GNUstep Makefile Package

Table of Contents

1 Makefile Package

1.1 Introduction

The Makefile package is a system of make commands that is designed to encapsulate all the complex details of building and installing various types of projects from libraries to applications to documentation. This frees the developer to focus on the details of their particular project. Only a fairly simple main makefile need to be written which specifies the type of project and files involved in the project.

1.2 Structure of a Makefile

Here is an example makefile (named GNUmakefile to emphasis the fact that it relies on special features of the GNU make program).

```
#
# An example GNUmakefile
#

# Include the common variables defined by the Makefile Package
include $(GNUSTEP_MAKEFILES)/common.make

# Build a simple Objective-C program
OBJC_PROGRAM_NAME = simple

# The Objective-C files to compile
simple_OBJC_FILES = simple.m

-include GNUmakefile.preamble

# Include in the rules for making Objective-C programs
include $(GNUSTEP_MAKEFILES)/objc.make

-include GNUmakefile.postamble
```

This is all that is necessary to define the project.

1.3 Running Make

Normally to compile a package which uses the Makefile Package it is purely a matter of typing `make` from the top-level directory of the package, and the package is compiled without any additional interaction.

1.3.1 Debug Information

By default the Makefile Package does not tell the compiler to generate debugging information when compiling Objective-C and C files. The following command illustrates how to tell the Makefile Package to pass the appropriate flags to the compiler so that debugging information is put into the binary files.

```
make debug=yes
```

When debugging is turned on, the Makefile Package turns off optimization so the user must override the optimization flag when running make if both debugging information and optimization is to be performed by the compiler. Use the variable *OPTFLAG* to override the optimization flag.

1.3.2 Profile Information

By default the Makefile Package does not tell the compiler to generate profiling information when compiling Objective-C and C files. The following command illustrates how to tell the Makefile Package to pass the appropriate flags to the compiler so that profiling information is put into the binary files.

```
make profile=yes
```

1.3.3 Static, Shared, and Dynamic Link Libraries

By default the Makefile Package will generate a shared library if it is building a library project type, and it will link with shared libraries if it is building an application or command line tool project type. The following command illustrates how to tell the Makefile Package not to build using shared libraries but using static libraries instead.

```
make shared=no
```

This default is only applicable on systems that support shared libraries; systems that do not support shared libraries will always build using static libraries. Some systems support dynamic link libraries (DLL) which are a form of shared libraries; on these systems, DLLs will be built by default unless the Makefile Package is told to build using static libraries instead, as in the above command.

1.4 Project Types

Projects are divided into different types described below. To create a project of a specific type, just include the particular makefile. For example, to create an application, include this line in your make make file:

```
include $(GNUSTEP_MAKEFILES)/application.make
```

Each project type is independent of the others. If you want to create two different types of projects within the same directory (e.g. a tool and a java program), include both the desired makefiles in your main make file.

1.4.1 Aggregate (`aggregate.make`)

An Aggregate project is a project that consists of several subprojects. Each subproject can be of any other valid project type (including the Aggregate type). The only project variable is the SUBPROJECTS variable

SUBPROJECTS [Aggregate project]

`SUBPROJECTS` defines the directory names that hold the subprojects that the Aggregate project should build.

1.4.2 Graphical Applications (`application.make`)

An application is an Objective-C program that includes a GUI component, and by default links in all the GNUstep libraries required for GUI development, such as the Base and Gui libraries.

1.4.3 Bundles (`bundle.make`)

A bundle is a collection of resources and code that can be used to enhance an existing application or tool dynamically using the NSBundle class from the GNUstep base library.

1.4.4 Command Line C Tools (`ctool.make`)

A ctool is a project that only uses C language files. Otherwise it is similar to the ObjC project type.

1.4.5 Documentation (`documentation.make`)

The Documentation project provides rules to use various types of documentation such as texi and LaTeX documentation, and convert them into finished documentation (info, PostScript, HTML, etc).

1.4.6 Frameworks (`framework.make`)

A Framework is a collection of resources and a library that provides common code that can be linked into a Tool or Application. In many respects it is similar to a Bundle.

1.4.7 Java (‘java.make’)

This project provides rules for building java programs. It also makes it easy to make java projects that interact with the GNUstep libraries.

1.4.8 Libraries (‘library.make’)

The Makefile Package provides a project type for building libraries; libraries can be built as static libraries, shared libraries, or dynamic link libraries (DLL) if the platform supports that type of library. Static libraries are supported on all platforms; while, shared libraries and DLLs are only supported on some platforms.

1.4.8.1 Project Variables

LIBRARY_NAME [Library project]

`LIBRARY_NAME` should be assigned the list of name of libraries to be generated. Most UNIX systems expect that the filename for the library has the word ‘`lib`’ prefixed to the name; i.e. the ‘`c`’ library has filename of ‘`libc`’. Prefix the ‘`lib`’ to the library name when it is specified in the `LIBRARY_NAME` variable because the Makefile Package will not automatically prefix it.

C_FILES [Library project]

`xxx_C_FILES` is the list of C files, with a ‘`.c`’ extension, that are to be compiled to generate the **xxx** library. Replace the **xxx** with the name of the library as listed by the `LIBRARY_NAME` variable.

OBJC_FILES [Library project]

`xxx_OBJC_FILES` is the list of Objective-C files, with a ‘`.m`’ extension, that are to be compiled to generate the **xxx** library. Replace the **xxx** with the name of the library as listed by the `LIBRARY_NAME` variable.

PSWRAP_FILES [Library project]

`xxx_PSWRAP_FILES` is the list of PostScript wrap files, with a ‘`.psw`’ extension, that are to be compiled to generate the **xxx** library. PostScript wrap files are processed by the ‘`pswrap`’ utility which generates a ‘`.c`’ and a ‘`.h`’ file from each ‘`.psw`’ file; the generate ‘`.c`’ file is the file actually compiled. Replace the **xxx** with the name of the library as listed by the `LIBRARY_NAME` variable.

