CHAPTER 4

Using GNU make

All serious software projects are built in pieces by many developers. These pieces consist of source code and header files, libraries and different tools. To combine these pieces into a product is a process commonly known as a *build*. The GNU *make* utility is one of the many available utilities used to build these software projects. Large software projects often have hundreds and even thousands of files. Compiling these files and linking them into final executable product files is done by a careful and well-defined automated process. This automated process, also known as the *software build* process, is controlled by the `make` utility through *makefiles*. The `make` utility reads one or more of these makefiles containing information about how to build a project. Makefiles contain different types of information, including variables, control structures and rules to compile and link source files, build libraries and so on. In the most common makefiles, rules are present to remove object and executable files from the source code directory to clean it up, if required. These rules are enforced based upon time stamps on different files. For example, if an existing object file is newer than the corresponding source code file, it is not recompiled. However if the object file is older than the source file, it shows that someone modified the source file after the object file was last built. In such a case, `make` detects it and rebuilds the object file. Depending upon the requirements of a project, someone can create different types of rules and commands to build software projects.

This chapter contains information on how to write makefiles for software projects and provides sufficient information for a reader to write makefiles for fairly complex projects. You can also refer to the reference at the end of this chapter for more comprehensive information about the `make` utility.

To demonstrate the use of `make`, several examples are presented in this chapter. These examples demonstrate how `make` can be used for different objectives. These examples are simple and easy to understand and are explained in the text. However you may write makefiles in many different ways for the same object. Features of `make` that are discussed in this chapter are those most commonly used. `Make` has many additional, less commonly used, features that are not covered here and you may want to use these as well while designing your makefiles.

4.1 Introduction to GNU `make`

The `make` utility has been used for a very long time in all types of software development projects. There are open-source as well as commercial variants available from many vendors. The most common and popular distribution of `make` is the GNU `make`, which is open source and is available for almost all UNIX and Microsoft Windows platforms. All of the Linux distributions have the GNU `make` as a standard package with the development system. If you have installed development software from a distribution, you don't need to install any additional software for `make` to work. The version of `make` currently installed can be displayed using the following command.

```
[root@conformix make]# make -v
GNU Make version 3.79.1, by Richard Stallman and Roland
McGrath.
Built for i386-redhat-linux-gnu
Copyright (C) 1988, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98,
99, 2000
        Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS
FOR A PARTICULAR PURPOSE.

Report bugs to <bug-make@gnu.org>.

[root@conformix make]#
```

Since this is one of the most widely used software packages in the development community, there is a wealth of information available about it on the Internet. This chapter is intended to provide sufficient information to use `make` in all types of software projects.

Simply put, `make` reads a text input file that contains information about how to compile, link and build target files. `Make` uses commands and rules specified in this file to carry out its operation. We shall discuss in detail the contents of these files, commonly known as *makefiles*.

4.1.1 Basic Terminology

Before we move ahead with more information about `make` and how to use it, let us get familiar with some basic terminology about the software build process and the `make` utility. The terms defined below will be used throughout this chapter as well as in the chapters that follow.

Input Files

The `make` utility uses input files to build software projects. These input files, also known as *makefiles*, contain information about when and how to compile, assemble or link files. These input files also contain commands and rules to carry out these tasks.

Rules

A rule in an input file provides information about how to build an individual target (defined next). A rule has three parts:

1. The target
2. Dependencies
3. Commands

A target is rebuilt whenever a dependency file has a timestamp that is newer than the target. A target may have no dependency, in which case it is always rebuilt. The commands in the rule are executed to build the target from dependencies. The general format of a rule is as follows:

```
Target: Dependencies
  Commands
```

The commands part always starts with a TAB character. There are two types of rules that will be explained later in this chapter. These are types are:

1. The explicit rules
2. The implicit rules

The rules defined above fall into explicit rules category. Implicit rules are predefined rules that `make` uses when it does not find an implicit rule to build a target.

Target

A target is usually a file or set of files that is the result of some operation on one or more other files. A target may be an object file, a final executable file or some intermediate file. Targets may be fake or phony, in which case no files are actually rebuilt. One such example is the phony target to remove all object and executable files from the source files directory. This process is often called a *cleaning process*. As a convention, all makefiles have a rule to clean the source code so that everything can be rebuilt freshly.

Dependency

The dependencies are files that determine when to build a target. The decision to rebuild a target is made if the timestamp on any dependency is newer than the target. This usually shows that a dependency file has been modified after the target was built the last time. Consider the example of an object file that was built from a C source file. After building the object file, if someone modifies the source code file, we need to rebuild the object file because it is no longer current. When the `make` utility scans the input files, it verifies this rule and rebuilds the object file automatically if it finds a change in the source file. If multiple machines are used, they need to be in time sync.

A target may have multiple dependencies separated by space characters. In case of multiple dependencies, the target is rebuilt if any of the dependencies has changed. If the list of dependencies is very long, you can write multiple lines for dependencies by using a backslash character at the end of a line to continue to the next line. For example, the following lines show that `tftp.o` has dependencies spanning three lines.

```
tftp.o: tftp.c tftp.h file1.c file2.c file3.c file4.c\
        file5.c file6.c file7.c file1.h file2.h file3.h\
        file4.h file5.h
```

Note that the starting character in the second and third line should not be a TAB character; otherwise `make` will consider these lines as command lines and will try to execute these.

Default Goal

An input file (makefile) usually contains many rules. Each of these rules has a target. However the `make` utility does not build all of these targets. It only tries to build the first target specified in the input makefile. This target is called the *default* goal of the makefile. However, note that if dependencies to this default goal are targets of some other rules, the `make` utility can also build those targets in an effort to build the default goal. This happens when dependencies of the default goal are out of date.

If you want to build a target that is not the default goal, you have to specify the target at the command line when starting `make`. This will be explained later in this chapter.

Phony Target

Some targets may have no dependencies. A common example is the `clean` target that removes some files from the source directory tree. A typical `clean` target is shown below:

```
clean:
  rm *.o
```

Whenever you invoke `make` to build this target, the `rm` command is always executed. This is because the target has no dependency. These types of targets are called phony targets.

However there is a problem with such targets. Sometimes if there is a file with the same name as the phony target, the `make` utility may get confused while building it and execute the command in the phony target when it is not required to do so. To avoid this problem, you can specify which targets are phony targets in the makefile. The following line in the makefile defines clean as a phony target.

```
.PHONY: clean
clean:
  rm *.o
```

4.1.2 Input Files

The `make` command, when invoked without a command line argument, expects certain file names as input. These files names are searched in the current directory in the following order.

1. `GNUmakefile`
2. `makefile`
3. `Makefile`

If none of these files is present, `make` will not take any action and displays the following error message:

```
[root@conformix make]# make
make: *** No targets specified and no makefile found.  Stop.
[root@conformix make]#
```

However there is a way to specify files with non-standard names as explained later. The common practice is to use "Makefile" as the name of the input file because it easy to distinguish from other files due to upper case M.

