9 Building

When we build, let us think that we build for ever.
John Ruskin

Well building hath three Conditions: *Commodity*, *Firmness*, and *Delight*.
Sir Henry Wotton, *Elements of Architecture*

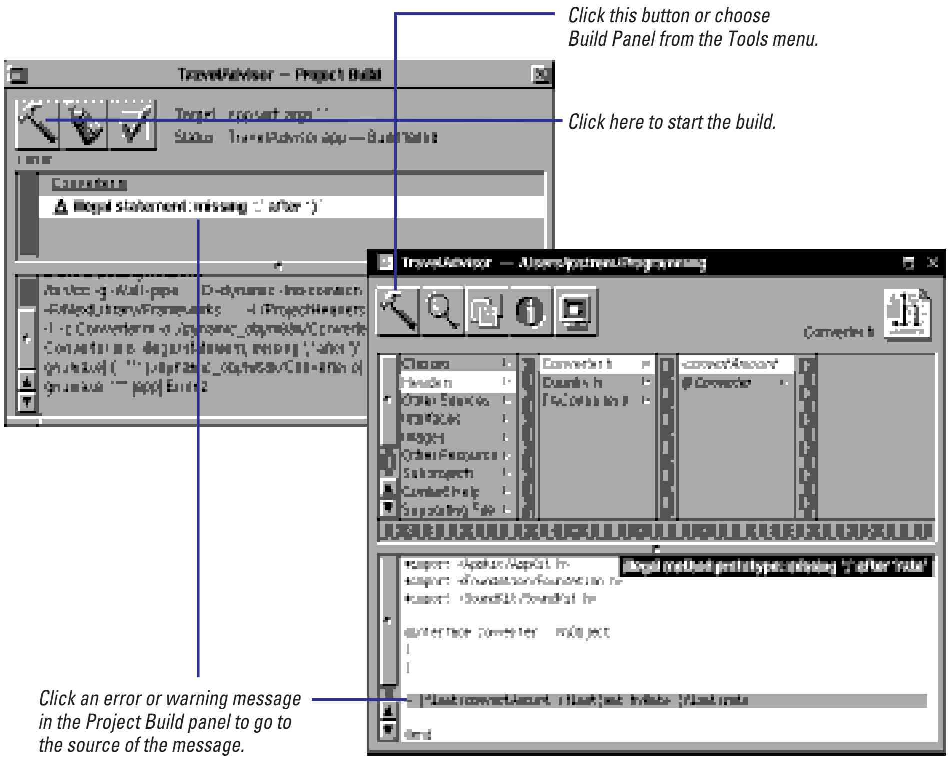
Burrow awhile and build, broad on the roots of things.
Robert Browning, *Abt Vogler*

Building the program

1 **Click the Build button to bring up the Project Build panel.**

2 **In the Project Build panel, click the Build button to start the build.**

When you build a program in Project Builder, you are really invoking the **make** utility to create an executable. The **make** utility invokes the compiler and linker using information from the project makefile.


Click this button or choose
Build Panel from the Tools menu.
Click here to start the build.
TravelAdvisor — Project Build
Illegal statement: missing ';' after ')'
Illegal method prototype: missing ';' after 'rate'
Click an error or warning message
in the Project Build panel to go to
the source of the message.

Before the build begins, Project Builder prompts you to save any unsaved source files, nib files, or the project itself. In this way, Project Builder ensures that you are always building the latest version of the project.

If the build fails because of link errors, look in the bottom browser for more information. The bottom browser shows the exact compiler and linker commands being executed, and it shows all messages produced by these commands. If the Project Build panel appears to only have one browser, drag the split view knob up until you can see the bottom browser.

Before you build, you might want to set specific build options for the project or modify its makefiles. The rest of this chapter describes these tasks.

If the build fails because of link errors, chances are you aren't linking against the correct library or framework. See "Some OPENSTEP Libraries" in this chapter.

All About make and gnumake

make is a standard command-line program 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 "GNU Make" provided by the Free Software Foundation.

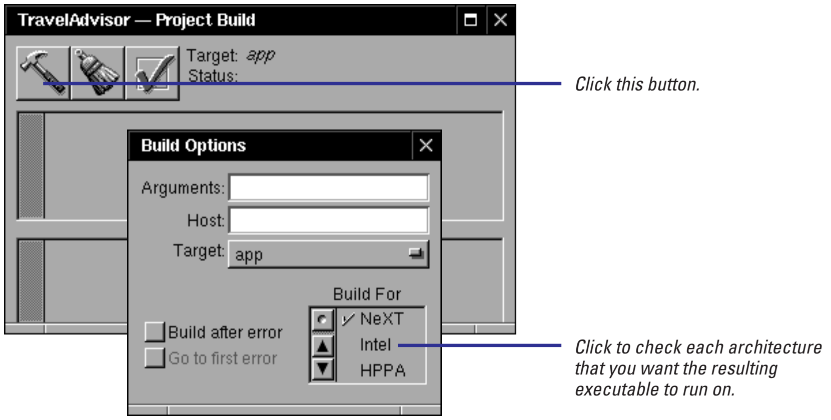
Building for multiple architectures

1 **Click the Build button to bring up the Project Build panel.**

2 **In the Project Build panel, click the check-mark button.**

3 **In the Build Options panel, select the architectures you want to build for.**

Usually when you build, you create an executable that runs only on the type of computer you used. If you build on an Intel computer, the executable will run only on Intel computers. If you want the executable to run on more than one type of computer, you need to set this up in the Build Options panel. You can build executables for any architecture that OPENSTEP currently supports: Intel, NeXT, and SPARC. You must have the libraries for that architecture installed on your computer.



