AllAboutmakeandgnumake;¬All About make and gnumake

make is a standard UNIX command that builds programs. Its main purpose is to make it easy for you to perform incremental builds. Project Builder uses **gnumake**, a version of **make** written by the Free Software Foundation.

A make Terminology Primer

You tell **make** how to build a program by creating a *makefile*. A makefile consists of *rules*, which in turn are made up of *build targets*, a list of *dependencies*, and one or more *commands*.

Build targets are the targets of the **make** command, that is, what you want to build. In Project Builder, there are several provided targets that build the project in different ways. The default is to build an optimized, debuggable executable. You can select a different target to turn optimizations off, to generate profiling information, and to install the project in its end location. (One special target is **clean**, which removes all object files and executables, forcing a full build the next time around.)

Dependencies are the input files used to create the target. For example, an executable depends on the object files linked together to create it. When **make** is asked to build an executable, it checks all object files listed as dependencies. If the object file is out of date or does not exist yet, it creates that object file by compiling the source file. Once all object files listed as dependencies are created and are up-to-date, **make** can link them together to create the executable.

Commands are the commands used to create the target. For example, to create an application, you need to specify commands that compile all of the source code files into object files, link the object files into an executable, create the application's wrapper directory, and copy the executable and the application's resources into that directory.

A makefile can also contain *macros*, which make it easier to write consistent makefile rules. Makefile macros serve the same purpose as **#define** macros in a C program. They make it easier to update the makefile. For

example, you can define a macro **CFLAGS** to be all of the flags you usually pass to the Objective-C compiler and use it everywhere you specify a compile command. Then, if you want to change one of these options, you only need to change it in one place, in the definition of **CFLAGS**.

If you need to update makefiles at all in Project Builder, you usually only need to redefine some provided macros. If you want to, you can also create your own targets. This chapter describes how to perform these tasks.

gnumake

gnumake, from the Free Software Foundation, is now the default **make** utility for OPENSTEP. **gnumake** has many features that aren't available in other **make** utilities.

Some of the unique things you can do if you use **gnumake** are:

- Perform parallel compiles so that your project builds faster.
- Use conditional statements in a rule. You can use this feature to specify in a single rule different compiler arguments based on which type of compiler you are using. In other **make** utilities, you would have to define two different rules.
- Use a standard set of functions to manipulate strings and filenames used in the makefile. For example, **gnumake** provides functions that return a specified file's extension, its basename, and its directory.
- Define a macro based on its own previous definition. For example, **gnumake** allows you to say `**CFLAGS := $(CFLAGS) -O**' which assigns to the macro **CFLAGS** its previous definition with **-O** appended to it.
- Define a macro that contains a newline using the **define** directive.
- Use **MAKELEVEL** to keep track of recursive use of **make**.
- Declare a phony target with **.PHONY**.

- Specify a search path for included makefiles and specify extra makefiles to be read with an environment variable.
- Use **vpath** to specify a search path for files with a particular extension.
- Use a special search mechanism for libraries by specifying **-lname** as a dependency. This causes **gnumake** to search for the library in the **VPATH**, then in **vpath**, then in **/lib**, **/usr/lib**, and **/usr/local/lib**.

For More Information

A number of books describe **make** in general terms. If you need to learn more about **gnumake**, see the document ªGNU Makeº provided by the Free Software Foundation.

370776_TableRule.eps ¬

PortabilityDo'sandDon'ts;¬Portability Do's and Don'ts

If you build multiple architecture (ªfatº) binaries or you build code on one architecture to run on another, make sure you're writing portable code. If you use the OpenStep libraries and avoid hard-wired data values, your application will probably be portable. Here's a list of some specific do's and don'ts for writing portable code.