NOTE If you have multiple input files in the same directory with names `GNUmakefile`, `makefile` and `Makefile`, only the `GNUmakefile` will be used. As a rule of thumb, `make` will use only the first available file in its priority list to build a project. This means that if we have two files with names `makefile` and `Makefile`, only the file named `makefile` will be used.

You can specify any other filename using command line switch `–f` with the `make` command. For example, if you have a file named `myrules.make`, you can use it as input file to the `make` utility as follows:

```
make –f myrules.make
```

From now on in this book, we shall use only "Makefile" as the name of the input file for the `make` command. This is a usual convention in most of the software development projects. Input files with any other name will be used as include files in the Makefile. The common conventional name for include file is `Makefile.in`.

4.1.3 Typical Contents of a Makefile

The makefiles that are used as input to the `make` command have similar types of contents. Some of the typical contents of a makefile are as follows.

Variables

Variables are usually the first part of any makefile. Variables are used to associate a name to a text string in the makefiles. These variables are then substituted into other targets, dependencies or commands later in the makefile. A simple equal sign can associate a text string to a name. The following line defines a variable *targets* in the makefile. This variable is equal to two target files `tftp.o` and `fto.o`.

```
targets = tftp.o ftp.o
```

To use this variable later on, you enclose the variable name in parentheses and use a dollar sign with it. The targets variable will be used as `$(targets)` in the makefile to refer to these two object files. A typical use of this variable is shown in the following three lines.

```
$(targets): tftp.c tftp.h ftp.c ftp.h
  gcc –c tftp.c
  gcc –c ftp.c
```

This rule tells `make` command that if any of the four dependency files (`tftp.c`, `tftp.h`, `ftp.c`, `ftp.h`) has a timestamp newer than the object files, the object files should be rebuilt.

Now you may have noticed that this is not a wise rule. If only `tftp.c` is changed, then we need to build only `tftp.o` but with this rule we are also building `ftp.o` which may not be required. However this type of rule may be useful when a single dependency may affect many targets. For example, both `ftp.o` and `tftp.o` may be dependent on `common.h` file and the following rule will make sense.

```
$(targets): common.h
  gcc –c tftp.c
  gcc –c ftp.c
```

We shall discuss more about these things and how to write wiser and more comprehensive rules in makefiles later in this chapter.

Rule to Build the Default Goal

After defining the variables, the first rule defines the default goal. This is usually used to build one or more final executable from all of the source, object and library files.

Rules to Build Other Goals

After defining the default goal, you can define as many other rules as you like. These other rules are used to build individual parts of a big project. For example, if a project consists of three main parts, the FTP server, the TFTP server and the DNS resolver, you can have three individual rules for all of these individual parts. The default goal may be used to build all of these three targets in a single step.

Rules to Build Individual Objects

In addition to defining rules to build goals, you may have to define rules to build individual objects or intermediate files. One example of these rules is to build object files from the source files. These object files may then be used to build executable or library files.

Rules to Install

Some rules may be required to install the files into appropriate directories created by other rules. For example, after building library files, you may want to install them into `/lib` or `/usr/lib` directory. Similarly, after building executable files, you may want to install them into `/bin`, `/usr/bin` or `/sbin` directory or any other directory of your choice. For this reason, one or more rules may be defined in the makefiles so that `make` copies the appropriate files into their target location.

Rules to Clean Previous Builds

As mentioned earlier, the rule to clean previous builds is usually a phony rule to clean up the source code tree from the previously built intermediate and target files. This rule is useful when you want to hand over a clean directory to someone or to force everything to be rebuilt, perhaps with new options.

Comments and Blank Lines

As a good convention, you can insert some comment lines in makefiles so that you and others can make sense of what is there. All comments start with a hash (#) symbol. You can also put a comment after a command. The rule of thumb is that anything after the hash character will be considered a comment. If this character is the first character in a line, all of the line is a comment line.

CAUTION Any empty line that begins with a TAB is not really an empty line. It is considered an empty command. Empty commands are sometimes useful when you don't want to take any action for a particular dependency.

4.1.4 Running make

In its basic and most widely used form, simply entering "make" on the command line invokes the make program. Upon invocation, it reads its input file, in a priority order as explained earlier, and executes different commands in it to build the default goal. In this chapter, we are using a hypothetical project that builds three applications: ftp, tftp and dnsresolver. Reference to this project will be used in many parts of the remaining chapter. See section 4.4 for a complete listing of the makefiles used.

Building a Default Goal

If we have a Makefile in the current directory and the default goal is to build all of the ftp, tftp, and dnsresolver, specified by "all" rule, a typical session will be as follows:

```
[root@rr2 test]# make
gcc -g -O2 -c ftp.c
gcc -g -O2 -c common.c
gcc -static ftp.o common.o -o ftp
gcc -g -O2 -c tftp.c
gcc -static tftp.o common.o -o tftp
gcc -g -O2 -c dnsresolver.c
gcc -static dnsresolver.o common.o -o dnsresolver
[root@rr2 test]#
```

Building a Non-Default Goal

If you have multiple goals in a Makefile and you don't want to build a default goal, you have to specify the goal you want to build on the command line. The following line builds only the ftp server.

```
[root@rr2 test]# make ftp
gcc -g -O2 -c ftp.c
gcc -g -O2 -c common.c
gcc -static ftp.o common.o -o ftp
[root@rr2 test]#
```

When a Target is Updated

There are certain rules about when a target will be built or rebuilt. These are as listed below.

- *Target is not present*. If a target is not present and a rule is executed, the target will always be rebuilt.
- *Target is outdated*. This means that one or more dependencies of a target are newer than the target. This happens when someone changed a dependency after the target was built last time.
- *Target is forced to be rebuilt*. You can define rules to force a target to be rebuilt whether a dependency has changed or not. This is also true for phony targets.

4.1.5 Shell to Execute Commands

All commands in a makefile are executed in a subshell invoked by the `make` command. It is important to know which shell will be used to execute commands in the makefile. By default the shell name is `/bin/sh`. However this can be changed using the `SHELL` variable in the makefile. Please note that this is not the environment variable `SHELL`, but is set within the makefile. This is to avoid conflict with someone's personal preference of using a particular shell.

Each line that starts with TAB character represents a command and is executed by a single instance of shell. So, for example, the following line will cause a directory change to `tftp` and the command `rm *.o` to be executed from that directory. After executing this command, the `next` command is executed in the original directory because a new subshell is invoked and the effect of the `cd tftp` command is gone.

```
tftp: tftp/Makefile
  cd tftp; rm *.o
  @echo "Rebuilding . . ."
```

As mentioned earlier, one line can be split into multiple lines by placing a backslash at the end of the line. The above three lines can be written as follows without any change in their result:

```
tftp: tftp/Makefile
  (cd tftp;\
rm *.o)
  @echo "Makefile changed. Rebuilding …"
```

4.1.6 Include Files

In case of large projects, makefile rules can be split into many files. These smaller files can then be included into a makefile. When the `make` utility is scanning through the makefile and it finds an `include` file, it will go through the `include` file before going to the next line in the makefile. Logically, you can think of contents of the `include` file being inserted into the makefile at the place where the file is included.

