look in lower layers for the build setting specification.
4. Resolves `$(LAYERED)` starting at the project layer, obtaining `project, $(LAYERED)`.

5. Looks in the `ConfigFile.xcconfig` file on which the active build configuration of the project is based, again obtaining the specification `configuration file, $(LAYERED)`. In this example, the project-level build configuration is based on the same configuration file as the target-level build configuration; however, they could just as easily be based on different configuration files.
6. Resolves `$(LAYERED)` starting at the environment layer, obtaining `environment`.

   The evaluation of `LAYERED` stops here because there are no build setting layers below the environment layer. When all the references are resolved, the final value of the build setting is computed as `command line, target, configuration file, project, configuration file, environment`.

Knowing how the build system evaluates build settings in configuration files lets you easily determine where to configure build settings. For example, say you have a configuration file that contains settings common to most of your Xcode projects. Imagine that, for a particular project, you want to override the value of a single setting in the configuration file. Rather than creating a separate configuration file that defines this one build setting differently, you can simply define that build setting in the project inspector for that project. Figure 4-5 shows how the build system evaluates the `LAYERED` build setting when that build setting is overridden in the project inspector of the Xcode application.

Figure 4-5 The `LAYERED` build setting overridden in the project inspector



These are the steps the build system takes to evaluate the LAYERED build setting:

1. Looks for a definition for `LAYERED` in the command-line layer (the highest available to `xcodebuild`). It finds the specification `command line, $(LAYERED)`.

2. Resolves `$(LAYERED)` starting at the next layer down, the target layer. At this layer, it obtains the specification `target, $(LAYERED)`.
3. Looks in the `ConfigFile.xcconfig` file on which the active build configuration of the active target is based. In the configuration file, it obtains the specification `configuration file, $(LAYERED)`.
4. Resolves `$(LAYERED)` starting at the project layer, obtaining `project`.

   The evaluation of `LAYERED` stops here because there are no references to other build settings in its specification. Even though `LAYERED` is configured in the project layer, that specification has been overridden in the target layer; therefore, it's ignored. The final value of `LAYERED` in this example is `command line, target, configuration file, project`.

Linking

The Link Binary With Libraries build phase in Xcode projects links frameworks and libraries with object files to produce a binary file. Source files that use code in a framework or a library must include a reference to the appropriate programming interface contained in them.

To learn the basics of linking frameworks and libraries, see Managing Libraries and Frameworks.

Specifying the Type of Binary to Create

The binary file (or image) a target produces can be of one of these types:

- **Executable.** Executable files are binaries containing the core logic of a program. Specifically, they contain the program's entry point, which in C-based programs is the `main` function. Once loaded, however, these programs may require the loading of dynamic libraries to perform functionality they don't implement. To generate an executable file, a target may include static libraries directly, or dynamic libraries, either directly or through frameworks. Executable binaries don't have an extension; for example, `MyProgram`).
- **Bundle.** Bundles are executable files that can be loaded at runtime by other products. Plug-ins are implemented using bundles. The term *bundle* in this context refers to the binary itself, not to a structured hierarchy. Bundles have the `.bundle` extension; for example, `MyBundle.bundle`.
- **Static Library.** Static libraries (also known as archives) are repositories of code that other programs incorporate as their own, which produces larger images. The code obtained from the static library is loaded at the same time the product is loaded. These products, however, do not benefit from improvements made in the static library after they are built. To get new functionality, the products must be rebuilt with the improved version of the static library. Static libraries have the `.a` extension; for example, `MyLibrary.a`.
- **Dynamic Library.** Dynamic libraries contain code that programs load at runtime. The images of products that use dynamic libraries are smaller because they don't incorporate the library's code into their own. Instead, these products load the appropriate library at runtime. Using dynamic libraries, products can automatically take advantage of features introduced in versions of the libraries that were not available when the products were built. That is, a program can take advantage of new functionality without being rebuilt. Dynamic libraries have the .dylib extension; for example, `libMyLibrary.1.0.dylib`.

The Mach-O Type (MACH_O_TYPE) build setting allows you to specify the type of binary a target produces. By default, this build setting is set to the appropriate value depending on the target's type. For example, in a Cocoa static library target, Mach-O Type is Static Library; in a BSD dynamic library target, Mach-O Type is Dynamic Library. However, you can change Mach-O Type for a target when you need a different binary type than the target default. For example, if you want to implement a Cocoa plug-in as a dynamic library instead of a bundle, create a Cocoa-bundle project and change Mach-O Type from Bundle to Dynamic Library.

Specifying the Search Order of External Symbols

The order in which frameworks and libraries are listed in the Link Binary With Libraries build phase specifies the order in which external symbols are resolved by the static linker at build time and the dynamic linker at runtime. When either of the linkers encounters an undefined external symbol, they look for the symbol starting with the first framework or library listed in the build phase.

When a program is built, the static linker replaces references to external symbols with the addresses of the symbols in the referenced libraries (this is called *prebinding*), or tells the dynamic linker to resolve the references when a program is loaded or when a symbol is referenced. Having the dynamic linker resolve references to external symbols at runtime provides the most flexibility, because a program can link with new versions of the symbols as they become available. However, this approach is not recommended for all situations, because linking with newer versions of a method or a function may cause problems during a program's execution.

In addition, how frameworks and libraries are listed in a Link Binary With Libraries build phase tells the static linker the approach (or the semantics) to use when binding or resolving references to external symbols defined in libraries.

Placing static libraries after dynamic libraries in the Link Binary With Libraries build phase ensures that the static linker binds references to external symbols defined in static libraries at build time, even when newer versions of the static libraries the application originally was linked with are present in the user's system.