HEADER_FILES [Library project]

`xxx_HEADER_FILES` is the list of header filenames that are to be installed with the library. If a filename has a directory path prefixed to it then that prefix will be maintained when the headers are installed. It is up to the user to make sure that the installation directory exists; otherwise, an error will occur when the library is installed, see Section 1.4.8.1

[xxx_HEADER_FILES_INSTALL_DIR], page 4. Replace the **xxx** with the name of the library as listed by the `LIBRARY_NAME` variable.

HEADER_FILES_DIR [Library project]

`xxx_HEADER_FILES_DIR` is the relative path from the current directory, where the makefile is located, to where the header files specified by `xxx_HEADER_FILES` are located. If a filename specified in `xxx_HEADER_FILES` has a directory path prefixed to it then that path will not be removed when the Makefile Package accesses the files, so do not specify a path with `xxx_HEADER_FILES_DIR` that is already prefixed to the header filenames, see Section 1.4.8.1 [xxx_HEADER_FILES_INSTALL_DIR], page 4. `xxx_HEADER_FILES_DIR` is optional; leaving it blank or undefined, and the Makefile Package assumes that the relative path to the header files is the current directory where the makefile resides. Replace the **xxx** with the name of the library as listed by the `LIBRARY_NAME` variable.

HEADER_FILES_INSTALL_DIR [Library project]

`xxx_HEADER_FILES_INSTALL_DIR` specifies the relative subdirectory path below `GNUSTEP_HEADERS` where the header files are to be installed. If this directory or any of its parent directories do not exist, then the Makefile Package will create them. The Makefile Package prefixes `xxx_HEADER_FILES_INSTALL_DIR` to each of the filenames in `xxx_HEADER_FILES` when they are installed, so if the filenames in `xxx_HEADER_FILES` already have a directory path prefixed then the user is responsible for creating that directory, the Makefile Package will not create. `xxx_HEADER_FILES_INSTALL_DIR` is optional; leaving it blank or undefined, and the Makefile Package assumes that the installation directory is just `GNUSTEP_HEADERS` with no subdirectory. Replace the **xxx** with the name of the library as listed by the `LIBRARY_NAME` variable.

CPPFLAGS [Library project]

`xxx_CPPFLAGS` are additional flags that will be passed to the compiler preprocessor when compiling Objective-C and C files to generate the **xxx** library. Adding flags here does not override the default `CPPFLAGS`, see Section 1.7.5 [CPPFLAGS], page 22, they are in addition to `CPPFLAGS`. These flags are specific to the **xxx** library, see Section 1.5 [ADDITIONAL_CPPFLAGS], page 11, to see how to specify global preprocessor flags. Replace the **xxx** with the name of the listed as listed by the `LIBRARY_NAME` variable.

OBJCFLAGS [Library project]

`xxx_OBJCFLAGS` are additional flags that will be passed to the compiler when compiling Objective-C files to generate the **xxx** library. Adding flags here does not override the default `OBJCFLAGS`, see Section 1.7.5 [OBJCFLAGS], page 22, they are in addition to `OBJCFLAGS`. These flags are specific to the **xxx** library, see Section 1.5 [ADDITIONAL_OBJCFLAGS], page 11, to see how to specify global compiler flags. Replace the **xxx** with the name of the library as listed by the `LIBRARY_NAME` variable.

CFLAGS [Library project]

xxx_CFLAGS are additional flags that will be passed to the compiler when compiling C files to generate the **xxx** library. Adding flags here does not override the default CFLAGS, see Section 1.7.5 [CFLAGS], page 22, they are in addition to CFLAGS. These flags are specific to the **xxx** library, see Section 1.5 [ADDITIONAL_CFLAGS], page 11, to see how to specify global compiler flags. Replace the **xxx** with the name of the library as listed by the LIBRARY_NAME variable.

LDFLAGS [Library project]

xxx_LDFLAGS are additional flags that will be passed to the linker when it creates the **xxx** library. Adding flags here does not override the default LDFLAGS, see Section 1.7.5 [LDFLAGS], page 22, they are in addition to LDFLAGS. These flags are specific to the **xxx** library, see Section 1.5 [ADDITIONAL_LDFLAGS], page 11, to see how to specify global linker flags. Replace the **xxx** with the name of the library as listed by the LIBRARY_NAME variable.

INCLUDE_DIRS [Library project]

xxx_INCLUDE_DIRS is the list of additional directories that the compiler will search when it is looking for include files; these flags are specific to the **xxx** library, see Section 1.5 [ADDITIONAL_INCLUDE_DIRS], page 11, to see how to specify additional global include directories. The directories should be specified as '-I' flags to the compiler. The additional include directories will be placed before the normal GNUstep and system include directories, and before any global include directories specified with ADDITIONAL_INCLUDE_DIRS, so they will always be searched first. Replace the **xxx** with the name of the library as listed by the LIBRARY_NAME variable.

1.4.8.2 Example Makefile

This example makefile illustrates two libraries, 'libone' and 'libtwo', that are to be generated.

```
#
# An example makefile
#

# Include the common variables defined by the Makefile Package
include $(GNUSTEP_MAKEFILES)/common.make

# Two libraries
LIBRARY_NAME = libone libtwo

#
# The files for the libone library
#
```

```
# The Objective-C files to compile
libone_OBJC_FILES = one.m draw.m

# The C source files to be compiled
libone_C_FILES = parse.c

# The PostScript wrap source files to be compiled
libone_PSWRAP_FILES = drawing.psw

# The header files for the library
libone_HEADER_FILES_DIR = ./one
libone_HEADER_FILES_INSTALL_DIR = one
libone_HEADER_FILES = one.h draw.h

#
# The files for the libtwo library
#
# The Objective-C files to compile
libtwo_OBJC_FILES = two.m another.m test.m

# The header files for the library
libtwo_HEADER_FILES_DIR = ./two
libtwo_HEADER_FILES_INSTALL_DIR = two
libtwo_HEADER_FILES = two.h another.h test.h common.h

# Option include to set any additional variables
-include GNUmakefile.preamble

# Include in the rules for making libraries
include $(GNUSTEP_MAKEFILES)/library.make

# Option include to define any additional rules
-include GNUmakefile.postamble
```

Notice that the '`libone`' library has Objective-C, C, and PostScript wrap files to be compiled; while, the '`libtwo`' library only has some Objective-C files.