Files can be included using the *include* directive inside a makefile. The following line includes a file `myrules.in` in a makefile.

```
include myrules.in
```

Multiple files can be included into a makefile on the same line. The following line includes two files `myrules.in` and `bootarules.in` into a makefile.

```
include myrules.in bootarules.in
```

When multiple files are included, these are read in the order they are present in the makefile. In the above example, first `myrules.in` will be read by the `make` utility and then `boot-arules.in` will be read.

You can also use wildcards and variables to specify file names with the include directives. The following include directive includes all files with last part as ".in" and files specified by the variable `MKINCS`.

```
include *.in $(MKINCS)
```

4.2 The `make` Rules

Rules are the most important part of a makefile. Rules control how a project should be built. Each rule has a common structure which is discussed in this section. A typical makefile is introduced as an example to show you how rules are written in makefiles.

4.2.1 Anatomy of a Rule

As mentioned earlier, a typical rule looks like the following:

```
Target: Dependencies
  Commands
```

A target is the objective of the rule that depends upon a list of dependencies. Commands are used to build the target. The target name is followed by a colon. Commands start with a TAB character.

CAUTION Each line containing commands starts with a TAB character. The TAB character distinguishes command lines from other lines. A common error is to replace the TAB character with a multiple space character. The result is that commands on this line are not executed.

Multiple Commands in the Same Line

A rule can have multiple commands to build target. These commands can be listed in multiple lines or in a single line. If commands are listed in a single line, they are separated by a semicolon. It should be noted that the difference is whether or not they are executed in the same subshell. Commands listed in one line are executed in the same subshell. If each command is listed in a separate line, each line starts with a TAB character in the start of the line. The following rule, used to remove some files from the current directory, has two commands. The first is `echo` and the second is the `rm` command.

```
clean:
  @echo "Deleting files ..."
  rm -f $(OBJS) *~
```

These two commands can also be listed as follows:

```
clean:
  @echo "Deleting files ..." ; rm -f $(OBJS) *~
```

Command Echoing

By default all commands in the makefile are echoed to standard output as these are executed. This provides information about the activity of `make` as it is going through the makefile. However you can suppress the command echo by starting commands with the character `@`.

This is especially useful in printing information from within the makefile using the `echo` command. For example, you can create a makefile that prints information before it starts executing. A typical rule in the makefile that displays information about the object before executing any other command follows. It displays a line showing that `make` is going to build FTP:

```
ftp: $(SRCS) $(HDRS)
        @echo "Building FTP"
        @$(CC) $(CFLAGS) $(INCLUDES) ftp.c
        @$(CC) $(LDFLAGS) $(COMMON) $(OBJS) -lcommon -o ftp
```

When `make` reads this file, it displays the following output:

```
[root@conformix ftp-dir]# make ftp
Building FTP
[root@conformix ftp-dir]#
```

If you don't insert the `@` characters in this file, the result will be as follows. You can easily find the difference between this and the earlier output of the `make` command.

```
[root@conformix ftp-dir]# make ftp
echo "Building FTP"
Building FTP
gcc -g -O2 -c -I../common-dir ftp.c
gcc -static -L../common-dir  ftp.o  -lcommon -o ftp
[root@conformix ftp-dir]#
```

NOTE There are other ways to suppress the command `echo`. One of these is the use of `–s` or `–silent` flag on the command line. However the most common method is the use of the `@` character.

Rules with no Dependencies

Many rules are used with no dependencies. These rules are used to execute commands without looking into any dependencies. Commands in these rules are guaranteed to be executed.

4.2.2 A Basic Makefile

Let us start with a simple makefile shown below. This makefile is used to build `ftp`, which is the default target.

```
####################################################
# Makefile created to demonstrate use of the make
# utility in the "Linux Development Platform" book.
#
# Author: Rafeeq Ur Rehman
#         rr@conformix.net
####################################################

# Variable definition
OBJS = ftp.o common.o
HDRS = ftp.h common.h
CFLAGS = -g -O2
TARGETS = ftp
CC = gcc

# Default Target

ftp: $(OBJS) $(HDRS)
  $(CC) $(OBJS) -o ftp

ftp.o: ftp.c $(HDRS)
  $(CC) $(CFLAGS) -c ftp.c

common.o: common.c common.h
  $(CC) $(CFLAGS) -c common.c

clean:
  rm -f $(TARGETS) $(OBJS)
```

This makefile has initial comments that show information about the file. After that, variables are defined. The rule to build default target is listed next. This rule has two dependencies which are defined by variables `$(OBJS)` and `$(HDRS)`. Rules to build the object files are listed after the default rule. The last rule, `clean`, is used to remove files created during the `make` process so that the directory contains only the initial source code files when you invoked this rule for the first time. This rule has no dependency, so the `rm` command is guaranteed to be executed.

When building the default target `ftp`, `make` checks the dependencies. For this it expands two variables `$(OBJS)` and `$(HDRS)`. After expansion, `make` finds out that the target is dependent upon the following four files.

1. `ftp.o`
2. `ftp.h`
3. `common.o`
4. `common.h`

Now make also finds out that there are other rules to build ftp.o and common.o files. If these two files are not up-to-date, make rebuilds these object files using corresponding rules. For example, if common.c file has been changed since make built common.o last time, make will rebuild it. After checking dependencies, make will build ftp if any of the dependency is newer than the ftp file or ftp file does not exist.

If you want to remove the target file and the object files, you can use the make clean command that invokes the clean rule. This rule has no dependency so the rm command will always execute, removing the ftp and any object files.

4.2.3 Another Example of Makefile

The following Makefile is a little bit bigger and is used to build three targets. These are as follows:

1. The ftp server
2. The tftp server
3. The dnsresolver

Before going into detail of how it is done, let us look at the Makefile itself. It is listed below.

```
###################################################
# Makefile created to demonstrate use of the make
# utility in the "Linux Development Platform" book.
#
# Author: Rafeeq Ur Rehman
#         rr@conformix.com
###################################################

# Variable definition
SRCS = ftp.c tftp.c dnsresolver.c common.c
OBJS = ftp.o tftp.o dnsresolver.o common.o
FTPOBJS = ftp.o common.o
FTPHDRS = ftp.h common.h
TFTPOBJS = tftp.o common.o
TFTPHDRS = tftp.h common.h
DNSRESOLVEROBJS =  dnsresolver.o common.o
DNSRESOLVERHDRS =  dnsresolver.h common.h
CC = gcc
CFLAGS = -g -O2
LDFLAGS = -static
TARGETS = ftp tftp dnsresolver
INSTALLDIR = /root

# Default Target
```

```
all: $(TARGETS)

# Rule to build object files

$(OBJS): $(SRCS)
  $(CC) $(CFLAGS) -c $(@:.o=.c)

# Rules to build individual targets

ftp: $(FTPOBJS) $(FTPHDRS)
  $(CC) $(LDFLAGS) $(FTPOBJS) -o ftp

tftp: $(TFTPOBJS) $(TFTPHDRS)
  $(CC) $(LDFLAGS) $(TFTPOBJS) -o tftp

dnsresolver: $(DNSRESOLVEROBJS) $(DNSRESOLVERHDRS)
  $(CC) $(LDFLAGS) $(DNSRESOLVEROBJS) -o dnsresolver

clean:
  rm -f $(TARGETS) $(OBJS)

install:
  cp $(TARGETS) $(INSTALLDIR)
# Additional Dependencies

ftp.o: $(FTPHDRS) tftp.o: $(TFTPHDRS)
dnsresolver.o: $(DNSRESOLVERHDRS)
```