When static libraries are listed before a dynamic library in the Link Binary With Libraries build phase, the static linker doesn't resolve references to symbols in those static libraries. Instead, those symbols are resolved by the dynamic linker at runtime. This may cause problems when the static libraries are updated, because the application links to the new, potentially incompatible versions of the symbols instead of the ones the developer intended.

For details on how symbols are resolved, see "Finding Imported Symbols" in Executing Mach-O Files in *Mac OS X ABI Mach-O File Format Reference* and the `ld` man page.

Preventing Prebinding

Mac OS X includes a prebinding mechanism used to speed up application launch in programs that link against dynamic libraries. When a user installs an application or upgrades the operating system, a prebinding agent links the application against new versions of the dynamic libraries. Sometimes, however, you may want to prevent this behavior for specific applications.

To link the binary file, framework, library, or plug-in so that prebinding is never performed on it, you need to add the `-nofixprebinding` option to the linker invocation. To do this, add `-nofixprebinding` to the Other Linker Flags (OTHER_LDFLAGS) build setting. See the `ld` man page for more information.

> **Note:** Prebinding is essential for applications running on Mac OS X version 10.3.3 or earlier. Improvements to the dynamic linker in Mac OS X v10.3.4 made prebinding applications unnecessary and in Mac OS X v10.4 prebinding is no longer recommended for applications. By default, prebinding is off for all products except dynamic libraries when targeting Mac OS X v10.4.

Reducing the Number of Exported Symbols

By default, Xcode builds binary files that export all their symbols. To reduce the number of symbols you want to export from a binary file, create an export file and set the Exported Symbols File (EXPORTED_SYMBOLS_FILE) build setting to the name of the file. For more information, see Improving Locality of Reference.

Reducing Paging Activity

To help reduce your application's paging activity at runtime, you may specify an order file to the linker. You do this by setting the Symbol Ordering Flags (SECTORDER_FLAGS) build setting to `-sectorder __TEXT __text <order_file>`. For information on order files, see Improving Locality of Reference in *Code Size Performance Guidelines* .

Dead-Code Stripping

The static linker (`ld`) supports the removal of unused code and data blocks from executable files. This process (known as *dead-code stripping*) helps reduce the overall size of executables, which in turn improves performance by reducing the memory footprint of the executable. It also allows programs to link successfully when unused code refers to an undefined symbol (instead of resulting in a link error).

Dead-code stripping is not limited to removing only unused functions and executable code from a binary. The linker also removes any unused symbols and data that reside in data blocks. Such symbols might include global variables, static variables, and string data, among others.

When dead-code stripping is enabled, the static linker searches for code that is unreachable from an initial set of live symbols and blocks. The initial list of live symbols and blocks may include the following:

- Symbols listed in an exports file; alternatively, it would include the symbols absent from a list of items marked as *not* to be exported. For dynamic libraries or bundles without an exports file, all global symbols are part of the initial live list. See Preventing the Stripping of Unused Symbols (page 87) for more information.
- The block representing the default entry point or the symbol listed after the `-e` linker option, either of which identifies the specific entry point for an executable. See the `ld` man page for more information on the `-e` option.
- The symbol listed after the `-init` linker option, which identifies the initialization routine for a shared library. See the `ld` man page for more information.
- Symbols whose declaration includes the `used` attribute. See Preventing the Stripping of Unused Symbols (page 87) for more information.
- Objective-C runtime data.
- Symbols marked as being referenced dynamically (via the REFERENCED_DYNAMICALLY bit in `/usr/include/mach-o/nlist.h`).

Enabling Dead-Code Stripping in Your Project

To turn on dead-code stripping in your project, you must pass the appropriate command-line options to the linker. From Xcode, you add these options in the Build pane of the target or project inspector; otherwise, you must add these options to your makefile or build scripts. Table 5-1 lists the Xcode build settings that control dead-code stripping. Turning on either of these build settings causes Xcode to build with the corresponding linker option.

Table 5-1 Xcode build settings for dead-code stripping

Build setting	Linker option
Dead Code Stripping (DEAD_CODE_STRIPPING)	`-dead_strip`
Don't Dead-Strip Inits and Terms (PRESERVE_DEAD_CODE_INITS_AND_TERMS)	`-no_dead_strip_inits_and_terms`

Table 5-2 lists the basic dead-code stripping options.

Table 5-2 Linker options for dead-code stripping

Linker option	Description
`-dead_strip`	Enables basic dead-code stripping by the linker.
`-no_dead_strip_inits_and_terms`	Prevents all constructors and destructors from being stripped when the `-dead_strip` option is in effect, even if they are not live.

After turning on dead-code stripping, you may want to set the *strip style* used. The strip style specifies the level of stripping performed. There are three levels of stripping available:

- All Symbols. All symbols are stripped from the binary, which includes the symbol table and relocation information.
- Non-Global Symbols. Nonglobal symbols are removed. External symbols remain.
- Debugging Symbols: Debugging symbols are removed. Local and global symbols remain.

Use the Strip Style (STRIP_STYLE) build setting to specify the desired strip level.

You must recompile all object files using the compiler included with Xcode 1.5 or later before dead-code stripping can be performed by the linker. You must also compile the object files with the `-gfull` option to ensure that the resulting binaries can be properly debugged. In Xcode, change the value of the Level of Debug Symbols (GCC_DEBUGGING_SYMBOLS) build setting to All Symbols (`-gfull`).