The header files for the '`libone`' library reside in the '`one`' subdirectory from where the sources are located, and the header files will be installed into the '`one`' subdirectory within `GNUSTEP_HEADERS`. Likewise the header files for the '`libtwo`' library reside in the '`two`' subdirectory from where the sources are located, and the header files will be installed into the '`two`' subdirectory within `GNUSTEP_HEADERS`.

1.4.9 Native Library ('`native-library.make`')

A "native library" is a project which is to be built as a shared library on most targets and as a framework on Darwin. (Currently this is only the case for apple-apple-apple.) In other words, it is to be built as the most appropriate native equivalent of a traditional shared library (see Section 1.4.8 [library.make], page 4 and Section 1.4.6 [framework.make], page 3).

NATIVE_LIBRARY_NAME [Native Library project]

`NATIVE_LIBRARY_NAME` should be the name of the native library, without the 'lib'. All the other variables are the same as the ones used in libraries and frameworks.

To compile something against a native library, you can use `ADDITIONAL_NATIVE_LIBS += MyLibrary` This will be converted into -lMyLibrary link flag on for most targets and into -framework MyLibrary link flag for apple-apple-apple.

1.4.10 Objective-C Programs ('objc.make')

The Makefile Package provides a project type that is useful for building Objective-C programs that do not depend upon the GNUstep libraries. Objective-C programs which only use the Objective-C Runtime Library and the classes it defines are candidates for this project type.

1.4.10.1 Project Variables

Most of the project variables work the same as in Library projects (see Section 1.4.8 [library.make], page 4).

OBJC_PROGRAM_NAME [Objective-C program project]

`OBJC_PROGRAM_NAME` is the list of names of Objective-C programs that are to be built; each name should be unique as it is the name of the executable file that will be generated.

OBJC_LIBS [Objective-C program project]

`xxx_OBJC_LIBS` is the list of additional libraries that the linker will use when linking to create the **xxx** Objective-C program executable file. These libraries are specific to the **xxx** Objective-C program, see Section 1.5 [ADDITIONAL_OBJC_LIBS], page 11, to see how to specify additional global libraries. These libraries are placed before all of the Objective-C Runtime and system libraries, and before the global libraries specified with `ADDITIONAL_OBJC_LIBS`, so that they will be searched first when linking. The additional libraries should be specified as '`-l`' flags to the linker as the following example illustrates. Replace the **xxx** with the name of the program as listed by the `OBJC_PROGRAM_NAME` variable.

1.4.10.2 Example Makefile

This makefile illustrates two Objective-C programs, '`simple`' and '`list`' that are to be generated.

```
#
# An example makefile
#

# Include the common variables defined by the Makefile Package
include $(GNUSTEP_MAKEFILES)/common.make

# Build a simple Objective-C program
OBJC_PROGRAM_NAME = simple list

# Have the Objective-C runtime macro be defined for simple program
simple_CPPFLAGS = $(RUNTIME_DEFINE)

# The Objective-C files to compile for simple program
simple_OBJC_FILES = simple.m

# The Objective-C files to compile for list program
list_OBJC_FILES = list.m linkedlist.m

# The C files to compile for list program
list_C_FILES = sort.c

# Option include to set any additional variables
-include GNUmakefile.preamble

# Include in the rules for making Objective-C programs
include $(GNUSTEP_MAKEFILES)/objc.make

# Option include to define any additional rules
-include GNUmakefile.postamble
```

The '`simple`' Objective-C program only consists of single Objective-C file; while, the '`list`' Objective-C program consists of two Objective-C files and one C file. The '`simple`' Objective-C program use the variable defined by the Makefile Package, `RUNTIME_DEFINE`, to define a macro based upon the Objective-C Runtime library; presumably '`simple.m`' has code which is dependent upon the Objective-C Runtime.

1.4.11 Palettes ('`palette.make`')

A palette is a Bundle that provides some kind of GUI functionality. Otherwise it is similar to the Bundle project.

1.4.12 RPMs ('`rpm.make`')

The RPM project provides rules for automatically generating RPM spec files in order to make RPM distributions. Note that this project makefile is included automatically when you include any other project type in your GNUmakefile. It is non necessary to include '`rpm.make`'.

Except for PACKAGE_NAME, which is required, all the following variables are optional. It is recommended that you set them anyway in order to provide the standard information that is present in most RPM distributions.

PACKAGE_NAME [RPM]

PACKAGE_NAME defines the name of the RPM distribution. In most cases this will be the same as the name of your project type. For instance, if you are creating a application, and have set APP_NAME to 'MyApplication', Then set PACKAGE_NAME to the same thing, or just use PACKAGE_NAME=$(APP_NAME). if PACKAGE_NAME is not set, it defaults to unnamed-package

PACKAGE_VERSION [RPM]

Set PACKAGE_VERSION to the release version number of your package. If not set, it defaults to 0.0.1

GNUSTEP_INSTALLATION_DIR [RPM]

Set GNUSTEP_INSTALLATION_DIR to the installation directory. Typically this is either $(GNUSTEP_SYSTEM_ROOT)), $(GNUSTEP_LOCAL_ROOT), or $(GNUSTEP_USER_ROOT). If not set it defaults to $(GNUSTEP_LOCAL_ROOT).

RPM_DISABLE_RELOCATABLE [RPM]

Set this to YES if the package must be in $(GNUSTEP_SYSTEM_ROOT) and is not relocatable.

PACKAGE_NEEDS_CONFIGURE [RPM]

Set this to YES if a configure script needs to be run before compilation

In addition you need to provide a stub spec file named for the package name, such as this example 'libobjc.spec.in' file:

```
Release:        1
Source:         ftp://ftp.gnustep.org/pub/gnustep/libs/%{gs_name}-%{gs_version}.
tar.gz
Copyright:      GPL
Group:          Development/Libraries
Summary:        Objective-C Runtime Library
Packager:       Adam Fedor <fedor@gnu.org>
Vendor:         The GNUstep Project
URL:            http://www.gnustep.org/

%description
Library containing the Objective-C runtime.
```

1.4.13 Services ('service.make')

A Service is like a Tool that provides a service to a running GNUstep program.