Click this button.

Click to check each architecture that you want the resulting executable to run on.

Portability Do's and Don'ts

If you build multiple architecture ("fat") 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.

Building a Multiple Architecture (“Fat”) Binary

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

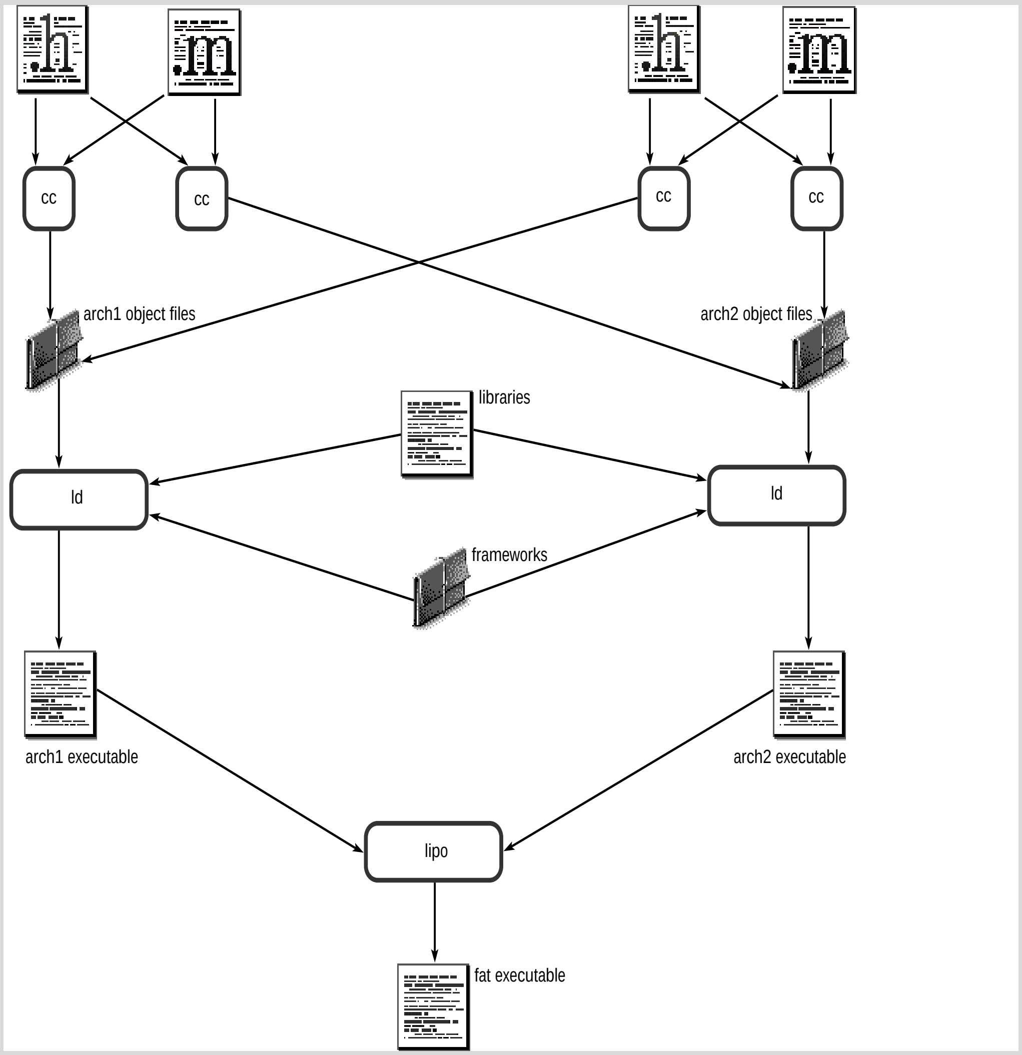
- Each source file is compiled once for each architecture to produce thin object files.

  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*.



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.