After comments in the start of the makefile, the first part of the file defines variables used in this file. As you shall see, defining variables is very helpful if you need any modification in the Makefile or want to add or remove more files later on. After defining variables, the default target is defined with the name "all" as follows:

```
all: $(TARGETS)
```

This in turn has dependencies on the three targets that we want to `make`. Please note that we don't have any command to build this default target. When `make` reaches the default target, it tries to meet dependencies to make sure that these are up-to-date. If any of these dependencies is not up-to-date, `make` will try to recreate it.

The three dependencies to the default target (`ftp`, `tftp` and `dnsresolver`) have their own rules to build. For example, the following ftp rule shows that it depends upon two variables `$(FTPOBJS)` and `$(FTPHDRS)`.

```
ftp: $(FTPOBJS) $(FTPHDRS)
  $(CC) $(LDFLAGS) $(FTPOBJS) -o ftp
```

This means that if any of the files listed in these two variables has changed, the `ftp` target will be rebuilt. These files defined in the two variables are as follows:

CH04.fm Page 115 Monday, October 7, 2002 8:54 PM

- `ftp.o`
- `common.o`
- `ftp.h`
- `common.h`

We have another rule to build object files from source files. This rule is as follows:

```
$(OBJS): $(SRCS)
  $(CC) $(CFLAGS) -c $(@:.o=.c)
```

It will build any of the `.o` object file, which is not up-to-date from its corresponding `.c` source code file. The `@` sign is used to substitute the name of the target by replacing `.o` with `.c` so that if we want to build the target `common.o`, the `@` symbol will replace `.o` with `.c` in the target, making it `common.c`. This way the compiler gets the correct argument after the `-c` switch in the command line. This technique is especially useful if you are dealing with files with different extensions. For example, if the source code files are C++ files ending with `.cpp`, the above rule may be written as follows:

```
$(OBJS): $(SRCS)
  $(CC) $(CFLAGS) -c $(@:.o=.cpp)
```

Now the `@` will replace `.o` with `.cpp` for each file in the target.

The `clean` rule will delete all of the object and executable files. The install rule will copy files into the /root directory. Both of these rules have no dependency. This means that the commands in these rules don't depend on anything and will always be executed whenever this rule is invoked.

The last part of the makefile lists some additional dependencies not covered by earlier rules. Note that these three rules in the end have no command to execute. You can have as many rules to build a target as you like, but only one of these rules should contain commands to build it.

Now you can see how the use of variables is useful in this Makefile. If you add or remove some files or dependencies, you need to modify only the upper part of the makefile where variables are defined, without worrying about the rest of the `makefile`. For example, if you add a new header file to the dependency list of ftp, you just need to modify the variable `FTPHDRS`. Similarly to add a new source code file `common.c` that is being used by ftp, you need to add it to the following two variables:

1. `FTPOBJS`
2. `OBJS`

By modifying these two variables, this will automatically be used by `ftp`, `$(OBJS)` and `clean` rules.

Now is the time to use this makefile and see how the `make` program executes different commands. The output of the `make` command is shown below that needs some attention to understand how the `make` program checks dependencies and builds different targets.

```
[root@rr2 test]# make
gcc -g -O2 -c ftp.c
gcc -g -O2 -c common.c
gcc -static ftp.o common.o -o ftp
gcc -g -O2 -c tftp.c
gcc -static tftp.o common.o -o tftp
gcc -g -O2 -c dnsresolver.c
gcc -static dnsresolver.o common.o -o dnsresolver
[root@rr2 test]#
```

The default target is "all" which has dependency upon the variable $(TARGETS). This variable has three words: ftp, tftp and dnsresolver. First make tries to verify that ftp is up-to-date. For this it finds that ftp file does not exist so it tries to build it using some other rule. It finds the following rule to build ftp:

```
ftp: $(FTPOBJS) $(FTPHDRS)
  $(CC) $(LDFLAGS) $(FTPOBJS) -o ftp
```

Now it tries to find if dependencies of this rule are up-to-date. The object files are not present, so first it tries to build the object file. To build the object files, it looks for another rule and finds the following rule to build two object files (ftp.o and common.o) in the dependency list.

```
$(OBJS): $(SRCS)
  $(CC) $(CFLAGS) -c $(@:.o=.c)
```

When make executes the command in this rule, you see the following two lines in the output of the make command.

```
gcc -g -O2 -c ftp.c
gcc -g -O2 -c common.c
```

Now it has all of the dependencies for the ftp target up-to-date and it builds ftp using the corresponding rule. At this point you see the following line in the output of the make command:

```
gcc -static ftp.o common.o -o ftp
```

By building ftp, make has satisfied one dependency in the default goal. Now it will try to meet the second dependency, which is tftp. Since the tftp file is not present, it will locate a rule that can be used to build tftp. In the tftp rule dependencies, the common.o file is already up-to-date, so it will not recompile it. However since it does not find tftp.o, it will rebuild tftp.o. At this point you see the following line in the output:

```
gcc -g -O2 -c tftp.c
```

Now it has successfully built dependencies for tftp and it will build the tftp target and display the following line in the output:

```
gcc -static tftp.o common.o -o tftp
```

The same process is repeated for the `dnsresolver` and the following two lines are displayed:

```
gcc -g -O2 -c dnsresolver.c
gcc -static dnsresolver.o common.o -o dnsresolver
```

After building `dnsresolver`, nothing is left in the default target rule "all," so `make` will stop at this point.