> **Note:** The GCC compiler's `-g` option normally defaults to `-gused`, which reduces the size of `.o` files at the expense of symbol information. Although the `-gfull` option does create larger `.o` files, it often leads to smaller executable files, even without dead-code stripping enabled.

Identifying Stripped Symbols

If you want to know what symbols were stripped by the static linker, you can find out by examining the linker-generated load map. This map lists all of the segments and sections in the linked executable and also lists the dead-stripped symbols. To have the linker generate a load map, add the `-M` option to your linker command-line options. In Xcode, you can add this option to the Other Linker Flags build setting.

Note: If you are passing this option through the `cc` compiler driver, make sure you pass this flag as `-Wl,-M` so that it is sent to the linker and not the compiler.

Preventing the Stripping of Unused Symbols

If your executable contains symbols that you know should not be stripped, you need to notify the linker that the symbol is actually used. You must prevent the stripping of symbols when external code (such as plug-ins) use's those symbols but local code does not.

There are two ways to tell the linker not to strip a symbol. You can include it in an exports file or mark the symbol declaration explicitly. To mark the declaration of a symbol, you include the `used` attribute as part of its definition. For example, you would mark a function as used by declaring it as follows:

```
void MyFunction(int param1) __attribute__((used));
```

Alternatively, you can provide an exports list for your executable that lists any symbols you expect to be used by plug-ins or other external code modules. To specify an exports file from Xcode, use the Exported Symbols File (EXPORTED_SYMBOLS_FILE) build setting; enter the path, relative to the project directory, to the exports file. To specify an exports file from the linker command line, use the `-exported_symbols_list` option. (You can also specify a list of symbols *not* to export using the `-unexported_symbols_list` option.)

If you are using an exports list and building either a shared library, or an executable that will be used with `ld`'s `-bundle_loader` flag, you need to include the symbols for exception frame information in the exports list for your exported C++ symbols. Otherwise, they may be stripped. These symbols end with `.eh`; you can view them with the `nm` tool.

Assembly Language Support

If you are writing assembly language code, the assembler now recognizes some additional directives to preserve or enhance the dead-stripping of code and data. You can use these directives to flag individual symbols, or entire sections of assembly code.

Preserving Individual Symbols

To prevent the stripping of an individual symbol, use the `.no_dead_strip` directive. For example, the following code prevents the specified string from being stripped:

```
.no_dead_strip _my_version_string
.cstring
_my_version_string:
```



```
.ascii "Version 1.1"
```

Preserving Sections

To prevent an entire section from being stripped, add the `no_dead_strip` attribute to the section declaration. The following example demonstrates the use of this attribute:

```
.section __OBJC, __image_info, regular, no_dead_strip
```

You can also add the `live_support` attribute to a section to prevent it from being stripped if it references other code that is live. This attribute prevents the stripping of some code that might actually be needed but not referenced in a detectable way. For example, the compiler adds this attribute to C++ exception frame information. In your code, you might use the attribute as follows:

```
.section __TEXT, __eh_frame, coalesced, no_toc+strip_static_syms+live_support
```

Dividing Blocks of Symbols

The `.subsections_via_symbols` directive notifies the assembler that the contents of sections may be safely divided into individual blocks prior to dead-code stripping. This directive makes it possible for individual symbols to be stripped out of a given section if they are not used. This directive applies to all section declarations in your assembly file and should be placed outside of any section declarations, as shown below:

```
.subsections_via_symbols


; Section declarations...
```

If you use this directive, make sure that each symbol in the section marks the beginning of a separate block of code. Implicit dependencies between blocks of code might result in the removal of needed code from the executable. For example, the following section contains three individual symbols, but execution of the code at `_plus_three` ends at the `blr` statement at the bottom of the code block.

```
.text
.globl _plus_three
_plus_three:
addi r3, r3, 1
.globl _plus_two
```

```
_plus_two:
addi r3, r3, 1
.global _plus_one
_plus_one:
addi r3, r3, 1
blr
```

If you were to use the `.subsections_via_symbols` directive on this code, the assembler would permit the stripping of the symbols `_plus_two` and `_plus_one` if they were not called by any other code. If this occurred, `_plus_three` would no longer return the correct value because part of its code would be missing. In addition, if `_plus_one` were stripped, the code might crash when it continued executing into the next block.

Reducing Build Times

Objective-CSwift

Xcode offers a number of features you can take advantage of to decrease build time for your project. For example, you can use distributed builds to shorten the time it takes to build your whole project or multiple projects. ZeroLink and predictive compilation, on the other hand, improve turnaround time for single file changes, thereby speeding up the edit–build–debug cycle. All of these features reduce the amount of time you spend idle while waiting for your project to build.

This chapter describes the following features:

- Precompiled prefix headers let you decrease the amount of time spent building compiling each source file in a target by specifying a single header file that includes all of the headers commonly used by the target's files and compiling this header a single time.
- Predictive compilation reduces the time required to compile single file changes by beginning to compile a file while you are still editing it.
- Distributed builds can dramatically reduce build time for large projects by distributing compiles to available machines on the network.

Fix and Continue, described in Modifying Running Code, improves your debugging efficiency by allowing you to make changes to your application and see the results of your modifications without stopping your debugging session.

Using a Precompiled Prefix Header

A precompiled header is a file in the intermediate form used by the compiler to compile a source file. Using precompiled headers, you can significantly reduce the amount of time spent building your product. Often, many of the source code files in a target include a subset of common system and project headers. For example, each source file in a Cocoa application typically includes the `Cocoa.h` system header, which in turn includes a number of other headers. When you build a target, the compiler spends a great deal of time repeatedly processing the same headers.