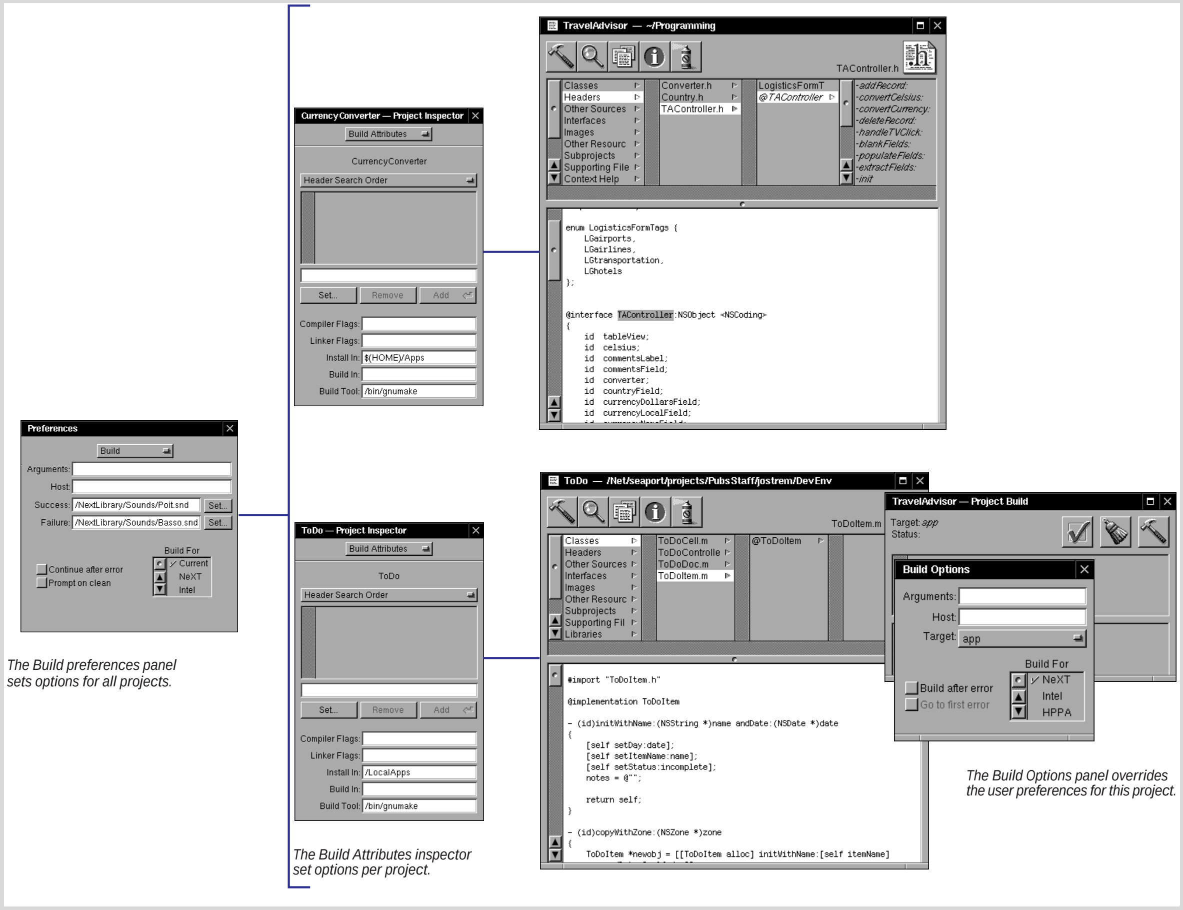
  For example, you may want to change the sound you hear upon each successful build. To open the Build Preferences panel, choose Info ▶ Preferences, then choose Build from the pop-up list in the Preferences panel.

- 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 ▶ Inspector ▶ 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 the 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.



Preferences

Arguments:
Host:
Success: /NextLibrary/Sounds/Poit.snd
Failure: /NextLibrary/Sounds/Basso.snd
Build For: ✓ Current, NeXT, Intel

CurrencyConverter — Project Inspector

Compiler Flags:
Linker Flags:
Install In: $(HOME)/Apps
Build In:
Build Tool: /bin/gnumake

ToDo — Project Inspector

Compiler Flags:
Linker Flags:
Install In: /LocalApps
Build In:
Build Tool: /bin/gnumake

```
enum LogisticsFormTags {
    LGairports,
    LGairlines,
    LGtransportation,
    LGhotels
};


@interface TAController:NSObject <NSCoding>
{
    id  tableView;
    id  celsius;
    id  commentsLabel;
    id  commentsField;
    id  converter;
    id  countryField;
    id  currencyDollarsField;
    id  currencyLocalField;
```

```
#import "ToDoItem.h"

@implementation ToDoItem

- (id)initWithName:(NSString *)name andDate:(NSDate *)date
{
    [self setDay:date];
    [self setItemName:name];
    [self setStatus:incomplete];
    notes = @"";

    return self;
}

- (id)copyWithZone:(NSZone *)zone
{
    ToDoItem *newobj = [[ToDoItem alloc] initWithName:[self itemName]
```

TravelAdvisor — Project Build

Target: *app*

Build Options

Arguments:
Host:
Target: app
Build For: ✓ NeXT, Intel, HPPA

The Build preferences panel sets options for all projects.

The Build Attributes inspector set options per project.

The Build Options panel overrides the user preferences for this project.