1.4.14 Subprojects ('subproject.make')

A Subproject provides a way to organize code in a large application into subunits. The code in the subproject is merged in with the main tool or application.

1.4.15 Command Line Tools ('tool.make')

A tool is an ObjC project that by default links in the GNUstep base library. Otherwise it is similar to the ObjC project type.

1.5 Global Variables ('GNUmakefile.preamble')

'GNUmakefile.preamble' is an optional file that may be put within the package for declaring global makefile variables for the package. The filename, 'GNUmakefile.preamble', is just a convention; likewise, the variables defined within it can be put in the normal 'GNUmakefile' versus in this special file. However, the reason for this convention is that the 'GNUmakefile' may be automatically maintained by a project management system, like Project Center, so any changes made to 'GNUmakefile' may be discarded by that project management system.

The file, 'GNUmakefile.preamble', in the Makefile Package is a template that can be used the project's 'GNUmakefile.preamble'. It is not necessary to have a 'GNUmakefile.preamble' with the project unless it is actually needed, the Makefile Package will only include it if it is available, see Section 1.2 [Makefile Structure], page 1 for information on how the Makefile Package includes a 'GNUmakefile.preamble'.

The rest of this section describes the individual global variables that the Makefile Package uses which are generally placed in the package's 'GNUmakefile.preamble'.

ADDITIONAL_CPPFLAGS [Variable]

`ADDITIONAL_CPPFLAGS` are additional flags that will be passed to the compiler preprocessor. Generally any macros to be defined for all files are placed here; the are passed for both Objective-C and C files that are compiled. `RUNTIME_DEFINE`, `FOUNDATION_DEFINE`, `GUI_DEFINE`, and `GUI_BACKEND_DEFINE` are some makefile variables which define macros that can be assigned to `ADDITIONAL_CPPFLAGS`. The following example illustrates the use of `ADDITIONAL_CPPFLAGS` to define a macro for the Objective-C Runtime Library plus an additional macro that is specific to the package.

```
ADDITIONAL_CPPFLAGS = $(RUNTIME_DEFINE) -DVERBOSE=1
```

ADDITIONAL_OBJCFLAGS [Variable]

ADDITIONAL_OBJCFLAGS are additional flags that will be passed to the compiler when compiling Objective-C files. Adding flags here does not override the default OBJCFLAGS, see Section 1.7.5 [OBJCFLAGS], page 22, they are in addition to OBJCFLAGS. Generally ADDITIONAL_OBJCFLAGS are placed before OBJCFLAGS when the compiler is executed, but one should avoid having any placement sensitive flags because the order of the flags is not guaranteed. The following example illustrates how you can pass additional Objective-C flags.

```
ADDITIONAL_OBJCFLAGS = -Wno-protocol
```

ADDITIONAL_CFLAGS [Variable]

ADDITIONAL_CFLAGS are additional flags that will be passed to the compiler when compiling C files. Adding flags here does not override the default CFLAGS, see Section 1.7.5 [CFLAGS], page 22, they are in addition to CFLAGS. Generally ADDITIONAL_CFLAGS are placed before CFLAGS when the compiler is executed, but one should avoid having any placement sensitive flags because the order of the flags is not guaranteed. The following example illustrates how you can pass additional C flags.

```
ADDITIONAL_CFLAGS = -finline-functions
```

ADDITIONAL_LDFLAGS [Variable]

ADDITIONAL_LDFLAGS are additional flags that will be passed to the linker when it creates an executable; these flags are passed when linking a command line tool, and application, or an Objective-C program. Adding flags here does not override the default LDFLAGS, see Section 1.7.5 [LDFLAGS], page 22, they are in addition to LDFLAGS. Generally ADDITIONAL_LDFLAGS are placed before LDFLAGS when the linker is executed, but one should avoid having any placement sensitive flags because the order of the flags is not guaranteed. The following example illustrates how you can pass addition linker flags.

```
ADDITIONAL_LDFLAGS = -v
```

ADDITIONAL_INCLUDE_DIRS [Variable]

ADDITIONAL_INCLUDE_DIRS is the list of additional directories that the compiler will search when it is looking for include files. The directories should be specified as '-I' flags to the compiler. The additional include directories will be placed before the normal GNUstep and system include directories, so they will always be searched first. The following example illustrates two additional include directories; /usr/local/gnu/include will be searched first, then /usr/gnu/include, and finally the GNUstep and system directories which are automatically defined by the Makefile Package.

```
ADDITIONAL_INCLUDE_DIRS = -I/usr/local/gnu/include -I/usr/gnu/include
```

ADDITIONAL_LIB_DIRS [Variable]

`ADDITIONAL_LIB_DIRS` is the list of additional directories that the linker will search when it is looking for library files. The directories should be specified as '`-L`' flags to the linker. The additional library directories will be placed before the GNUstep and system library directories so that they will be searched first by the linker. The following example illustrates two additional library directories; `/usr/local/gnu/lib` will be searched first, then `/usr/gnu/lib`, and finally the GNUstep and system directories which are automatically defined by the Makefile Package.

```
ADDITIONAL_LIB_DIRS = -L/usr/local/gnu/lib -L/usr/gnu/lib
```

ADDITIONAL_OBJC_LIBS [Variable]

`ADDITIONAL_OBJC_LIBS` is the list of additional libraries that the linker will use when linking command line tools, applications, and Objective-C programs, see Section 1.4.15 [tool.make], page 11, Section 1.4.2 [application.make], page 3, and Section 1.4.10 [objc.make], page 8. For Objective-C programs, `ADDITIONAL_OBJC_LIBS` is placed before all of the Objective-C Runtime and system libraries so that they will be searched first when linking. For command line tools and applications, `ADDITIONAL_OBJC_LIBS` is placed *before* all of the Objective-C Runtime and system libraries but *after* the Foundation and GUI libraries. Libraries specified with `ADDITIONAL_OBJC_LIBS` should only depend upon the Objective-C Runtime and/or system functions, not Foundation or GUI classes; Foundation dependent libraries should be specified with `ADDITIONAL_TOOL_LIBS` and GUI dependent libraries should be specified with `ADDITONAL_GUI_LIBS`. The additional libraries should be specified as '`-l`' flags to the linker as the following example illustrates.

```
ADDITIONAL_OBJC_LIBS = -lSwarm
```