You can significantly reduce build time by providing a prefix header that includes the set of common headers used by all or most of the source code files in your target and by having Xcode precompile that prefix header.

If you have indicated that Xcode should do so, Xcode precompiles the prefix header when you build the target. Xcode then includes that precompiled header file for each of the target's source files. The contents of the prefix header are compiled only once, resulting in faster compilation of each source file. Furthermore, subsequent builds of the target can use that same precompiled header, provided that nothing in the prefix header or any of the files on which it depends has changed. Each target can have only one prefix header.

When Xcode compiles your prefix header, it generates a variant for each C language dialect used by your source files; it stores these in the folder specified by the `SHARED_PRECOMPS_DIR` (Precompiled Headers Cache Path) build setting. It also generates—as needed—a variant for each combination of source header and compiler flags. For example, you may have per-file compiler flags set for some of the files in the target you are building. Xcode will create a variant of the precompiled header by precompiling the prefix header with the set of compiler flags derived from the target and the individual source file. As Xcode invokes the compiler to process each source file in your target, the compiler searches the precompiled header directory for a precompiled header variant matching the language and compiler flags for the current compile. Xcode uses the first precompiled header variant that is valid for the compilation.

Precompiled headers can be reused across targets and projects. Targets that share the same prefix header and common compiler settings will automatically share the same precompiled header.

Xcode automatically regenerates the precompiled header whenever the prefix header, or any of the files it depends on are changed, so you don't need to manually maintain the precompiled header.

Creating the Prefix Header

To take advantage of precompiled headers in Xcode, you must first create a prefix header. Create a header file containing any common `#include` and `#define` statements used by the files in your target.

> **Note:** You can use a prefix header to include a common set of header files for each source file in your target without precompiling the prefix header.

Do not include in the prefix header anything that changes frequently. Xcode recompiles your precompiled header file when the prefix header, or any of the headers it includes, change. Each time the precompiled header changes, all of the files in the target must be recompiled. This can be an expensive operation for large projects.

Because the compiler includes the prefix header file before compiling each source file in the target, the contents of the prefix header must be compatible with each of the C language dialects used in the target. For example, if your target uses Cocoa and contains both Objective-C and C source files, the prefix header needs to include the appropriate guard macros to make it compatible with both language dialects, similar to the example shown here:

```
#ifdef __OBJC__
#import <Cocoa/Cocoa.h>
#endif
#define MY_CUSTOM_MACRO 1
#include "MyCommonHeaderContainingPlainC.h"
```

Configuring Your Target to Use the Precompiled Header

Once you have created the prefix header, you need to set up your target to precompile that header. To do this, you must provide values for the two build settings described here. You can edit these build settings in the Build pane of the target inspector or Info window. The settings you need to change are:

- `GCC_PREFIX_HEADER` (Prefix Header). Change the value of this build setting to the project-relative path of the prefix header file. If you have a precompiled header file from an existing project, set the prefix header path to the path to that file.
- `GCC_PRECOMPILE_PREFIX_HEADER` (Precompile Prefix Header). Make sure that this option is turned on. A checkmark is present in the Value column if this option is enabled.

You must provide values for these settings in each target that uses a precompiled prefix header, even if those targets use the same prefix header.

By default, Xcode precompiles a version of the header for each C-like language used by the target (C, C++, Objective-C, or Objective-C++). The `GCC_PFE_FILE_C_DIALECTS` (C Dialects to Precompile) build setting lets you explicitly specify the C dialects for which Xcode should produce versions of the precompiled header.

Sharing Precompiled Header Binaries

It is possible to share a precompiled header binary across multiple targets in multiple projects, provided that those targets use the same prefix header and compiler options. In order to share a precompiled header binary, each individual target must have the same prefix header specified. To use the same prefix header for multiple targets, for each target set the value of the Prefix Header build setting to the path to the header.

Each target must also specify a common directory as the destination for the precompiled header generated by Xcode. The `SHARED_PRECOMPS_DIR` (Precompiled Headers Cache Path) build setting specifies the location of the directory to which Xcode writes the precompiled header binary. By default, this is defined as `<Xcode_Persistent_Cache>/com.apple.Xcode.$(UID)/SharedPrecompiledHeaders` for all newly created or upgraded projects. When Xcode invokes GCC to compile each source file in a target, GCC searches this common directory for the appropriate header binary. For any targets that use the same prefix header and compiler options, GCC uses the same precompiled header binary when it builds those targets.

Xcode also defines the build setting `PRECOMPS_INCLUDE_HEADERS_FROM_BUILT_PRODUCTS_DIR` (Precompiled Header Uses Files From Build Directory). This build setting is a hint to the build system; it indicates whether a target's prefix header includes headers files from the target's build directory. If your prefix header includes header files from the build directory, set this build setting to `YES` by selecting the checkbox next to the build setting in the Build pane. If your targets share a common build directory outside of the project directory, Xcode still shares precompiled headers across compatible targets. If, however, you use the default build directory location within the project directory, Xcode shares precompiled headers across compatible targets only within the same project.

If your target's prefix header does not include headers from its build directory, set the `PRECOMPS_INCLUDE_HEADERS_FROM_BUILT_PRODUCTS_DIR` build setting to `NO`, to increase the chances that Xcode will be able to use a shared precompiled header when building the target.

If you specify the same prefix header for multiple targets but do not specify a common location for the precompiled binary, Xcode precompiles the prefix header once for each target as it is built.