Building on a remote computer

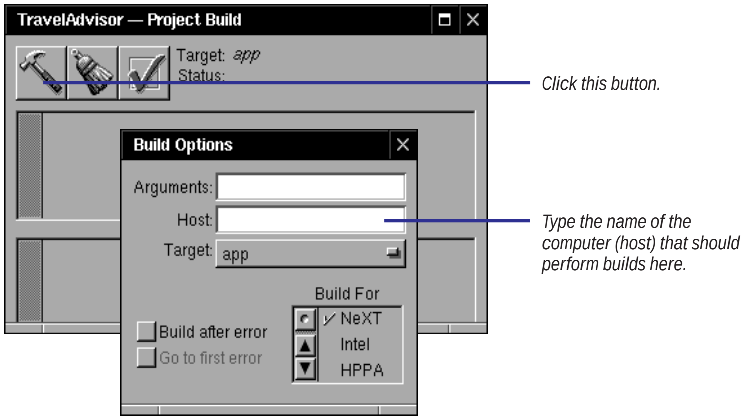
1 Click the Build button to bring up the Project Build panel.

2 In the Project Build panel, click the check-mark button.

3 Type the host name of the computer that should perform the build in the Host field.

4 Build the program.

A build takes up a lot of your computer's CPU time and disk resources. You can still perform other tasks while the build is running, but these other tasks may run slower. If this happens, you may choose to build remotely on another computer on your network. This way, the CPU on your computer can be dedicated to the other tasks you are performing.



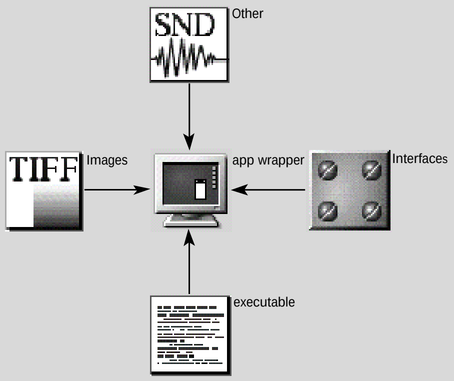
There's no difference in what you see when you build on another host; the Project Build panel still displays the status of the build and updates as the build status changes.

Note: Be sure you know what version of OPENSTEP the host is running before you use it to build your project.

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 "bundle." When the bundle contains an application, it is often called an "app wrapper" because it wraps up all of the things an application needs into a single unit.



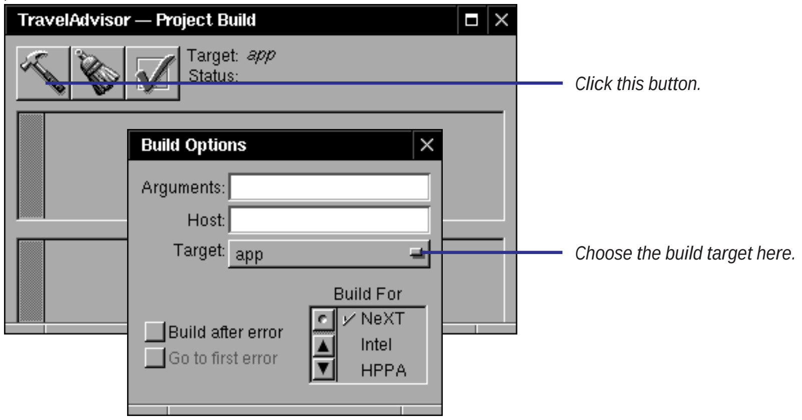
Using build targets

1 **Click the Build button to bring up the Project Build panel.**

2 **In the Project Build panel, click the check-mark button.**

3 **In the Build Options panel, choose a target from the Target pop-up list.**

A *build target* is an argument passed to the **make** utility that tells it which makefile rules to use when building. The default build target, which is named for the project type, produces an optimized, debuggable executable and places it in the project directory. This target is often suitable, so in many cases you don't have to worry about the build target. If you need a different target, choose it from the Build Options panel before you build the project



To do a make clean, click the broom button on the Project Build panel. As described in "All About make and gnumake," make clean is a special target that deletes all object and executable files.

If you're running OPENSTEP on a RISC architecture and you need to debug, you may want to choose a target that does not optimize your code.

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.