- *Do* use relative values when positioning windows on the screen. *Don't* use absolute positions.
- *Do* use the NSEvent **characters** method to find out what key was pressed. *Don't* use **keyCode**.
- *Do* use **sizeof** when passing the size value to **malloc()**. *Don't* use a constant.
- *Do* refer to a structure's fields and a function's parameters by name. *Don't* try to deconstruct data formats, such as **float** or **struct**, or a function's argument list yourself.
- *Do* use the OpenStep objects NSData, NSString, and NSDictionary to read and write external data. *Don't* rely on a particular byte order or alignment when reading and writing external data.

370776_TableRule.eps ¬

BuildingaMultipleArchitecture(ªFatº)Binary;¬Building a Multiple Architecture

(ªFatº) Binary

The following sequence of events occurs when you build a fat binary:

- Each source file is compiled once for each architecture to produce thin object files.

  The object files are stored in subdirectories under **dynamic_obj** (or **dynamic_debug_obj**) that are named for the processor. For example, if you build for both Intel and NeXT, there are two directories under **dynamic_obj**: **i386**, containing object files for Intel, and **m68k**, containing object files for NeXT.

- After all of the source files have been compiled, the linker is invoked once for each architecture to produce thin executables.

  Each executable has an extension that describes the type of processor it runs on, for example, *MyProject***.m68k**.

- After an executable has been built for each processor, the **lipo** command is invoked to combine the executables into one binary file named *MyProject*.

FatBinary.eps ¬

370776_TableRule.eps ¬

ThreeWaystoSetBuildOptions;¬Three Ways to Set Build Options

You can set build options in three different places: in the Preferences panel, in the Project Inspector panel, and in the Build Options panel. Each panel has a unique purpose.

- Use the Build Preferences panel to set options you're always going to use, no matter which project you're working on.

  For example, you may want to change the sound you hear upon each successful build. To open the Build Preferences panel, choose Info arrow.eps ¬ Preferences, then choose Build from the pop-up list in the Preferences panel.

- Use the Build Attributes inspector to set options that apply to a specific project, no matter which user is working on the project.

  For example, you might define a build target specific to one project. Or you might want to use a specific compiler option for one project. To bring up the Build Attributes inspector, choose Tools arrow.eps ¬ Inspector arrow.eps ¬ Show Panel, then choose Build Attributes from the pop-up list in the Project Inspector panel.

- Use the Build Options panel to set your preferences for a specific project. Options on this panel apply only when you yourself are building this project.

  To bring up this panel, click the check mark button on the Project Build panel. The options on the Build Options panel remain set even after you quit Project Builder.

ThreeBuildOptions.eps ¬

370776_TableRule.eps ¬

TheAppWrapperandOtherBundles;¬The App Wrapper and Other Bundles

Some project types (namely Application, Loadable Bundle, and Framework) don't just produce an executable when you build them. They create the executable, and then they create a directory containing the executable and the project's resources (the files under Images, Interfaces, and Other Resources in the project browser).

The generic term for a directory containing such items is ªbundle.º When the bundle contains an application, it is often called an ªapp wrapperº because it wraps up all of the things an application needs into a single unit.

Appwrapper.eps ¬

370776_TableRule.eps ¬

OtherBuildTargets;¬Other Build Targets

Besides the default target, the one named for the project type, the other available targets are:

debug Compiles with -DDEBUG on and optimizations off. Use this target if your program uses the DEBUG

macro to provide more debugging information or if you want
to make sure local variables do not get optimized while you are debugging.

install Places the executable in the installation directory specified in the Build Attributes inspector.

profile Generates (with -DPROFILE, -pg, and all warnings on) a file containing code to generate a **gprof** report. Use this target when you are tuning the performance of an application. See the **gprof** man page for details.

<default> Uses the first target listed in the makefile. **Warning:** If you place a target at the end of the **Makefile.preamble** file, it becomes the default target.

370776_TableRule.eps ¬

SomeOPENSTEPLibraries;¬Some OPENSTEP Libraries

Most of the OPENSTEP libraries are now delivered as frameworks. Because all of the files associated with a framework are in one location, it's pretty easy to find out which framework you need to link with if you import one of its headers.