Regenerating Precompiled Headers

Xcode compiles your prefix header whenever it fails to find a matching precompiled header binary in the shared precompiled header location. As described earlier in this chapter, Xcode automatically rebuilds your precompiled prefix header if you make changes to the prefix file or to any files it includes. Xcode also recompiles your prefix header if you make changes to your target's build settings that affect the compiler options used to build the target's files.

You can also force Xcode to rebuild your precompiled header by:

- Touching the prefix file.
- Cleaning the target. By default, cleaning a target removes all of the intermediate and product files generated for that target by the build system, including any precompiled headers. The next time you build the target, Xcode rebuilds the target's precompiled header files, along with all other files in the target. See Removing Build Products and Intermediate Build Files.

By default, cleaning a target removes the precompiled header for that target—whether or not Xcode initially generated the precompiled header binary as part of building that target. Because rebuilding precompiled headers can be time consuming, you may want to clean an individual target without discarding the precompiled header binaries for that target if you are sharing precompiled headers across many targets in multiple projects. When you initiate a clean, Xcode displays a dialog that lets you choose whether to preserve precompiled headers. To keep precompiled headers, deselect the Also Remove Precompiled Headers option. By default, this option is enabled.

Controlling the Cache Size Used for Precompiled Headers

Xcode caches the precompiled header files that it generates. To control the size of the cache devoted to storing those files, use the `BuildSystemCacheSizeInMegabytes` user default. In the Terminal application, type:

```
defaults write com.apple.xcode BuildSystemCacheSizeInMegabytes defaultCacheSize
```

Specifying 0 for the cache size gives you an unlimited cache. The default cache size set by Xcode is 1024 MB. If the cache increases beyond the default size, Xcode removes as many precompiled headers as is necessary to reduce the cache to its default size when Xcode is next launched. Xcode removes the oldest files first.

Restrictions

To take advantage of Xcode's automatic support for precompiled headers you must:

- Use GCC 3.3 or later.
- Use a native target. Automatic support by Xcode for precompiled prefix headers using PCH is available only for targets that use the Xcode native build system. Xcode automatically handles many of the restrictions upon using precompiled headers with GCC. If you are using an external target to work with another build system—such as `make`—you can still use precompiled headers, but you must create and maintain them yourself. For more information on using precompiled headers with GCC, see *GNU C/C++/Objective-C 3.3 Compiler*. Jam-based targets can also use precompiled prefix headers, but they are limited to the PFE (persistent front end) mechanism introduced with GCC 3.1. PFE is no longer recommended.
- Use one and only one prefix header per target.
- Set the Prefix Header and Precompile Prefix Header build settings for every target that uses precompiled headers.

Predictive Compilation

Predictive compilation is a feature introduced to reduce the time required to compile single file changes and speed up the edit–compile–debug cycle of software development. If you have predictive compilation enabled for your project, Xcode begins compiling the files required to build the current target even before you tell Xcode to build.

Predictive compilation uses the information that Xcode maintains about the build state of targets that use the native build system. Xcode keeps the graph of all files involved in the build and their dependencies, as well as a list of files that require updating. At any point in time, Xcode knows which of the files used in building a target's product are out of date and what actions are required to bring those files up to date. A file can be updated when all of the other files on which it depends are up to date. As files become available for processing, Xcode begins to update them in the background, even as you edit your project.

Xcode will even begin compiling a source code file while you are editing it. Xcode begins reading in and parsing headers, making progress compiling the file even before you initiate a build. When you do choose to save and build the file, much of the work has already been done.

Until you explicitly initiate a build, Xcode does not commit any of the output files to their standard location in the file system. When you indicate that you are done editing, by invoking one of the build commands, Xcode decides whether to keep or discard the output files that it has generated in the background. If none of the changes made subsequent to its generation affect the content of a file, Xcode commits the file to its intended location in the file system. Otherwise, Xcode discards its results and regenerates the output file.

You can turn on predictive compilation by selecting "Use Predictive Compilation" option in the Building pane of the Xcode Preferences window.

Predictive compilation works only with GCC 4.0 or later and native targets. All predictive compilation is done locally on your computer, regardless of whether you have distributed builds enabled. On slower machines, enabling predictive compilation may interfere with Xcode performance during editing.

Using Multiple SDKs

When developing Mac OS X applications, you normally use a *system SDK*, such as the Mac OS X v10.4 SDK or the Mac OS X v10.5 SDK. In this document, an SDK (software development kit) is a set of frameworks or libraries that provide particular functionality to your product. A system SDK is one included in Xcode to provide an interface to core Mac OS X functionality. System SDKs reside in `<Xcode>/SDKs`.

There may be times when, in addition to a system SDK, you need link against *sparse SDKs*. A *sparse SDK* is an SDK that is not a system SDK. Sparse SDKs may be provided by third parties, or you may build them yourself.

To specify additional SDKs to link against, use the `ADDITIONAL_SDKS` (Additional SDKs) build setting. You can specify the paths to sparse SDKs in this build setting.

At build time, the build system groups the target's additional SDKs into a single SDK, known as a *composite SDK*. Composite SDKs are cached in the `<Xcode_Persistent_Cache>` directory (see Overview of Xcode Projects for details on the Xcode persistent-cache location). Therefore, targets (from any of your projects) that list the same set of additional SDKs, share the composite SDK the build system generates, which improves compilation times.

Distributing Builds Among Multiple Computers

Building a product involves several tasks. Some tasks, such as compiling precompiled headers and linking a binary, can be performed effectively only by the computer that hosts a build. However, the main task performed in a build—compiling source files—can be carried out in parallel by several computers. Builds that use several computers to compile source files are known as *distributed builds*. When you use distributed builds, Xcode distributes as many compilation tasks among the computers available for this purpose within a network.