Now let us see what happens if you modify the `ftp.h` file and run `make` again. The output will be as follows:

```
[root@rr2 test]# make
gcc -g -O2 -c ftp.c
gcc -static ftp.o common.o -o ftp
[root@rr2 test]#
```

This time `make` only rebuilt `ftp` because `ftp.h` is a dependency only for the target `ftp`. However if you modify `common.h`, it will rebuild `ftp`, `tftp` and `dnsresolver` as follows:

```
[root@rr2 test]# make
gcc -g -O2 -c ftp.c
gcc -static ftp.o common.o -o ftp
gcc -g -O2 -c tftp.c
gcc -static tftp.o common.o -o tftp
gcc -g -O2 -c dnsresolver.c
gcc -static dnsresolver.o common.o -o dnsresolver
[root@rr2 test]#
```

Modifying `common.c` will cause rebuilding of all object files as well. This result of `make` after modification in `common.c` is as follows:

```
[root@rr2 test]# make
gcc -g -O2 -c ftp.c
gcc -g -O2 -c common.c
gcc -static ftp.o common.o -o ftp
gcc -g -O2 -c tftp.c
gcc -static tftp.o common.o -o tftp
gcc -g -O2 -c dnsresolver.c
gcc -static dnsresolver.o common.o -o dnsresolver
[root@rr2 test]#
```

You can also use rules to build individual targets. For example, if you want to build only `ftp`, you use the following command line instead:

```
[root@rr2 test]# make ftp
gcc -g -O2 -c ftp.c
gcc -g -O2 -c common.c
gcc -static ftp.o common.o -o ftp
[root@rr2 test]#
```

Now let us clean the files we have created using the following rule.

```
clean:
  rm -f $(TARGETS) $(OBJS)
```

The result of running the command `make clean` will be as follows:

```
[root@rr2 test]# make clean
rm -f ftp tftp dnsresolver ftp.o tftp.o dnsresolver.o common.o
[root@rr2 test]#
```

As you can see from the above output, `make` has replaced variables with values present in the variables before running the `rm` command. This is what is done every time `make` invokes a command.

4.2.4 Explicit Rules

There are two types of rules: *explicit* rules and *implicit* rules. All of the rules that we have been using in this chapter until now are explicit rules. An explicit rule has three parts: target, dependencies and commands. Explicit rules are defined in detail and they perform exactly as they are written. In general, all rules present in a makefile are explicit rules.

4.2.5 Implicit Rules

Implicit rules are used by `make` to build certain targets by itself. These rules are usually language-specific and operate depending upon file extension. For example, `make` can build `.o` files from `.c` files using an implicit rule for C language compilation. Consider the basic makefile we used earlier in this chapter as shown below:

```
# Variable definition
OBJS = ftp.o common.o
HDRS = ftp.h common.h
CFLAGS = -g -O2
TARGETS = ftp
CC = gcc

# Default Target

ftp: $(OBJS) $(HDRS)
  $(CC) $(OBJS) -o ftp

ftp.o: ftp.c $(HDRS)
  $(CC) $(CFLAGS) -c ftp.c

common.o: common.c common.h
  $(CC) $(CFLAGS) -c common.c

clean:
  rm -f $(TARGETS) $(OBJS)
```

You can modify this file so that it uses implicit rules to build object files. The modified file is shown below:

```
# Variable definition
OBJS = ftp.o common.o
HDRS = ftp.h common.h
CFLAGS = -g -O2
TARGETS = ftp
CC = gcc

# Default Target

ftp: $(OBJS) $(HDRS)
  $(CC) $(OBJS) -o ftp

clean:
  rm -f $(TARGETS) $(OBJS)
```

Note that we have completely taken out two rules that are used to build the object files. Now when make needs to build these object files, it uses its implicit rule for this purpose. Running make on this makefile produces the following result.

```
[root@conformix make]# make
gcc -g -O2   -c -o ftp.o ftp.c
gcc -g -O2   -c -o common.o common.c
gcc ftp.o common.o -o ftp
[root@conformix make]#
```

Note that the first two lines of the output create object files using implicit rules. You may also have noted that the CFLAGS variable is also used in this process. Like CFLAGS, implicit rules use other variables while building targets. For a more detailed discussion, please see the reference at the end of this chapter.

While using implicit rules, you should be careful about the process, because make can build a target using explicit rule depending upon which source files are available. For example, make can produce an object file from a C source code file as well as Pascal source code file. In the above example, if common.c file is not present but common.p (Pascal source code file) is present in the current directory, make will invoke Pascal compiler to create common.o file, which may be wrong. You also have less control over options on implicit rules.

4.3 Using Variables

Variables are an important part of all makefiles used in a project. Variables are used for many purposes; the most important of which is to have structured and easily understandable makefiles. This section contains more information about variables used in makefiles.

4.3.1 Defining Variables

Variables can be defined in a usual way by typing in the variable name, followed by an equal sign. The equal sign is followed by the value that is assigned to the variable. Note that there may be space characters on one or both sides of the equal sign. A typical variable assignment may be as follows:

```
CFLAGS = -g -O2
```

Another example of defining the C compiler name is as follows:

```
CC = gcc
```

This variable than can be used in other parts of the makefile by placing a `$` sign and a pair of parenthesis or braces. The following line in a makefile is used to compile a file `tftp.c` and create an object file `tftp.o` with the help of above two variables;

```
$(CC) $(CFLAGS) -o tftp.o  tftp.c
```

This line can also be written as follows:

```
${CC} ${CFLAGS} -o tftp.o  tftp.c
```

NOTE Variables are case sensitive in makefiles. A variable `$(OUTFILES)` is different from a variable `$(OutFiles)`.

4.3.2 Types of Variables

There are two types of variables. Variables defined using the = sign are *recursively expanded variables*. This means that a variable may be expanded depending upon value of a variable at a later stage in the makefile. Consider the following lines in a makefile.

```
OBJ1 = ftp.o
OBJ2 = common.o
OBJS = $(OBJ1) $(OBJ2)
printobjs:
  @echo $(OBJS)
OBJ1 = ftp.o tftp.o
```

Variable `OBJS` will contain a list of three files, `ftp.o`, `tftp.o` and `common.o`, although `tftp.o` was added *after* the `echo` command. Output of this makefile is as follows.

```
[root@conformix make]# make
ftp.o tftp.o common.o
[root@conformix make]#
```

This is because `make` used the default target rule and executed the `echo` command. Before printing out the value of the `OBJS` variable, it scanned the makefile to the end to re-evaluate the value of the `OBJ1` variable and hence the `OBJS` variable.

The other types of variables are *simply expanded* variables and the value of these variables is determined at the time of their definition. These variables are defined using the `:= symbol` instead of just the = symbol. If we change the OBJS line in the above makefile to this type of variable, the value printed by the makefile will be as follows:

```
[root@conformix make]# make
ftp.o common.o
[root@conformix make]#
```

Now make did not take into consideration the changed value of the OBJ1 variable later in the makefile.

There are advantages and disadvantages to both types of variables. You can decide which type of variable to use in a particular situation.

4.3.3 Pre-Defined Variables

The make utility can also take variables from the shell environment. For example, the CFLAGS variable can be set through shell startup file (e.g. /etc/profile). If this variable is not redefined inside the makefile, its value will be taken from the environment.

4.3.4 Automatic Variables

Some variables are pre-defined and are called *automatic variables*. They are usually very short in length but play a very important role in the decision-making process. The most commonly used automatic variables are listed below.