There are still a few old-style libraries delivered with OPENSTEP. Here's a list of the more commonly used ones and when you would link against them:

- **/usr/lib/libcurses** Contains cursor control functions. Link with this library if you import the header file **curses.h**.
- **/usr/lib/libdbm** Contains UNIX database subroutines. Link with this library if you import the header file **dbm.h**.
- **/usr/lib/libDriver** Link with this library if your program interacts with a device driver.
- **/usr/lib/libg++** Contains the C++ libraries. Link with this library for projects that contain C++ code.
- **/usr/lib/libiostream** Contains C++ I/O streams support.

- **/usr/lib/libMallocDebug** Contains a special implementation of **malloc**. Link with this library if you want to use the MallocDebug application to examine your application's memory usage.

370776_TableRule.eps ¬

InterestingCompilerandLinkerOptions;¬Interesting Compiler and Linker Options

Here are some interesting compiler and linker options you may want to try. For more options, see the **cc(1)** and **ld(1)** man pages.

Compiler Options

-ansi Use strict ANSI C definition.

-traditional Use the traditional Kernigan & Ritchie C definition.

-bsd Use strict BSD semantics.

-Wpointer-arith Print a warning if pointer arithmetic is used on a void pointer or a function pointer.

-finline-functions Make all simple functions inline.

-pipe Use pipes in place of temporary intermediate files.

Linker Options

-sectorder Order the blocks in a specified section.

-undefined Specify how undefined symbols are treated: as errors, warnings, or ignored.

-whyload Indicate why each member of a library is loaded.

-y*sym* For a given symbol, list files that referenced it.

-Y*n* For the first *n* undefined symbols, lists the file that referenced the symbol.

370776_TableRule.eps ¬

DynamicLinking;¬Dynamic Linking

OPENSTEP 4.0 introduces dynamic linking. When you use dynamic linking, references are resolved at run time instead of at link time. This means you don't have to relink your application every time a definition in a dynamic library changes. You get the benefit of the changes without having to perform a build.

Dynamic linking is the default, and you must use it if you link with a dynamic shared library. All frameworks are dynamic shared libraries. (If you want to create your own dynamic shared library, see Chapter 12, ªCreating Frameworks and Dynamic Shared Libraries.º ;../../05_SpecialTasks/12_FrameworksLibraries/FrameworksLibraries.rtfd;;¬)

The main difference between static and dynamic linking is in how libraries are searched for unresolved references. When you use static linking, each library is searched for unresolved references exactly once.

When you use dynamic linking, the static linker must simulate the dynamic link editor to see if there are any unresolved references. It places each library in a search list. Then, whenever an unresolved reference is encountered, it searches each library in the search list in order until it can resolve the symbol. With dynamic linking, a library might be searched several times.

DynLink.eps ¬

370776_TableRule.eps ¬

ThePlatformPop-Up'sPurpose;¬The Platform Pop-Up's Purpose

Do you want your project to run on multiple platforms? For example, are you writing an application that you plan to have run on both Mach and Windows? Or maybe you're writing a framework that you also want to build on one of the Portable Distributed Objects (PDO) platforms.

If this is your situation, the platform pop-up list is for you. (This is the list that appears just above the compiler options on the Build Attributes inspector.) Different platforms have different requirements. For example, you might install the application in a different location on a Windows platform than you would a Mach platform. You'll need to use different linker options because the platforms each use a native linker. Set the options as you want them for one platform, then change the pop-up, and set them for the other.

The platform pop-up list is just a convenience that allows you to have one version of the project even though you're building for two platforms. It doesn't magically build an executable that will run on all the platforms you want. That is, if you're building an application on Windows, you'll get an application that runs on Windows, not on Mach. To get a version that runs on Mach, you'll need to transfer the project directory to a machine running OPENSTEP for Mach and build again.