> **Software requirements:** The contents of this section apply to Xcode 3.2.2 on Mac OS X v10.6.2.

Distributed builds use two types of computers: build clients and build servers.

- A *build client* is the computer that performs the build. This is the computer that runs the Xcode or xcodebuild instance that carries out the build command.
- A *build server* is a computer that a build host can use to perform compilation tasks. Build servers do not need to run Xcode or xcodebuild to aid a build host. But they must at least be running the same version of the Mac OS as the build server.
- A *build set* is made up of the host names of build servers to which a build client distributes compilation tasks. To distribute a build, you must define at least one build set on the build client, or use the Bonjour set when available.

Xcode distributes builds among computers using a technology known as shared workgroup builds.

> **Note:** Earlier releases of Xcode supported a type of distributed build called dedicated network build (DNB). Dedicated network builds are not supported in Xcode 3.2.2.

To specify whether builds started by a particular user on a computer are to be distributed to other computers (that is, whether the computer acts as a build client for that user), use the "Distribute via shared workgroup builds" option and pop-up menu in the Distributed Builds preferences while logged in as that user (see Distributed Builds Preferences (page 101) for details). You may also specify build servers in `xcodebuild` invocations.

The following sections describe distributed workgroup builds in more detail and provide guidelines for deciding which type of distributed build to use for a particular project.

Shared Workgroup Builds

Shared workgroup builds work best when building small to medium projects using up to ten build servers. Their main advantage is easy set up (including Bonjour support). Their main disadvantage is the requirements it imposes on the computers to be used as build servers by a build client.

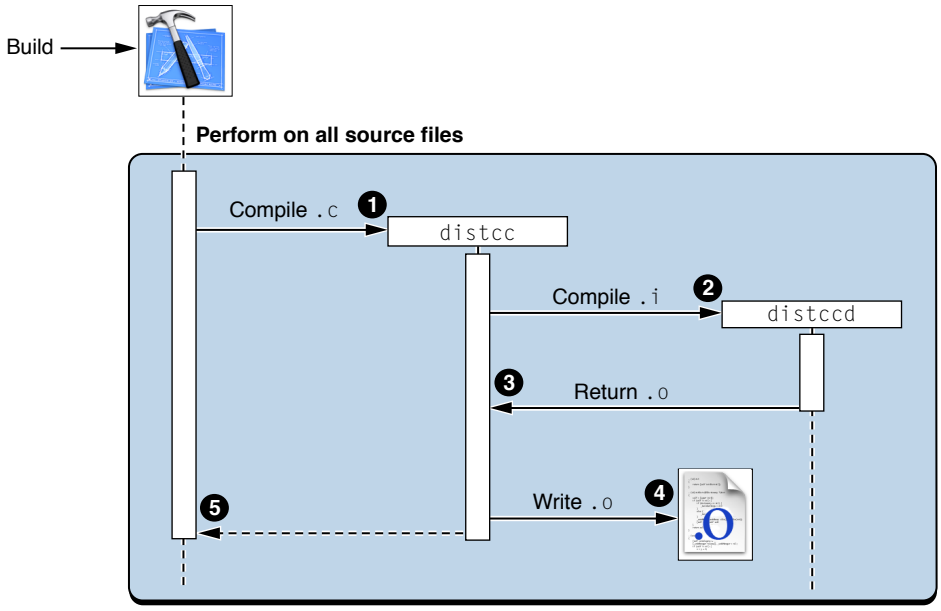
How Shared Workgroup Builds Work

To perform a shared workgroup build, build clients distribute compilation tasks to the build servers specified in the host list in Xcode Preferences > Distributed Builds or in the build-server list given to `xcodebuild`. The build client performs the build set-up, such as compiling precompiled headers, and linking. (Precompiled headers are compiled by the build client to ensure that all build servers use the same precompiled header in a build.)

To distribute the compilation task of a build, Xcode and `xcodebuild` invoke the `distcc` tool instead of `gcc`. The `distcc` tool is the client-side process that distributes compilation tasks among build servers.

Build servers run the `distccd` daemon, which responds to compilation requests sent by `distcc` in the build client. When a build server receives a compilation request, it obtains the preprocessed implementation files from the build client, invokes `gcc` to compile the files into an object file, and sends the object file to the build client. Figure 6-1 illustrates this process.

Figure 6-1 Shared workgroup build process



This list explains the process shown in Figure 6-1:

1. Xcode (or `xcodebuild`) invokes `distcc` in the build client to compile a source file.

2. Process `distcc` preprocesses the source file using the compiler specified by the target, finds a compatible build server, and sends the preprocessed source file to the `distccd` process running on that computer. See Requirements for Using Shared Workgroup Builds (page 98) for information on how compatible build servers are identified.
3. The `distccd` process in the build server invokes `gcc` with the file provided by `distcc` as input and returns the output `gcc` generates.
4. The `distcc` process in the build client places the output from `distccd` in the project's build directory.
5. The `distcc` process tells Xcode (or `xcodebuild` whether the compilation task was successful.

If a build client's compilation request fails—for example, if communication with the build server is lost or the build server cannot execute the compilation request—Xcode (or `xcodebuild`) performs the compilation on the build client.

Build servers respond to compilation requests as they are able. That is, a build server that is processing a compilation request, may not respond to another compilation request from a build client until it's done. Xcode and `xcodebuild` distribute compilation operations to the most capable build servers first. But, when there are enough compilation operations to distribute among the available build servers, Xcode uses all of them, regardless of their processing power. For example, when a build client has three build servers available, two fast ones and one slow one, and a build requires three compilation tasks, all the servers receive a compilation request, even it the two fast servers would complete the tasks faster by themselves than when the slower server is used.