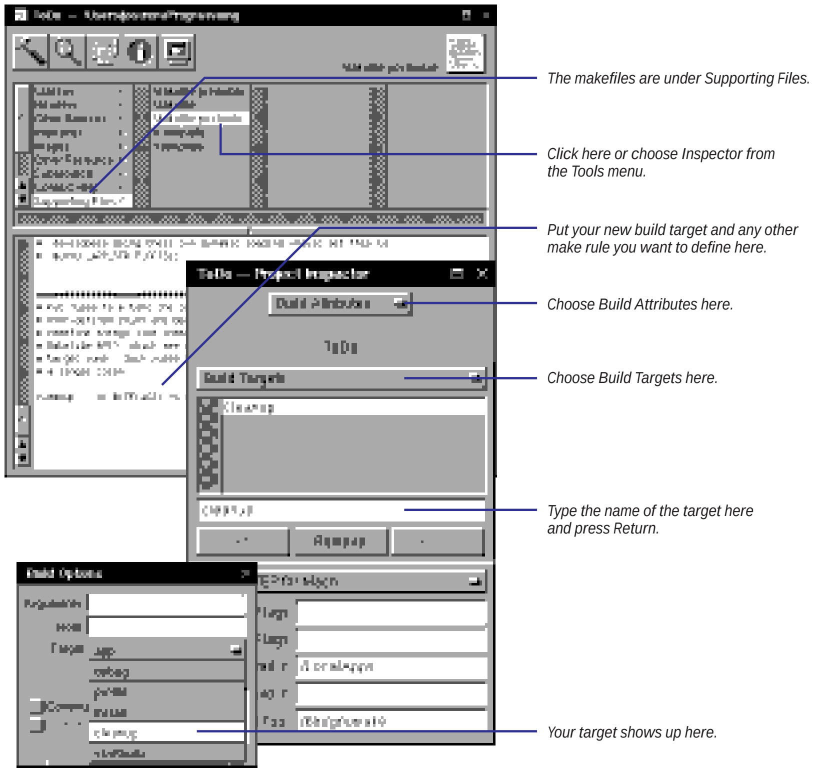
Creating your own build targets

1 **Define a new target in the Makefile.postamble file and save the file.**

2 **Click the Project Inspector button and choose Build Attributes in the Project Inspector panel.**

3 **Type the name of the target in the Build Targets list.**

If the targets provided by Project Builder don't meet your needs, you can define your own target in the file **Makefile.postamble**.


The makefiles are under Supporting Files.
Click here or choose Inspector from the Tools menu.
Put your new build target and any other make rule you want to define here.
Choose Build Attributes here.
Choose Build Targets here.
Type the name of the target here and press Return.
Your target shows up here.
Project Inspector
Build Attributes
Build Targets
Build Options

Project Builder uses the information you specify in the Build Attributes inspector to index the project and to update the project makefile. You'll read about the other fields in the Build Attributes inspector later in this chapter.

If you refer to the executable name in your target, use the **EXECUTABLE_EXT** makefile macro to give it the correct extension. **EXECUTABLE_EXT** is **.exe** in Windows environments and nothing on Rhapsody and supported UNIX environments. For other makefile macros, see the section "Customizing your makefiles" in this chapter.

Caution: Don't use **Makefile.postamble** to redefine targets that are already defined (debug, install, profile, app, and so on). If you do, the results are unpredictable.

After you define a target, you need to let Project Builder know about it so that it appears in the Target pop-up list in the Build Options panel. You do this using the Build Attributes inspector.

Setting search paths

1 **Click the Project Inspector button and choose Build Attributes in the Project Inspector panel.**

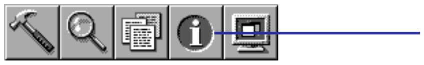
2 **Add the library's location to the Library Search Order list.**

3 **Add the location of its header files to the Header Search Order list.**

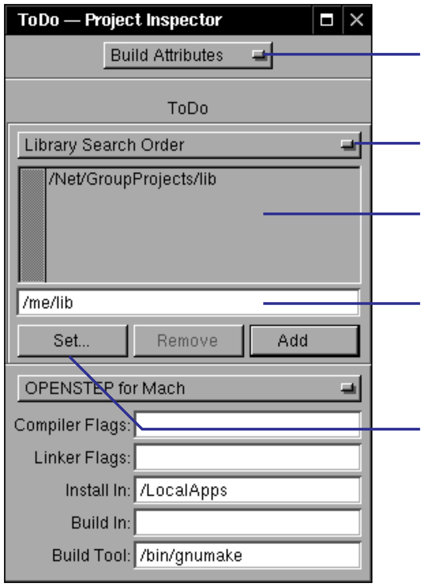
Or

2 **Add the framework's location to the Framework Search Order list.**

The compiler and linker search a standard set of directories for library executables and header files. If you link with a library or framework that is not stored in one of the standard locations, you need to add its location to the search path. You do this from the Build Options inspector.



Click here or choose Inspector from the Tools menu.



Choose Build Attributes.

Choose the appropriate search order list.

Nonstandard directories that are already a part of the search path are displayed here.

Type the directory to be searched here and click Add.

Or click here and choose the directory to be searched from the panel that appears.

The standard search paths are:

Type of file	Search path
Frameworks	/LocalLibrary/Frameworks /NextLibrary/Frameworks
Header files	the project directory /LocalDeveloper/Headers /NextDeveloper/Headers
Libraries	/lib /usr/lib /usr/local/lib

For libraries in nonstandard locations, add the library location to Library Search Order and the header file location to Header Search Order. For frameworks, just add the framework location to Framework Search Order; Project Builder already knows to look inside of a framework for its header files.

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 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.

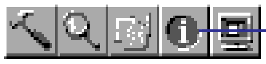
Setting compiler and linker options

1 **Click the Project Inspector button and choose Build Attributes in the Project Inspector panel.**

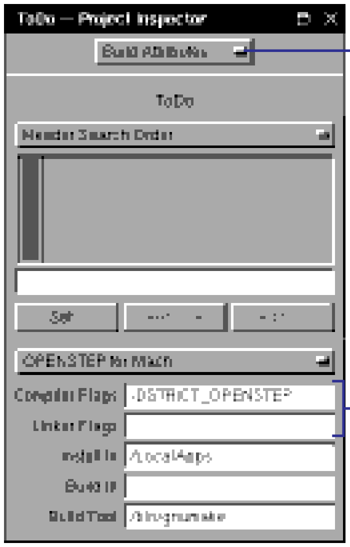
2 **Type compiler options in the Compiler Flags field.**

3 **Type linker options in the Linker Flags field.**

The **make** utility passes the same options to the compiler and linker every time you build a project. You can add to these options using the Build Attributes inspector. The compiler and linker options you specify here are also added to the compiler and linker options used when building the project's subprojects.