- The $@ variable contains the value of the target of a rule.
- The $< variable always contains the first dependency of a rule.
- The $? variable contains a list of modified files in the dependency list. If the target is being built for the first time, it contains a list of all dependencies. Consider the following makefile.

```
# Variable definition
OBJS = ftp.o common.o
HDRS = ftp.h common.h

CFLAGS = -g -O2
TARGETS = ftp
CC = gcc

# Default Target

ftp: $(OBJS) $(HDRS)
  @echo $?
  @echo $@
  @echo $<
  $(CC) $(OBJS) -o ftp
```

Let's try to build the default target for the first time and see the output of these echo commands.

```
[root@conformix make]# make
gcc -g -O2   -c -o ftp.o ftp.c
gcc -g -O2   -c -o common.o common.c
ftp.o common.o ftp.h common.h
ftp
ftp.o
gcc ftp.o common.o -o ftp
[root@conformix make]#
```

The first two lines in the output use implicit rules to build object files. The third line in the output is generated by the first echo command and it has a list of all the dependencies. The second echo command displays the fourth line in the output, which is just the name of the target, i.e. ftp. The last echo command prints the first dependency in the fifth line of the output.

Now let us modify the common.c file and run make once again. The output will be as follows:

```
[root@conformix make]# make
gcc -g -O2   -c -o common.o common.c
common.o
ftp
ftp.o
gcc ftp.o common.o -o ftp
[root@conformix make]#
```

This time make used an implicit rule to build common.o. The important thing to note is the second line in the output, which is displayed by the first echo command in the makefile. This time it displayed only one file in the dependency list because common.o is the only file that is changed in the dependencies.

Automatic variables are very useful in control structures and in the decision-making process when you carry out an operation based upon the result of some comparison.

4.4 Working with Multiple Makefiles and Directories

Until now we have used simple makefiles to build projects or targets. We used only one makefile in the project and all of the source files were present in the same directory. In real-life software development projects, we have multiple directories containing different parts of a software product. For example, one directory may contain all library files, another header files and a third one common files used by different parts. Similarly every part of the software project may have its own subdirectory.

This section deals with the same source files and targets that we used earlier in this chapter. The project contains three targets to be built. These targets are ftp, tftp and dnsresolver. We are using the same set of source files. However to demonstrate the use of multiple directories and multiple makefiles, we have created each component in a separate directory. This may not be the best example for a real-life situation, but it is well suited to demonstrate one of many possible ways to handle such a situation. The following directory tree shows the arrangement of these files in different directories.

```
 [root@conformix make]# tree
.
|-- Makefile
|-- common-dir
|   |-- Makefile
|   |-- common.c
|   `-- common.h
|-- dns-dir
|   |-- Makefile
|   |-- dnsresolver.c
|   `-- dnsresolver.h
|-- ftp-dir
|   |-- Makefile
|   |-- ftp.c
|   `-- ftp.h
`-- tftp-dir
    |-- Makefile
    |-- tftp.c
    `-- tftp.h

4 directories, 13 files
[root@conformix make]#
```

As you can see, you have four directories, each having its own makefile as well as source files. There is a makefile in the top directory that is used with the `make` command. This top-level makefile used makefiles in sub-directories. We can carry out the common tasks of building, installing and cleaning different targets while sitting in the top directory. Let's have a look at makefile in each directory.

4.4.1 Makefile in The Top Directory

Makefile in the top directory is used to build all of the targets using different rules. To build a particular target, we move to the directory where source files for that target are present and run `make` in that directory. This makefile is shown below.

```
###################################################
# Makefile created to demonstrate use of the make
# utility in the "Linux Development Platform" book.
#
# Author: Rafeeq Ur Rehman
#         rr@conformix.com
###################################################

# Variable definition
```

```
FTPDIR = ftp-dir
TFTPDIR = tftp-dir
DNSDIR = dns-dir
COMDIR = common-dir

SUBDIRS = $(COMDIR) $(FTPDIR) $(TFTPDIR) $(DNSDIR)

# Default Target

all:
  @echo
  @echo "#######################################"
  @echo "###        BUILDING ALL TARGETS     ###"
  @echo "#######################################"
  @echo
  for i in $(SUBDIRS) ; do   \
         ( cd $$i ; make ) ;        \
  done

# Rules to build individual targets

libs:
  @cd $(COMDIR) ; make

ftp:
  @cd $(FTPDIR) ; make

tftp:
  @cd $(TFTPDIR) ; make

dnsresolver:
  @cd $(DNSDIR) ; make

clean:
  rm -f *~
  for i in $(SUBDIRS) ; do   \
         ( cd $$i ; make clean) ;  \
        done

install:
  for i in $(SUBDIRS) ; do   \
         ( cd $$i ; make install); \
  done
```

N O T E Please note that the following lines:

```
ftp:
  @cd $(FTPDIR) ; make
```

are not equal to the following three lines:

```
ftp:
  @cd $(FTPDIR)
make
```

In the first case, `make` changes the directory to `$(FTPDIR)` and then executes the `make` command in that directory, which is the right thing to do. However in the second case, the cd command is executed and after that the next `make` command is again executed in the current directory. The effect of the cd command is lost when `make` goes to the next line to execute the next command. This is because of a new instance of sub-shell. that executes commands in each line as discussed earlier in this chapter.

After defining variables, we have a rule for each target. This rule basically has two commands on the same line. The first command, `cd,` is used to change the directory where source files for that target are located. In the second command, `make` uses makefile in that directory to build that target. Please note that we can also use the `$(MAKE)` variable for this purpose.

In the `clean` and `install` rules, we use the for loop to go into each directory and execute some commands. The `for` loop is explained later in this chapter.

4.4.2 Makefile in common-dir Directory

The files in the `common-dir` directory are used to build a simple library that is used by other targets at link time. The makefile in this directory is listed below:

```
# Variable definition
SRCS = common.c
OBJS = common.o
HDRS = common.h
LIBCOMMON = libcommon.a
INSTALLDIR = /root

CC = gcc
CFLAGS = -g -O2 -c

# Default Target

$(LIBCOMMON): $(SRCS) $(HDRS)
  $(CC) $(CFLAGS) common.c
  ar -cr $(LIBCOMMON) $(OBJS)
  ranlib $(LIBCOMMON)
```

```
install:
  cp $(LIBCOMMON) $(INSTALLDIR)
clean:
  rm -f $(OBJS) $(LIBCOMMON) *~
```

This makefile builds an archive library file `libcommon.a`, which is the default target. Note that this makefile can also be used as a standalone in this directory so that if someone is working only on the library part of the project, he/she can use this makefile to test only the compilation and library building process. This is useful because each developer can stay in his/her own directory without building all parts of the project from the main makefile in the top directory.