Build servers cache precompiled headers to speed-up subsequent build requests that require the same precompiled header. Before compiling a preprocessed file, the build server determines whether it needs to get a new copy of the precompiled header from the build client. Therefore, modifying precompiled headers (as for in non–distributed builds) adversely affects the performance of shared workgroup builds.

Requirements for Using Shared Workgroup Builds

Use of distributed builds is subject to the following constraints:

- You must use GCC 3.3 or later.
- Computers that compilation tasks are distributed to must be running the same version of the compiler specified by the target and the same operating system as the build client. These computers must also have the same architecture as the build client. For example, an Intel-based Macintosh can distribute builds only among other Intel-based Macintosh computers. A PowerPC-based Macintosh can distribute builds only among other PowerPC-based Macintosh computers.
- Shared workgroup builds are available only to native targets. Jam-based targets or targets using another external build system are not compatible with shared workgroup builds.

- Shared workgroup builds support C, C++, Objective-C, and Objective-C++.
- Shared workgroup builds are enabled on a per-user basis for all projects and targets built by that user.

Sharing a Computer as a Shared Workgroup Build Server

To advertise a computer as a shared workgroup build server, select the appropriate option in Xcode Preferences > Distributed Builds. As an alternative, you may execute the following command:

```
sudo /bin/launchctl load -w /System/Library/LaunchDaemons/distccd.plist
```

> **Note:** When you make a computer a build server, build clients may use it at any time, whether Xcode is running and whether you are logged in to the computer. See "Distributed Builds Preferences Pane" for more information.

You should not share your computer if you are using it to host distributed builds. That is, you should not have both "Share my computer for building" and "Distribute builds" selected in the Distributed Builds preference pane. Because precompilation and the distribution of build tasks are done on your local computer, allowing others to distribute build tasks to your computer can significantly slow down your own builds.

Getting the Most Out of Distributed Builds

Using Xcode to distribute builds across multiple computers will reduce the time it takes to build your product. However, the degree of improvement in build times you get depends on many factors. These are a few of them:

- **Networking speed.** In shared workgroup builds Xcode sends files to be built on other computers; those computers, in turn, send back the resulting object files. To see a significant reduction in build time, your network must be fast enough that the cost of transferring files between computers is minimal. Very busy or slow networks may have a detrimental effect on distributed build performance.
- **Network accessibility.** Build clients that have different network interface (for example, Ethernet and AirPort) may see different sets of build servers if the networks are not completely bridged.

  For example, consider a workgroup in which the Ethernet network is protected by a firewall but the AirPort network is public. In such a workgroup, a build client that uses AirPort exclusively cannot access any of the Ethernet-based build servers.
- **Number of available computers.** The more processors you have at your disposal, the more compilation tasks can be performed in parallel and the more significant the reduction in build time that you see. If you have a limited number of build servers available—especially if those computers do not have significant processing capacity—you may not see any noticeable improvements in build performance. The overhead

of managing the distribution of compilation tasks may overcome the benefit you get from compiling on additional machines. You may find that you get better build performance from utilizing other optimizations, such as precompiled headers or parallel builds, on your local computer.

- **Avoid using the host computer as a build server.** Xcode automatically performs compilation operations locally when there's a shortage of available build servers or when a build server is unable to carry it out a request within a certain period.

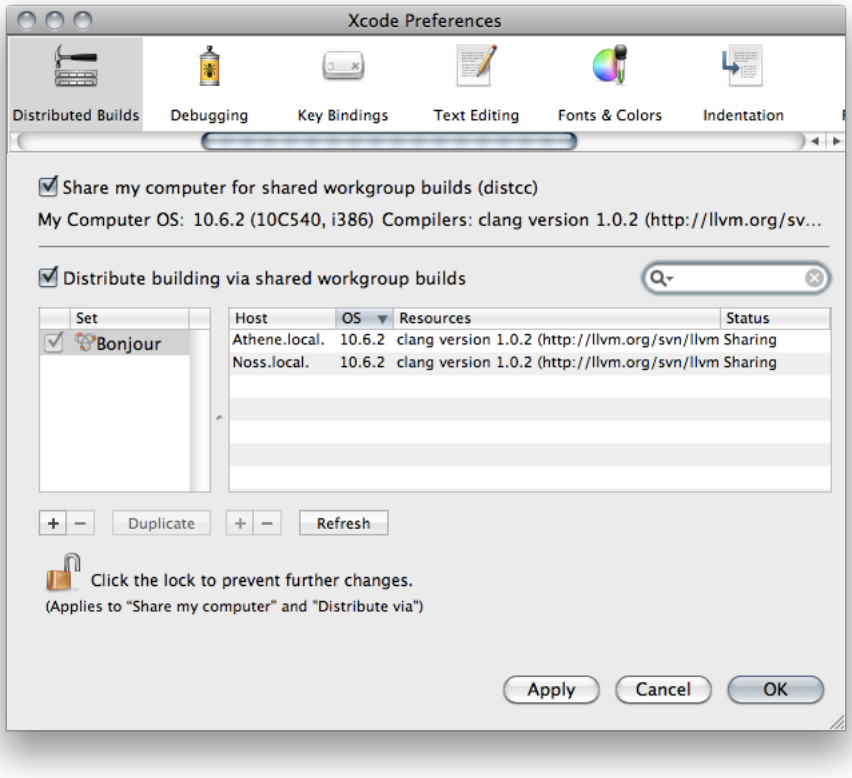
Xcode must completely finish building one target before starting to build another, which means that all the source files for a given target must complete compiling before any files from dependent targets can start compiling. Therefore, to maximize the potential for parallel compilation, instead of many targets with few source files per target, you should have fewer targets with more source files per target. A target with a single file will effectively stall all the build servers.