Click here or choose Inspector from the Tools menu.



OPENSTEP for Mach

Compiler Flags: -DSTRICT_OPENSTEP

Linker Flags:

Install In: /LocalApps

Build In:

Build Tool: /bin/gnumake

Choose Build Attributes.

Type compiler and linker options in these fields. For library projects, the linker options are passed to ***libtool****.*

The pop-up list above the compiler flags controls the target platform for those flags. You can specify different values for the bottom five options depending on what platform you are building for—Mach, Windows, or a PDO platform. For more information, see "The Platform Pop-Up's Purpose."

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.

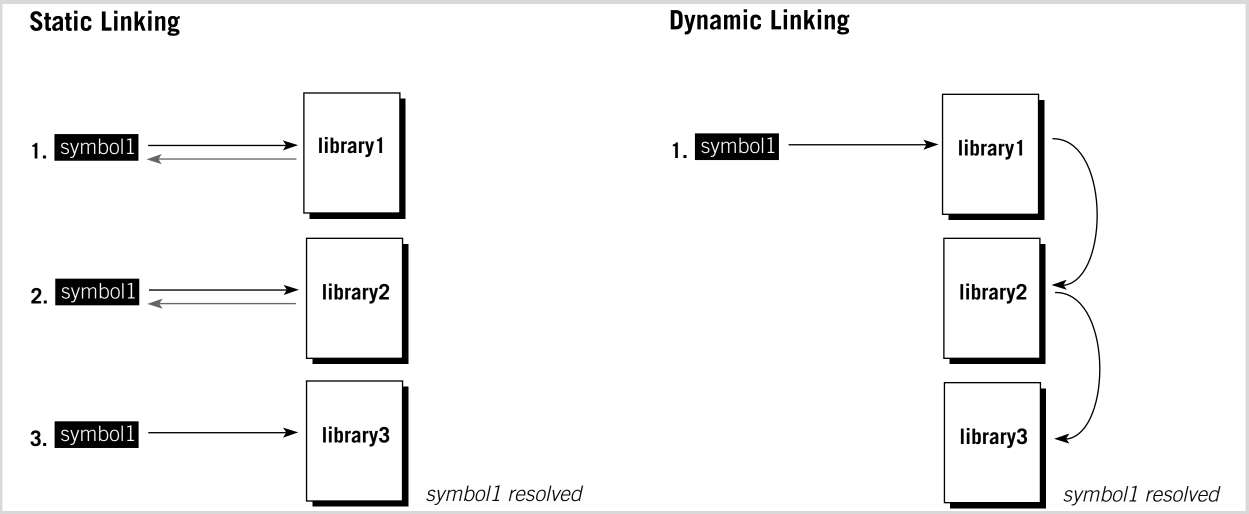
Dynamic Linking

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, "Creating Frameworks and Dynamic Shared Libraries.")

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.



Creating a precompiled header

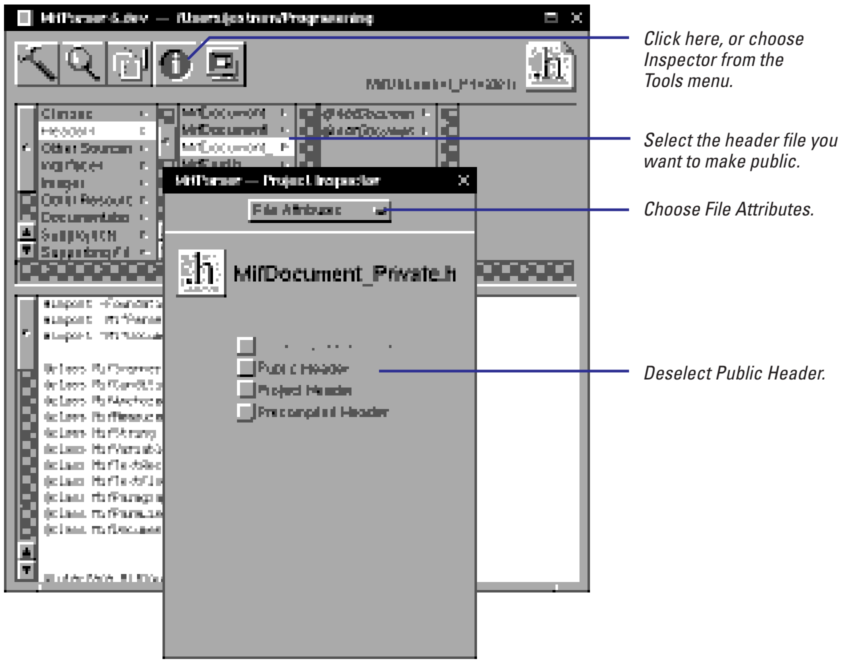
1 **Select the header file in the project browser.**

2 **Click the Inspector button.**

3 **Choose File Attributes in the Inspector panel.**

4 **Select Precompiled Header in the inspector.**

A precompiled header is a header file that has been parsed and preprocessed, thereby improving compile time and reducinng symbol table size. Only those macros and external declarations needed to compile the file are read from the prcompiled header. Precompiled headers have a **.p** extension. You can define one for any type of project.


