The makefiles used by ProjectBuilder may be divided into three groups, which will be discussed below.

Project-Type Makefiles

The project-type makefiles contain rules and variables specific to building a particular project type.

Imported Variables

Project-type makefiles assume that the following variables will be provided by **common.make**:

- `PRODUCT_DIR`: An absolute path to the directory where the product should be located.
- `DEPENDENCIES`: A list of all .o files, definition files, and other files required by the product.
- `LOADABLES`: A list of all .o files, definition files, and other files required by the product, correctly formatted for **ld** (for example, if `DEPENDENCIES` contains "`foo.ofileList`" then `LOADABLES` will contain "`-filelist foo.ofileList`").
- `ALL_LDFLAGS`: A list of flags to pass to **ld**.
- `ALL_LIBTOOLFLAGS`: A list of flags to pass to **libtool**.

Exported Variables

Project-type makefiles define the following variables for use by **common.make**:

- `PROJTYPE:` The type of project (`BUNDLE`, `LIBRARY`, etc.).
- `PRODUCTS`: A list of products to be built by the project. All existing project types only define one product, prefixed by `PRODUCT_ROOT`.
- `STRIPPED_PRODUCTS`: A list of products to be stripped during installation. The paths in this variable should match the paths in `PRODUCTS`.
- `PROJTYPE_*FLAGS`: A list of flags to be passed to various tools. For example, **app.make** defines `PROJTYPE_LDFLAGS` to `-win` on Windows.

Exported Targets

Project-type makefiles must define the rules for building their products. These rules are invoked from **common.make**. A sample rule (assuming that `PRODUCTS = $(PRODUCT)`)may look like this:

```
$(PRODUCT):  $(DEPENDENCIES)
  $(LD) -o $@ $(ALL_LDFLAGS) $(LOADABLES)
```

aggregate.make

Aggregate projects contain other projects. None of the standard make targets have any particular meaning in aggregates -- the target is simply applied to all of the subprojects.

app.make
Applications are wrapped projects which contain a launchable executable and a number of resources.

bundle.make
Bundles are wrapped projects which contain a loadable executable and a number of resources.

framework.make
Frameworks are wrapped projects which contain a shared library, header files, and a number of resources. Framework projects may be versioned, whereby a given framework contains numerous historical releases of the library and applications load the version that they were linked against.

library.make
Library projects create either static or shared libraries. Unlike most project types, libraries have more than one destination when they are installed (the library itself, the public header files, and the private header files)

palette.make
Palettes are a special type of bundle used by InterfaceBuilder.

subproj.make
Subprojects are a way of organizing your code during development, but have no effect on the final product (i.e., a resource of a subproject will appear in the same location as a resource of the main project).

tool.make
Tools are standalone executables with no resources. Tools are almost always command-line programs.

Target Makefiles

The target makefiles contain the rules and variables specific to the top-level rules that will be invoked on the project. These rules recurse through the various subprojects, and also invoke earlier top-level rules. The "most final" target is **install**, which results in the following sequence of events:
prebuild is invoked on the current project and on every subproject it contains.
build is invoked on the current project and on every subproject it contains
install is invoked on the current project only
postinstall (found in install.make) is invoked on the current project and on every subproject it contains. Postinstall processing also invokes a nonrecursive **installhdrs** on each project as it processes it.

prebuild.make
The prebuild target is generally invoked implicitly by the build target, but it may be invoked directly as well. The prebuild target creates the resource directories, header file directories, and copies all public/project/private header files into their correct locations.

build.make
The build target is the target which is most commonly used, and is fired by the shorthand "all" target. The build target creates a version of the product in the $(SYMROOT) directory.

installhdrs.make
The installhdrs target is generally invoked as part of the install target, but may be invoked directly. The installhdrs target copies the public and private header files to their appropriate locations in $(DSTROOT).

install.make
The install target is used to install the finished product in $(DSTROOT). The install target implicitly invokes the prebuild, build, and installhdrs targets.

other targets
Two additional targets (sv and clean) are sufficiently simple that they appear in **common.make** rather than having their own files. The clean target deletes temporary files created during the build, and the sv target shows the contents of the variable whose name is in the variable V (i.e. `make sv V=ALL_CFLAGS` will list the compile flags).

Functionality Makefiles

The remaining makefiles are used to provide functionality that is needed by all builds. The following files are functionality makefiles:

flags.make
The flags makefile contains definitions for the flags used to compile and link source files. Clients may add flags via the OTHER and LOCAL variables. For example, OTHER_CFLAGS=-DDEBUG will add -DDEBUG to the compile flags of the current project and all its subprojects. LOCAL_LDFLAGS=-lm will add -lm to the link flags of the current project only.

implicitrules.make
This makefile contains implicit rules for generating object files from C, Objective-C, Objective-C++, and Assembly source files. It also contains implicit rules for generating intermediate source files from a number of specification files (lex, yacc, pswraps, etc.).

tools-NEXTSTEP.make
tools-WINDOWS.make

The tools makefiles contain paths to the various build tools (cc, ld, mv, etc.) on the various architectures.

common.make
The common makefile is the heart of the project makefiles. It includes the target and other functionality rules, and defines most of the internal variables used by the makefiles.

compatibility.make
The compatibility makefile contains variable definitions to provide backward-compatibility with project_makefiles projects.

platform.make
The platform makefile contains a single variable definition which lets the makfiles know what platform they are being used on.

recursion.make
The recursion makefile contains the rules required to build targets in subprojects. The pattern target@directory will change to **directory** and invoke make with **target**. If you want to go deeper, you can reverse-stack the directories (i.e., target@subdirectory@directory). Recursion is only explicitly supported for a small number of targets, but the R variable allows you to recurse on arbitrary rules (i.e., "make countlines@parser.subproj@interpreter.tproj R=countlines").

version.make
The version makefile is used to determine the current project version.

wrapped-common.make
The wrapped-common makefile contains rules common to all projects with wrapped-style products (applications, bundles, palettes, and frameworks).