ADDITIONAL_TOOL_LIBS [Variable]

`ADDITIONAL_TOOL_LIBS` is the list of additional libraries that the linker will use when linking command line tools and applications, see Section 1.4.15 [tool.make], page 11 and Section 1.4.2 [application.make], page 3. For command line tools, `ADDITIONAL_TOOL_LIBS` is placed before all of the GNUstep and system libraries so that they will be searched first when linking. For applications, `ADDITIONAL_TOOL_LIBS` is placed before the Foundation and system libraries but after the GUI libraries. Libraries specified with `ADDITIONAL_TOOL_LIBS` should only depend upon the Foundation classes and/or system functions, not GUI classes; GUI dependent libraries should be specified with `ADDITIONAL_GUI_LIBS`. The additional libraries should be specified as '`-l`' flags to the linker as the following example illustrates.

```
ADDITIONAL_TOOL_LIBS = -lone -lsimple
```

ADDITIONAL_GUI_LIBS [Variable]

`ADDITIONAL_GUI_LIBS` is the list of additional libraries that the linker will use when linking applications, see Section 1.4.2 [application.make], page 3.

ADDITIONAL_GUI_LIBS is placed before all of the GUI, Foundation, and system libraries so that they will be searched first when linking. The additional libraries should be specified as '-l' flags to the linker as the following example illustrates.

```
ADDITIONAL_GUI_LIBS = -lMiscGui
```

LIBRARIES_DEPEND_UPON [Variable]

LIBRARIES_DEPEND_UPON is the set of libraries that the shared library depends upon, see Section 1.4.8 [library.make], page 4 for more information about building shared libraries; this variable is only relevant for library project types. On some platforms when a shared library is built, any libraries which the object code in the shared library depends upon must be linked in the generation of the shared library. This is similar to the process of linking an executable file like a command line tool or Objective-C program except that the result is a shared library. Libraries specified with LIBRARIES_DEPEND_UPON should be listed as '-l' flags to the linker; when possible use variables defined by the Makefile Package to specify GUI, Foundation, or system libraries; like GUI_LIBS, FND_LIBS, OBJC_LIBS, or SYSTEM_LIBS. LIBRARIES_DEPEND_UPON is independent of ADDITIONAL_OBJC_LIBS, ADDITIONAL_TOOL_LIBS, and ADDITIONAL_GUI_LIBS, so any libraries specified there may need to be specified with LIBRARIES_DEPEND_UPON. The following example illustrates the use of LIBRARIES_DEPEND_UPON for a shared library that is depend upon the Foundation, ObjC, system libraries and an additional user library.

```
LIBRARIES_DEPEND_UPON = -lsimple $(FND_LIBS) $(OBJC_LIBS) $(SYSTEM_LIBS)
```

ADDITIONAL_INSTALL_DIRS [Variable]

ADDITIONAL_INSTALL_DIRS is the list of additional directories that should be created when the Makefile Package installs the file for the project. These directories are only one that the project needs to be created but that the Makefile Package does not automatically create. The directories should be absolute paths but use the GNUSTEP_INSTALLATION_DIR variable and other Makefile Package define variables, see Section 1.7.1 [Directory Paths], page 15, so that the directories get created in the appropriate place relative to the other file installed for the project. The following example illustrates how two additional directories can be created during installation.

```
ADDITIONAL_INSTALL_DIRS = $(GNUSTEP_INSTALLATION_DIR)/MyProject \
                          $(GNUSTEP_RESOURCES)/MyProject
```

1.6 Global Rules ('GNUmakefile.postamble')

The 'GNUmakefile.postamble' file is an optional file you may include in your package to define additional rules that should be executed when making and/or installing the project. There is a template 'GNUmakefile.postamble' file in the Makefile package that you can use as an example. Most of the rules are self explanatory. The 'before-' rules define things that should happen before a process is executed (e.g. 'before-all' for before compilation, 'before-install' for before installation). The 'after-' rules define things that should happen after a process is complete.

You can even define additional rules such as ones that a particular to your specific package or that are to be used by developers only.

1.7 Common Variables ('common.make')

Any of these variables that are defined by 'common.make' can and should be used by the user's makefile fragments to reference directories and/or perform any tasks which are not done automatically by the Makefile Package. Most variables refer to directory paths, both absolute and relative, where files will be installed, but other variables are defined based upon the target platform that the person is compiling for. Do not change the values of any of these automatically defined variables as the resultant behaviour of the Makefile Package is undefined.

1.7.1 Directory Paths

GNUSTEP_MAKEFILES [Variable]

`GNUSTEP_MAKEFILES` is the absolute path to the directory where the Makefile Package files are located. Use `GNUSTEP_MAKEFILES` to refer to a makefile fragment or script file from the Makefile Package within a makefile; the `GNUSTEP_MAKEFILES` variable should be only be used within makefiles and not referenced within C or Objective-C programs.

GNUSTEP_APPS [Variable]

`GNUSTEP_APPS` is the absolute path to the directory where GUI applications are installed. This variable is dependent upon the `GNUSTEP_INSTALLATION_DIR` variable, so the path will change accordingly if the user specifies a different installation root directory.

GNUSTEP_TOOLS [Variable]

`GNUSTEP_TOOLS` is the absolute path for the root directory where command line tools are installed. Only command line tools which are target platform independent should be installed in `GNUSTEP_TOOLS`; target platform dependent command line tools should be placed in the appropriate subdirectory of `GNUSTEP_TOOLS`, see Section 1.7.1 [GNUSTEP_TARGET_DIR], page 15, and Section 1.7.1 [TOOL_INSTALLATION_DIR], page 15. This variable is dependent upon the `GNUSTEP_INSTALLATION_DIR` variable, so the path will change accordingly if the user specifies a different installation root directory.