Click here, or choose Inspector from the Tools menu.
Select the header file you want to make public.
Choose File Attributes.
Deselect Public Header.
MifParser — Project Inspector
File Attributes
MifDocument_Private.h
Public Header
Project Header
Precompiled Header

When you create a header file that you intend to precompile, follow these guidelines:

- Make sure that you import files in the proper order to avoid undefined symbol errors. If ClassA defines an instance variable of the ClassB, ClassB's header should be listed before ClassA's header in the precompiled header.
- Only import the system headers that are necessary for the interface.

  A system header imports many other headers. The more headers you import into your precompiled header, the greater the risk of having a name conflict. For example, a system file might define a public struct that conflicts with a private struct declared in your project's headers. If a name conflict occurs, the compiler won't used the precompiled header.

If you select both Public Header and Precompiled Header in the File Attributes inspector, the **.h** file is installed, not the **.p** file. If you want the **.p** file installed (for example, if you're building a Framework or Library, see "Installing a precompiled header" in Chapter 12, "Creating Frameworks and Dynamic Shared Libraries/DLLs."

To avoid name conflicts, import all of the project's header files into one precompiled header, and import all other header files separately. Make sure that system files that aren't necessary for the interface aren't imported indirectly either.

For example, suppose you have a project with a precompiled header named **Precomp.h** that imports **ClassA.h** along with all other header files in the project. ClassA defines an object that uses Foundation API in its interface declaration, and the implementation of ClassA uses functions from libc. **ClassA.h** should import the Foundation's precompiled header and **Precomp.h**. It should not import **libc.h** because the interface declaration doesn't need it and importing it increases the risk of a name conflict. Instead, **libc.h** should be imported in the implementation file, **ClassA.m**, because it is necessary for the implementation.

ClassA.h

```
#import <Foundation/Foundation.h>
#import "Precomp.h" /* use this for faster compiles */

@interface ClassA : NSObject
{
   NSString *aString;
}
- (NSString *)aString;
...
@end
```

ClassA.m

```
#import "ClassA.h"
#import <libc/libc.h>

@implementation ClassA
...
@end
```

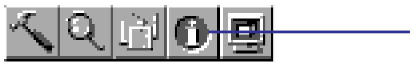
Customizing your makefiles

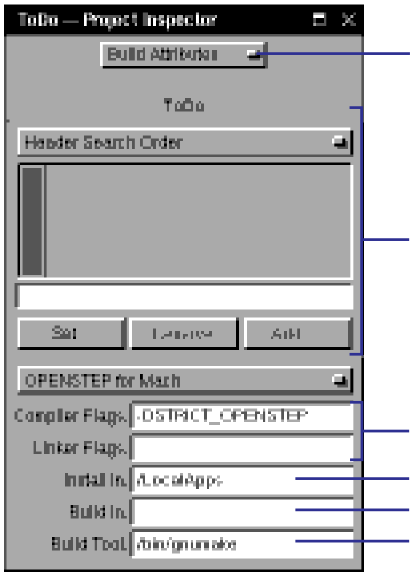
- ▶ **Set information in the Build Attributes inspector.**

  Or

- ▶ **Edit the files Makefile.preamble and Makefile.postamble.**

Sometimes it's necessary to alter the standard build process as defined by the project makefile. If you need to do this, first look for the options you need to change in the Build Attributes inspector. Project Builder uses the information you set in the Build Attributes inspector to index the project. So to make sure the project index is correct, you should use this inspector instead of editing the makefile directly. (Project Builder updates the makefile for you.)




ToDo — Project Inspector
Build Attributes
ToDo
Header Search Order
Set
Remove
Add
OPENSTEP for Mach
Choose Build Attributes.
Search paths for files in nonstandard locations.
Options for the **cc, ld,** and ***libtool*** commands.
Where the executable lives after it is installed.
Where intermediate files (such as **.o** files) are placed.
Path for the **make** utility.

Compiler Flags: -DSTRICT_OPENSTEP
Linker Flags:
Install In: /LocalApps
Build In:
Build Tool: /bin/gnumake

If the inspector does not have an option for what you need to set, edit the files **Makefile.preamble** and **Makefile.postamble**. Both files contain makefile variable definitions, and the comments in these files describe what each macro defines.

"Creating your own build targets," "Setting search paths," and "Setting compiler and linker options" in this chapter describe some of the fields in the Build Attributes inspector.

In general, **Makefile.preamble** contains macros that add to the standard makefile definitions, and **Makefile.postamble** contains macros that override the standard definitions. For example, the **LIBS** macro defines the libraries that your program should link with. The standard makefile sets this macro to the libraries in the project. If you want to change this definition, you uncomment the **LIBS** definition in **Makefile.postamble** and change the definition. However, if you want to link with more libraries than those added to the project, set **OTHER_LIBS** in **Makefile.preamble** to the additional library's name.