4.4.3 Makefile in the ftp-dir Directory

The makefile in the `ftp-dir` directory builds the `ftp` target. It compiles and then statically links the object files using the library we built in `common-dir` directory. This makefile is shown below.

```
# Variable definition
SRCS = ftp.c
OBJS = ftp.o
HDRS = ftp.h

CC = gcc
CFLAGS = -g -O2 -c
INCLUDES = -I../common-dir
LDFLAGS = -static -L$(LIBSDIR)
LIBSDIR = ../common-dir
INSTALLDIR = /root

# Default Target

ftp: $(SRCS) $(HDRS)
  $(CC) $(CFLAGS) $(INCLUDES) ftp.c
  $(CC) $(LDFLAGS) $(COMMON) $(OBJS) -lcommon -o ftp

install:
  cp ftp $(INSTALLDIR)

clean:
  @echo "Deleting files ..."
  rm -f ftp $(OBJS) *~
```

4.4.4 Makefile in the tftp-dir Directory

The makefile in the `tftp-dir` directory builds the `tftp` target. It compiles and then statically links the object files using the library we built in the `common-dir` directory. This makefile is shown below. It also has rules to install the target and clean the directory.

```
# Variable definition
SRCS = tftp.c
OBJS = tftp.o
HDRS = tftp.h

CC = gcc
CFLAGS = -g -O2 -c
INCLUDES = -I../common-dir
LIBSDIR = ../common-dir
LDFLAGS = -static -L$(LIBSDIR)
INSTALLDIR = /root

# Default Target

tftp: $(SRCS) $(HDRS)
  $(CC) $(CFLAGS) $(INCLUDES) tftp.c
  $(CC) $(LDFLAGS) $(COMMON) $(OBJS) -lcommon -o tftp

install:
  cp tftp $(INSTALLDIR)

clean:
  @echo "Deleting files ..."
  rm -f tftp $(OBJS) *~
```

4.4.5 Makefile in the dns-dir Directory

The makefile in the `dns-dir` directory builds the `dnsresolver` target. It compiles and then statically links the object files using the library we built in the `common-dir` directory. This makefile is shown below.

```
# Variable definition
SRCS = dnsresolver.c
OBJS = dnsresolver.o
HDRS = dnsresolver.h

CC = gcc
CFLAGS = -g -O2 -c
INCLUDES = -I../common-dir
LIBSDIR = ../common-dir
LDFLAGS = -static -L$(LIBSDIR)
INSTALLDIR = /root
```

```
# Default Target

dnsresolver: $(SRCS) $(HDRS)
  $(CC) $(CFLAGS) $(INCLUDES) dnsresolver.c
  $(CC) $(LDFLAGS) $(COMMON) $(OBJS) -lcommon -o dnsresolver

install:
  cp dnsresolver $(INSTALLDIR)

clean:
  @echo "Deleting files ..."
  rm -f dnsresolver $(OBJS) *~
```

4.4.6 Building Everything

After going through these makefiles, you are ready to build the targets. Go to the top directory and run the make command from there. It will read the makefile in the top directory and will try to build all targets. A typical output of this action is as follows:

```
[root@conformix make]# make

#######################################
###        BUILDING ALL TARGETS     ###
#######################################

for i in common-dir ftp-dir tftp-dir dns-dir  ; do   \
         ( cd $i ; make ) ;       \
done
make[1]: Entering directory `/root/make/common-dir'
gcc -g -O2 -c common.c
ar -cr libcommon.a common.o
ranlib libcommon.a
make[1]: Leaving directory `/root/make/common-dir'
make[1]: Entering directory `/root/make/ftp-dir'
gcc -g -O2 -c -I../common-dir ftp.c
gcc -static -L../common-dir  ftp.o  -lcommon -o ftp
make[1]: Leaving directory `/root/make/ftp-dir'
make[1]: Entering directory `/root/make/tftp-dir'
gcc -g -O2 -c -I../common-dir tftp.c
gcc -static -L../common-dir  tftp.o  -lcommon -o tftp
make[1]: Leaving directory `/root/make/tftp-dir'
make[1]: Entering directory `/root/make/dns-dir'
gcc -g -O2 -c -I../common-dir dnsresolver.c
gcc -static -L../common-dir  dnsresolver.o  -lcommon -o
dnsresolver
make[1]: Leaving directory `/root/make/dns-dir'
[root@conformix make]#
```

During the process of building all targets, you can see how make enters and leaves each directory and builds targets in each directory.

4.4.7 Cleaning Everything

The cleaning process is done the same way as we built all targets. Output of this process is shown below. Again you can see that make enters each directory, runs the make clean command and then leaves the directory. The clean rule in makefiles present in each subdirectory is used to remove files.

```
[root@conformix make]# make clean
rm -f *~
for i in common-dir ftp-dir tftp-dir dns-dir  ; do   \
         ( cd $i ; make clean) ;  \
        done
make[1]: Entering directory `/root/make/common-dir'
rm -f common.o  libcommon.a *~
make[1]: Leaving directory `/root/make/common-dir'
make[1]: Entering directory `/root/make/ftp-dir'
Deleting files ...
rm -f ftp ftp.o  *~
make[1]: Leaving directory `/root/make/ftp-dir'
make[1]: Entering directory `/root/make/tftp-dir'
Deleting files ...
rm -f tftp tftp.o  *~
make[1]: Leaving directory `/root/make/tftp-dir'
make[1]: Entering directory `/root/make/dns-dir'
Deleting files ...
rm -f dnsresolver dnsresolver.o  *~
make[1]: Leaving directory `/root/make/dns-dir'
[root@conformix make]#
```

The make clean command finishes its operation after going through all subdirectories.

4.4.8 Making Individual Targets

Instead of using a single big makefile, you can also build individual targets using smaller makefiles. While building only one target, make will go into only that directory and build that target. The following output shows the output of the make command when you build only the ftp target.

```
[root@conformix make]# make ftp
make[1]: Entering directory `/root/make/ftp-dir'
gcc -g -O2 -c -I../common-dir ftp.c
gcc -static -L../common-dir  ftp.o  -lcommon -o ftp
make[1]: Leaving directory `/root/make/ftp-dir'
[root@conformix make]#
```

4.5 Special Features of **make**

In addition to the normal use of the make utility, it has some special features. Two of these features are discussed here that may be of interest for a common user of make. These features are running make commands in *parallel* and running make in *non-stop* mode.

Running multiple commands in parallel enhances the efficiency of make. Running make in non-stop mode is useful for very large projects where you want make to go through everything while running a command without stopping, even in case of errors. You can redirect the output of make to a log file which can be viewed later to find out what error occurred during the make processes of the whole project.

4.5.1 Running Commands in Parallel

Usually make runs commands in serial fashion. This means that one command is executed and when that command is finished, the next command is executed. You can ask make to run many commands in parallel. This is especially useful in multi-processor systems to make execution fast. The only problem with this is that when multiple commands are running in parallel, output from different commands is displayed simultaneously and can get mixed up. To run multiple commands in parallel, use the –j (jobs) command line switch with make.