GNUSTEP_HEADERS [Variable]

GNUSTEP_HEADERS is the absolute path for the root directory where header files are installed. Normally header files are not installed in the GNUSTEP_HEADERS directory, but in a subdirectory as specified by the project which owns the files, see Section 1.4.8 [library.make], page 4 for more information. GNUSTEP_HEADERS should contain platform independent header files because the files are shared by all platforms. Any target platform dependent header files should be placed in the appropriate subdirectory as specified by GNUSTEP_TARGET_DIR. This variable is dependent upon the GNUSTEP_INSTALLATION_DIR variable, so the path will change accordingly if the user specifies a different installation root directory.

GNUSTEP_LIBRARIES_ROOT [Variable]

GNUSTEP_LIBRARIES_ROOT is the absolute path for the root directory where libraries are installed. Because libraries are binary objects and thus inherently target platform dependent, no libraries should actually reside in GNUSTEP_LIBRARIES_ROOT; libraries are placed in the appropriate subdirectory taking the target and possibly the library combo into account, see Section 1.7.1 [GNUSTEP_TARGET_LIBRARIES], page 15, and Section 1.7.1 [GNUSTEP_LIBRARIES], page 15. This variable is dependent upon the GNUSTEP_INSTALLATION_DIR variable, so the path will change accordingly if the user specified a different installation root directory.

GNUSTEP_TARGET_LIBRARIES [Variable]

GNUSTEP_TARGET_LIBRARIES is the absolute path for the directory where libraries are installed taking the target platform into account. It is a subdirectory of GNUSTEP_LIBRARIES_ROOT and is where libraries that do not depend upon the library combination, GNUstep or others, should be placed. This variable is dependent upon the GNUSTEP_INSTALLATION_DIR variable, so the path will change accordingly if the user specifies a different installation root directory.

GNUSTEP_LIBRARIES [Variable]

GNUSTEP_LIBRARIES is the absolute path for the directory where libraries are installed taking the target platform and library combination into account. It is a subdirectory of GNUSTEP_TARGET_LIBRARIES and therefore a subdirectory of GNUSTEP_LIBRARIES_ROOT. This directory is generally where library project types, see Section 1.4.8 [library.make], page 4, will install the library file. This variable is dependent upon the GNUSTEP_INSTALLATION_DIR variable, so the path will change accordingly if the user specifies a different installation root directory.

GNUSTEP_RESOURCES [Variable]

GNUSTEP_RESOURCES is the absolute path for the directory where resource files are installed; example resources are fonts, printer type information, model files for system panels, and system images. The resource files are generally associated with libraries, because resources for applications or bundles are included within the application or bundle directory wrapper. `GNUSTEP_RESOURCES` is a subdirectory of `GNUSTEP_LIBRARIES_ROOT`; it is dependent upon the `GNUSTEP_INSTALLATION_DIR` variable, so the path will change accordingly if the user specifies a different installation root directory.

GNUSTEP_HOST_DIR [Variable]

`GNUSTEP_HOST_DIR` is the subdirectory path for the host platform CPU and operating system. It is a composed from the `GNUSTEP_HOST_CPU` and `GNUSTEP_HOST_OS` variables.

GNUSTEP_TARGET_DIR [Variable]

`GNUSTEP_TARGET_DIR` is the subdirectory path for the target platform CPU and operating system. It is composed from the `GNUSTEP_TARGET_CPU` and `GNUSTEP_TARGET_OS` variables. `GNUSTEP_TARGET_DIR` is generally used as part of the installation path when platform specific files are installed.

GNUSTEP_OBJ_DIR [Variable]

`GNUSTEP_OBJ_DIR` is the subdirectory path where the Makefile Package places binary files: object files, libraries, executables, produced by the compiler. The Makefile Package separates binary files for different target platforms, different library combinations, and different compile options into different directories; these different directories are subdirectories from the current directory where the makefile resides. This structure allows a package to be compiled for different target platforms, different library combinations, and different compile options *in place*; i.e. the binary files are separated from each other so a compile pass from one set of options do not overwrite or erase binary files from a previous compile pass with different options. Generally the user does not use this variable; however, if the package needs to manually install some binary files than the makefile fragment uses this variable to reference the path where the binary file is located.

1.7.2 Scripts

CONFIG_GUESS_SCRIPT [Variable]

`CONFIG_GUESS_SCRIPT` is the absolute path to the '`config.guess`' script within the Makefile Package; this script is used to determine host and target platform information. The Makefile Package executes this script to determine the values of the host platform variables: `GNUSTEP_HOST`, `GNUSTEP_HOST_CPU`, `GNUSTEP_HOST_VENDOR`, `GNUSTEP_HOST_OS`, and the target platform variables: `GNUSTEP_TARGET`, `GNUSTEP_TARGET_CPU`, `GNUSTEP_TARGET_VENDOR`, `GNUSTEP_TARGET_OS`; generally the user does not need to execute this script because the Makefile Package executes it automatically.

CONFIG_SUB_SCRIPT [Variable]

CONFIG_SUB_SCRIPT is the absolute path to the '`config.sub`' script within the Makefile Package; this script takes a platform name and canonicalizes it, i.e. it puts the name in a standard form. The Makefile Package uses this script when the user specifies a target platform for compilation; the target platform name is canonicalized so that the Makefile Package can properly parse the name into its different components. Generally the user does not execute this script.

CONFIG_CPU_SCRIPT [Variable]

CONFIG_CPU_SCRIPT is the absolute path to the '`cpu.sh`' script within the Makefile Package; this script extracts the CPU name from a canonicalized platform name. Generally the user does not execute this script; it is used internally by the Makefile Package.

CONFIG_VENDOR_SCRIPT [Variable]

CONFIG_VENDOR_SCRIPT is the absolute path to the '`vendor.sh`' script within the Makefile Package; this script extracts the vendor name from a canonicalized platform name. Generally the user does not execute this script; it is used internally by the Makefile Package.

CONFIG_OS_SCRIPT [Variable]

CONFIG_OS_SCRIPT is the absolute path to the '`os.sh`' script within the Makefile Package; this script extracts the operating system name from a canonicalized platform name. Generally the user does not execute this script; it is used internally by the Makefile Package.