Reducing Compile Time

Each build begins by exporting any public headers to a location where they are visible to the rest of the project (for example, headers in subprojects are exported to the **derived_src** directory if you mark them as Project Headers in the File Attributes inspector). If you're building a project that does not export any headers (no boxes are turned on for header files in the File Attributes inspector), such as an application or tool with no distributed objects or library API, you can omit this step by setting this macro in **Makefile.preamble**:

Preamble Macro	Description
SKIP_EXPORTING_HEADERS	Skips the exporting headers step of the build.

Adding Make Dependencies

If you add dependencies or targets to the makefile, set these macros in **Makefile.preamble**.

Preamble Macro	Description
OTHER_PRODUCT_DEPENDS	Dependencies you defined that should be used in all builds.
OTHER_INITIAL_TARGETS	Targets you defined that should be built before subprojects.
OTHER_INSTALL_DEPENDS	Dependencies you defined that should be used for the install target.

Setting Up The Install Target

To set up the install target to work the way you want, set these macros in **Makefile.preamble**.

If you're building a library or framework, see Chapter 12, "Creating Frameworks and Dynamic Shared Libraries," for more information on setting up the makefiles.

Preamble Macro	Description
DSTROOT	Path to prepend to the installation path specified in the Build Attributes inspector. The default is **/**.

And these macros in **Makefile.postamble.**

Postamble Macro	Description
INSTALL_AS_USER	The owner of the installed files. The default is root.
INSTALL_AS_GROUP	The group for the installed files. The default is wheel.
INSTALL_PERMISSIONS	The installed files' permissions. The default is read and execute permissions turned on for all users.
APP_STRIP_OPTS	Stripping options to pass to the **strip** tool. The default is no options, which strips debugging symbols out of the executable. Only set options here if the application loads other bundles.
APP_WRAPPER_EXTENSION	The extension to use for application's output. The default is **.app**.

Setting Up Make Clean

To set up **make clean** to work the way you want, set these macros in **Makefile.preamble.**

Preamble Macro	Description
OTHER_GARBAGE	Files that should be deleted in addition to object files and executables.
CLEAN_ALL_SUBPROJECTS	If defined, **make clean** cleans subprojects as well. This macro is defined by default. To undefine it, comment it out in **Makefile.preamble**.

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.

Overriding Compiler and Linker Options

Most targets produce an optimized, debuggable executable and do not suppress compiler warnings. To change the compiler options used to produce the usual executable, override these macros in **Makefile.postamble**.

Postamble Macro	Description
OPTIMIZATION_CFLAG	Compiler optimization option, used by all but the debug target. The default is **-O**, which reduces code size and execution time.
LOCAL_DIR_INCLUDE_DIRECTIVE	Override if you don't want the current directory in the default search path. By default, this is defined as **-I.**
DEBUG_SYMBOLS_CFLAG	Compiler debug symbols options, used by all but the install target. The default is **-g**, which produces line number and symbol information.
WARNING_CFLAGS	Compiler warning message level, used by all targets. The default is **-Wall**, which suppresses none of the warning messages.
DEBUG_BUILD_CFLAGS	Compiler options used only by the debug target. The default is **-DDEBUG**, which defines the **DEBUG** preprocessor macro.
PROFILE_BUILD_CFLAGS	Compiler options used only by the profile target. The default is **-pg**, which produces information for **gprof**, and **-DPROFILE**, which defines the **PROFILE** preprocessor macro.

Setting Up Other Tools

Some tools are invoked by **make** if the project contains files with certain extensions. To set up these tools, set these macros in **Makefile.preamble**.

For more information about the tools listed here, see their man pages.

Preamble Macro	Description
PSWFLAGS	Options for the **pswrap** tool (invoked on **.psw** files).
YFLAGS	Options for the **yacc** tool (invoked on **.y.c** or **.ym.m** files).
LFLAGS	Options for the **lex** tool (invoked on **.l.c** or **.lm.m** files).
MSGFILES	Input files for the **msgwrap** tool. These should have the **.msg** extension.
DEFSFILES	Input files with a **.defs** extension for the **mig** tool.
MIGFILES	Input files with a **.mig** extension for the **mig** tool.
RPCFILES	Input files for the **rpcgen** tool (invoked on **.x** files).

Including More Files in the Build

There may be files that you don't want to add to the project but that should be included in the build. Use these macros in **Makefile.preamble** to have the build handle more files.

Preamble Macro	Description
OTHER_LIBS	Libraries to link with besides the libraries included in the project.
OTHER_OFILES	Object files to link into the executable besides those produced by the source files in the project.
OTHER_SOURCEFILES	Source files besides those included in the project.
INCLUDED_ARCHS	Architectures to which this project or subproject should be restricted to building for. Building for other architectures is skipped. Must be a subset of the architectures selected in the Build Options panel.
EXCLUDED_ARCHS	Similar to INCLUDED_ARCHS, but lists architectures that this project or subproject shouldn't build for instead of architectures it should build for. Don't use if using INCLUDED_ARCHS.