For shared workgroup builds, Xcode provides the default preference `XCMaxNumberOfDistributedTasks` to control the maximum number of parallel compilation tasks allowed during a build. Use this preference only if you want to reduce the number of compilation tasks Xcode creates to complete a build. For more information on these preferences, and on how to set them, see *Xcode User Default Reference*.

Distributed Builds Preferences

The Distributed Builds pane in Xcode preferences contains options for distributing build tasks to other computers in your network. Figure 6-2 shows the Distributed Builds pane.

Figure 6-2 Distributed Builds preferences pane



Here is what the pane contains:

- **Share my computer for shared workgroup builds.** Turning on this option makes your computer available to shared workgroup build clients.

  When you select this option, the pop-up menu specifies the priority assigned to compilation tasks that are distributed to your computer. This setting determines the amount of processing time allocated for a compilation task in relation to other tasks the computer is performing during a compilation task. You can choose high, low, or medium priority. The default value is “medium priority.” This value is appropriate for most situations. When you want build tasks to take precedence over any other tasks, choose “high priority.” When you want tasks other than build tasks to use the most processing power, choose “low priority.”

- **My Computer OS.** This text field indicates the operating system the local host is running.
- **Compilers.** This text field indicates the compilers available on the local host.

- **Distribute building via shared workgroup builds.** This option specifies whether to use distributed workgroup builds. When this option is selected, Xcode distributes builds to the sets selected in the build set list, explained next. Otherwise, all compilation tasks are performed on your computer.
- **Set list.** This list contains the names of build server sets to which builds can be distributed to. When "Distribute via shared workgroup builds" is selected, Xcode (or `xcodebuild`) distributes builds to the build servers that are members of the selected sets. A set's members are displayed in the host list when the set is highlighted in the build set list.

  The Bonjour set appears in the build set list only when you're distributing builds using shared workgroup builds. The build servers in this set are dynamically added or removed as they become available or unavailable for shared workgroup builds in your local network. See *Xcode Build System Guide* for more information.

  You can add, remove, and duplicate build sets using the buttons below the list. You can add host names to a set by dragging or copying names from another set or from a text file. (Note that you cannot modify the Bonjour build set.) Conversely, you can create host-list file containing the hosts in a build set by dragging or pasting a set to a text file. For example, you can create a host-list file identifying hosts you want members of your team to use for their distributed builds, and send it to them in an email message. Your colleagues can then create an empty build set and drag the host-list file onto it.
- **Host list.** This list contains the names of the hosts that are members of the sets highlighted in the build set list. The host list contains these columns:
  - **Host:** The name or IP address that identifies a build server. Note that once you add a host name or IP address to a build set, if the name or IP address of that host subsequently changes, Xcode (and `xcodebuild`) won't be able to locate it. You must update the name or IP address in the host list in order to distribute builds to that host.
  - **OS:** The operating system the host is running. When the host is not available, "Unavailable" appears in this column.
  - **Resources:** The compilers and other resources available in the host. Compilers that are incompatible with the corresponding compilers in the build client are shown in red. Xcode doesn't distribute a compilation task to a build server when the compiler required by the task is shown in red.
  - **Status:** The sharing state of the host. This column can have these values:
    - **Sharing:** The host is available as a build server. The host has been advertised as a build server. See How Shared Workgroup Builds Work (page 97) for details.
    - **Not Sharing:** The host is not available as a build server because it hasn't been advertised as such.
    - **Incompatible Service:** The host is not available as a build server because the version of the distributed build tool in the host is not compatible with the version in the build client. Make sure both computers have the same version of Xcode installed.
    - **Unreachable:** The host is not available as a build server because the local host cannot access it.

When you place the pointer over a host in the host list, a tooltip indicates the number of processors and speed of that host.

You can add and remove hosts from highlighted sets by using the buttons below the host list. You can also add hosts in the host list to a build set by dragging or pasting the hosts to the build set you want to add them to. You can create a host-list file with the host names of one or more hosts by selecting hosts in the host list and dragging or copying them to a text file. Then you can drag the host-list file into a build set to add the hosts identified in the file to the build set.

- **Search field.** This search field allows you to narrow the hosts shown in the host list.

Document Revision History

This table describes the changes to *Xcode Build System Guide*.

Date	Notes
2011-03-08	Replaced by the Xcode 4 User Guide and moved to the Legacy Reference Library.
2010-07-02	Made minor corrections.
2010-03-24	Updated details about distributed builds. Updated Distributing Builds Among Multiple Computers (page 96) to update the information for distributed workgroup builds, and to indicate that dedicated network builds (DNB) are not supported in Xcode 3.2.2 and later. Updated Run Script Build Phase (page 43) to describe how to avoid storing scripts in the project file.
2009-10-19	Added index.
2009-02-04	Made minor content changes.
2009-01-06	Made minor changes. Added example script to Run Script Build Phase (page 43).
2008-11-19	Made minor content addition. Added Distributed Builds Preferences (page 101) (previously published in *Xcode Workspace Guide*).
2008-09-09	Made minor content organization changes.
2008-07-08	New document that provides a detailed description of the Xcode build system and shows how to customize it.

Date	Notes
	The content for this document was previously published in *Xcode User Guide*.

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Index

H

L

M

P

R

S

T

X