CLEAN_CPU_SCRIPT [Variable]

CLEAN_CPU_SCRIPT is the absolute path to the '`clean_cpu.sh`' script within the Makefile Package; this script takes a platform CPU name and *cleans* it for use by the Makefile Package. The process of cleaning refers to the situation where numerous equivalent processors, which have different names, are mapped to a single name. For example, the Intel line of processors: i386, i486, Pentium, all have different CPU names, but the Makefile Package considers them equivalent and cleans those names so that the single name '`ix86`' is used. Generally the user does not execute this script; it is used internally by the Makefile Package.

CLEAN_VENDOR_SCRIPT [Variable]

CLEAN_VENDOR_SCRIPT is the absolute path to the '`clean_vendor.sh`' script within the Makefile Package; this script takes a platform vendor name and *cleans* it for use by the Makefile Package. The process of cleaning refers to the situation where numerous equivalent vendors, which have different names, are mapped to a single name. Generally the user does not execute this script; it is used internally by the Makefile Package.

CLEAN_OS_SCRIPT [Variable]

CLEAN_OS_SCRIPT is the absolute path to the '`clean_os.sh`' script within the Makefile Package; this script takes a platform operating system name and *cleans* it for use by the Makefile Package. The process of cleaning refers to the situation where numerous equivalent operating systems, which have different names, are mapped to a single name. Generally the user does not execute this script; it is used internally by the Makefile Package.

1.7.3 Host and Target Platform Information

GNUSTEP_HOST [Variable]

`GNUSTEP_HOST` is the canonical host platform name; i.e. the name of the platform which is performing compilation of programs. For example, a SPARC machine by Sun Microsystems running the Solaris 2.5.1 operating system has the name `sparc-sun-solaris2.5.1`.

GNUSTEP_HOST_CPU [Variable]

`GNUSTEP_HOST_CPU` is the CPU name for the canonical host platform name; i.e. the name of the CPU platform which is performing compilation of programs. The Makefile Package cleans this CPU name with the `CLEAN_CPU_SCRIPT` script before using it internally. For example, the canonical host platform name of `i586-pc-linux-gnu` has a CPU name of `ix86`.

GNUSTEP_HOST_VENDOR [Variable]

`GNUSTEP_HOST_VENDOR` is the vendor name for the canonical host platform; i.e. the name of the vendor platform which is performing compilation of programs. The Makefile Package cleans this vendor name with the `CLEAN_VENDOR_SCRIPT` script before using it internally. For example, the canonical host platform name of `sparc-sun-solaris2.5.1` has a vendor name of `sun`.

GNUSTEP_HOST_OS [Variable]

`GNUSTEP_HOST_OS` is the operating system name for the canonical host platform; i.e. the name of the operating system platform which is performing compilation of programs. The Makefile Package cleans this operating system name with the `CLEAN_OS_SCRIPT` script before using it internally. For example, the canonical host platform name of `i586-pc-linux-gnu` has an operating system name of `linux-gnu`.

GNUSTEP_TARGET [Variable]

`GNUSTEP_TARGET` is the canonical target platform name; i.e. compilation of programs generate object code for this platform. By default the target platform is the same as the host platform unless the user specifies a different target when running make, see Cross Compiling.

GNUSTEP_TARGET_CPU [Variable]

`GNUSTEP_TARGET_CPU` is the CPU name for the canonical target platform; i.e. compilation of programs generate object code for this CPU platform.

The Makefile Package cleans this operating system name with the CLEAN_CPU_SCRIPT script before using it internally. By default the target CPU platform is the same as the host CPU platform, GNUSTEP_HOST_CPU, unless the user specifies a different target platform when running make, see Cross Compiling.

GNUSTEP_TARGET_VENDOR [Variable]

GNUSTEP_TARGET_VENDOR is the vendor name for the canonical target platform; i.e. compilation of programs generate object code for this vendor platform. The Makefile Package cleans this vendor name with the CLEAN_VENDOR_SCRIPT script before using it internally. By default the target vendor platform is the same as the host vendor platform, GNUSTEP_HOST_VENDOR, unless the user specifies a different target platform when running make, see Cross Compiling.

GNUSTEP_TARGET_OS [Variable]

GNUSTEP_TARGET_OS is the operating system name for the canonical target platform; i.e. compilation of programs generate object code for this operating system platform. The Makefile Package cleans this operating system name with the CLEAN_OS_SCRIPT script before using it internally. By default the target operating system platform is the same as the host operating system platform, GNUSTEP_HOST_OS, unless the user specifies a different target platform, see Cross Compiling.

1.7.4 Library Combination

OBJC_RUNTIME_LIB [Variable]

OBJC_RUNTIME_LIB is assigned the code that indicates the Objective-C Runtime library which compiled Objective-C programs will use; the three possible values are: 'gnu' for the GNU Runtime, 'nx' for the NeXT Runtime, and 'sun' for the Sun Microsystems Runtime. The Objective-C Runtime library can be changed to use a library other than the default with the 'library_combo' make parameter, see Section 1.3 [Running Make], page 1 for more details. Read Section 1.7.4 [Library Combination], page 20 for more information on how the Makefile Package handles different library combinations. If a makefile must perform specific operations dependent upon the Objective-C Runtime library then this variable is the one to check.

RUNTIME_DEFINE [Variable]

RUNTIME_DEFINE is assigned a preprocessor flag that can be passed to the compiler which defines a macro based upon the Objective-C Runtime library that compiled Objective-C programs will use. This macro is useful if the compiled program must execute different code based upon the Objective-C Runtime being used. See Section 1.5 [GNUmakefile.preamble], page 11 for an example on how to pass this preprocessor flag when compiling. The three possible values are: '-DGNU_RUNTIME=1' for the GNU Runtime, '-DNeXT_RUNTIME=1' for the NeXT Runtime, and '-DSun_RUNTIME=1' for the Sun Microsystems Runtime.

FOUNDATION_LIB [Variable]

FOUNDATION_LIB is assigned the code that indicates the Foundation Kit library, as specified by the OpenStep specification, which compiled Objective-C programs will use; the four possible values are: 'gnu' for the GNUstep Base Library, 'nx' for the NeXT Foundation Kit Library, 'sun' for the Sun Microsystems Foundation Kit Library, and 'fd' for the libFoundation Library. The Foundation Kit library can be changed to use a library other than the default with the 'library_combo' make parameter, see Section 1.3 [Running Make], page 1 for more details. Read Section 1.7.4 [Library Combination], page 20 for more information on how the Makefile Package handles different library combinations. If a makefile must perform specific operations dependent upon the Foundation Kit library then this variable is the one to check.