If you want to specify a maximum number of concurrent commands, a number with the –j switch may be specified. For example, -j5 on the command line will force make to invoke at the maximum five commands simultaneously. It is useful to note that in general the most efficient builds can be done with -j equal to one or two times the total number of processors on the system. Also note that make rules are followed such that all dependencies are satisfied before a target is built. This means you won't always have the maximum number of jobs running simultaneously, depending upon the way makefile is written.

4.5.2 Non-Stop Execution

The make utility executes each command while building a target. After executing the command, it checks the result of that command. If the command was executed successfully, it goes to the next command and executes it. However if the command does not execute successfully, make exits without executing any further commands. It displays an error message to show that the target can't be built.

Sometime, however, it is not important that each command should succeed. As an example, look at the following lines in the clean rule.

```
clean:
  rm ftp
  rm ftp.o common.o
```

If the ftp file is not there, the rm command will return a failure code. However we still want to remove other files using the rm command in the next line. If the rule is listed in this way and the ftp file is not present, the output of the make clean will be as follows:

```
[root@conformix make]# make clean
rm ftp
rm: cannot remove `ftp': No such file or directory
make: *** [clean] Error 1
[root@conformix make]#
```

Now you can see that `make` did not attempt to execute the second `rm` command. To overcome this situation, you can use a hyphen at the start of the command whose result code should not be checked. This way `make` displays an error message but continues to the next command. The same rule with hyphen added to the first `rm` command will be as follows:

```
clean:
  -rm ftp
  rm ftp.o common.o
```

Now when you invoke this rule and `ftp` file is not present, `make` will do something like the following:

```
[root@conformix make]# make clean
rm ftp
rm: cannot remove `ftp': No such file or directory
make: [clean] Error 1 (ignored)
rm ftp.o common.o
[root@conformix make]#
```

As you can see from the above output, `make` ignored the error and continued to the next line to remove object files.

There are other ways to ignore errors during the `make` process. You can also use the `-i` command line switch to ignore errors during the `make` process. Another method is to use `.IGNORE` special built-in rule to ignore errors.

4.6 Control Structures and Directives

Control structures inside makefiles enable you to make some logical decisions depending upon certain conditions. `Make` supports four types of decision-making directives as listed below:

1. The `ifeq` directive
2. The `ifneq` directive
3. The `ifdef` directive
4. The `ifndef` directive

These directives are explained next. In addition to these directives, you can also use the `for` control structure to execute a loop a specified number of times. Let us have a look at each of these.

4.6.1 The ifeq Directive

The `ifeq` directive is used to compare two values and make a decision based upon the result. The general format of this directive is as follows:

```
ifeq (value1, value2)
  block if value 1 is equal to value2
else
  block if value 1 is not equal to value2
endif
```

The `else` part is optional and may not be present. This structure is useful if you want to make a decision on the basis of some criteria. For example, based upon the value of type of build (temporary or final) you want to use different levels of optimization. The following code in a makefile does that.

```
ifeq ($(BUILD), final)
  $(CC) -c -O2 ftp.c
else
  $(CC) -c -O1 ftp.c
endif
```

Please note that there is no TAB character before `ifeq`, `else` and `endif` words.

4.6.2 The ifneq Directive

The `ifneq` directive is similar to `ifeq` directive. The only difference is that sense of equality is reversed. The general syntax is as follows:

```
ifneq (value1, value2)
  block if value 1 is not equal to value2
else
  block if value 1 is equal to value2
endif
```

4.6.3 The ifdef Directive

The `ifdef` directive checks if the value of the variable is empty or not. If the variable is not empty, the `ifdef` block is executed, otherwise the else part is executed. The general structure of the directive is as follows:

```
ifdef  variable
  block if variable is non-empty
else
  block if variable is empty
endif
```

This directive is useful to verify if a variable is defined or not. The `else` part is optional.

CH04.fm Page 133 Monday, October 7, 2002 8:54 PM

4.6.4 The ifndef Directive

The ifndef directive is similar to the ifdef directive. However selection of a block of commands is reversed as compared to ifdef directive. Its format is as follows:

```
ifndef  variable
  block if variable is empty
else
  block if variable is not empty
endif
```

4.6.5 The for Control Structure

You have already used the for directive in the example of using make with multiple directories. This directive is used to perform repeated operation on multiple items. The following rule in the makefile is used to go into multiple directories and remove object files in each of these directories:

```
SUBDIRS = ftp-dir tftp-dir common-dir
clean:
  for dir in $(SUBDIRS) ; do  \
         ( cd $$dir ; rm *.o) ;  \
done
```

Each time a new value is assigned to variable dir from the values listed by the SUBDIRS variable until all of the values are used.

4.7 Getting the Latest Version and Installation

Almost all of the Linux distributions come with make. The Red Hat 7.1 distribution that we have used for this book already has the latest version of make. However if you want to get the latest version of GNU make, it is available at the following FTP site.

```
ftp://ftp.gnu.org/pub/gnu/make/
```

This FTP site has multiple versions of the make utility. Download the latest version, which will be a zipped tar file. The latest version at the time of writing this book is 3.79.1 and the filename is make-3.79.1.tar.gz.

4.7.1 Compilation

After downloading this file, just un-tar it into a directory using the following command:

```
tar zxvf make-3.79.1.tar.gz
```

The source files will be extracted into directory make-3.79.1. Change to this directory and run the configure script. It will create the makefiles appropriate for you. You should have some previous version of make in order to build the latest version. Use the make command; the default rule contains information about how to build make. Use the following command to verify that the build process for make was successful. This command will show a long list of tests and the output is truncated to save space.

```
[root@conformix make-3.79.1]# ./make check
Making check in i18n
make[1]: Entering directory `/opt/make-3.79.1/i18n'
make[1]: Nothing to be done for `check'.
make[1]: Leaving directory `/opt/make-3.79.1/i18n'
make[1]: Entering directory `/opt/make-3.79.1'
/opt/make-3.79.1/./make  check-local
make[2]: Entering directory `/opt/make-3.79.1'
cd tests && perl ./run_make_tests.pl -make ../make
-------------------------------------------------------
   Running tests for GNU make on Linux conformix.net 2.4.2-2
i686
                           GNU Make version 3.79.1
-------------------------------------------------------

Clearing work...
Finding tests...

features/comments ............................ ok
features/conditionals ........................ ok
features/default_names ....................... ok
features/double_colon......................... ok
features/echoing ............................. ok
features/errors .............................. ok
features/escape .............................. ok
features/include ............................. ok
```

If these tests are successful, you have created the latest and greatest make utility. If some of the tests fail, it is an indication that some of the features are missing from this build of make.

4.7.2 Installation

You can install the newly created make utility using the following command.

```
make install
```

The command should be executed in the same directory in which you did the build process.

4.8 References and Resources

1. GNU Make, Richard M. Stallman and Ronald McGrath, Free Software Foundation, ISBN:1-8822114-80-9. It is also available on many websites in Postscript form.
2. GNU make web site at http://www.gnu.org/software/make/make.html
3. GNU make manual at http://www.gnu.org/manual/make/index.html