FND_DEFINE [Variable]

FND_DEFINE is assigned a preprocessor flag that can be passed to the compiler which defines a macro based upon the Foundation Kit library, as specified by the OpenStep specification, which compiled Objective-C programs will use. This macro is useful if the compiled program must execute different code based upon the Foundation Kit library being used. See Section 1.5 [GNUmakefile.preamble], page 11 for an example on how to pass this preprocessor flag when compiling. The four possible values are: '-DGNUSTEP_BASE_LIBRARY=1' for the GNUstep Base Library, '-DNeXT_Foundation_LIBRARY=1' for the NeXT Foundation Kit Library, '-DSun_Foundation_LIBRARY=1' for the Sun Microsystems Foundation Kit Library, and '-DLIB_FOUNDATION_LIBRARY=1' for the libFoundation Library.

GUI_LIB [Variable]

GUI_LIB is assigned the code that indicates the Application Kit library, as specified by the OpenStep specification, which compiled Objective-C programs will use; the two possible values are: 'gnu' for the GNUstep GUI Library and 'nx' for the NeXT Application Kit Library. The Application Kit library can be changed to use a library other than the default with the 'library_combo' make parameter, see Section 1.3 [Running Make], page 1 for more details. Read Section 1.7.4 [Library Combination], page 20 for more information on how the Makefile Package handles different library combinations. If a makefile must perform specific operations dependent upon the Application Kit library then this variable is the one to check.

GUI_DEFINE [Variable]

GUI_DEFINE is assigned a preprocessor flag that can be passed to the compiler which defines a macro based upon the Application Kit library, as specified by the OpenStep specification, which compiled Objective-C programs will use. This macro is useful if the compiled program must execute different code based upon the Application Kit library being used. See Section 1.5 [GNUmakefile.preamble], page 11 for an example on how to pass this preprocessor flag when compiling. The two possible values are: '-DGNUSTEP_GUI_LIBRARY=1' for the GNUstep GUI Library and '-DNeXT_Application_LIBRARY=1' for the NeXT Application Kit Library.

GUI_BACKEND_LIB [Variable]

GUI_BACKEND_LIB is assigned the code that indicates the backend library which compiled Objective-C programs will use in conjunction with the GNUstep GUI Library. The three possible values are: 'xdps' for the GNUstep X/DPS GUI Backend Library, 'nsx' for the NSKit GUI Backend Library, and 'w32' for the MediaBook WIN32 GUI Backend Library. GUI_BACKEND_LIB is only relevant when GUI_LIB is set to 'gnu'; otherwise, GUI_BACKEND_LIB will be set to 'nil' to indicate that there is no backend library. GUI_BACKEND_LIB can be changed to use a library other than the default with the 'library_combo' make parameter, see Section 1.3 [Running Make], page 1 for more details. Read Section 1.7.4 [Library Combination], page 20 for more information on how the Makefile Package handles different library combinations. If a makefile must perform specific operations dependent upon the backend library then this variable is the one to check.

GUI_BACKEND_DEFINE [Variable]

GUI_BACKEND_DEFINE is assigned a preprocessor flag that can be passed to the compiler which defines a macro based upon the backend library which compiled Objective-C programs will use in conjunction with the GNUstep GUI Library. This macro is useful if the compiled program must execute different code based upon the backend library being used. See Section 1.5 [GNUmakefile.preamble], page 11 for an example on how to pass this preprocessor flag when compiling. The three possible values are: '-DXDPS_BACKEND_LIBRARY=1' for the GNUstep X/DPS GUI Backend Library, '-DNSX_BACKEND_LIBRARY=1' for the NSKit GUI Backend Library, and '-DW32_BACKEND_LIBRARY=1' for the MediaBook WIN32 GUI Backend Library. GUI_BACKEND_DEFINE is not defined if there is not backend library; i.e. GUI_BACKEND_LIB is 'nil'.

1.7.5 Overridable Flags

OBJCFLAGS [Variable]

OBJCFLAGS are flags that are passed to the compiler when compiling Objective-C files. The user can override this variable when running make and specify different flags as the following command illustrates:

```
make OBJCFLAGS="-Wno-implicit -Wno-protocol"
```

CFLAGS [Variable]

CFLAGS are flags that are passed to the compiler when compiling C files. The user can override this variable when running make and specify different flags as the following command illustrates:

```
make CFLAGS="-Wall"
```

OPTFLAG [Variable]

OPTFLAG is the flag used to indicate the optimization level that the compiler should perform when compiling Objective-C and C files; this flag is set to '-O2' by default, but the user can override this setting when running make as the following command illustrates:

```
make OPTFLAG=
```

This command sets the optimization flag to be empty so that no optimization will be performed by the compiler.

GNUSTEP_INSTALLATION_DIR [Variable]

GNUSTEP_INSTALLATION_DIR is the root directory where the package will install its files; overriding this variable when running make will change all of the variables within the Makefile Package that depend upon it; the following command illustrates the use of this variable:

```
make GNUSTEP_INSTALLATION_DIR="/GNUstep"
```

This command states that the '/GNUstep' directory should be used as the installation root directory; applications in the package will be installed under '/GNUstep/Apps' directory, libraries in the package will be installed under the '/GNUstep/Library/Libraries' directory, command line tools will be installed under the '/GNUstep/Tools' directory, and etc. By default the Makefile Package sets GNUSTEP_INSTALLATION_DIR to GNUSTEP_LOCAL_ROOT unless GNUSTEP_LOCAL_ROOT is empty in which case GNUSTEP_INSTALLATION_DIR will be set to GNUSTEP_SYSTEM_ROOT. Some packages will also override GNUSTEP_INSTALLATION_DIR within their makefile to force the package to install in (say) GNUSTEP_SYSTEM_ROOT instead of the default GNUSTEP_LOCAL_ROOT directory, but if the user sets the value of GNUSTEP_INSTALLATION_DIR when running make then that setting takes precedence over all others.