Birzeit University - Faculty of Engineering & Technology
Electrical & Computer Engineering Department - ENCS313
Linux laboratory

**Experiment #6**
**Compilation, Debugging, Project Management**
**Part III**

## 0.1 Introduction

The experiment intends to present the `gcc` compiler, the `gdb` debugger and show students how to use `makefile`s to better manage projects. Students are assumed to be already familiar with the C-language. Students will be shown how to compile source code and create executables using the `gcc` compiler. Moreover, students will be shown how to create libraries (both static libraries and dynamic libraries) using the archive command `ar` and how to link to these libraries during the compilation phase. They will be shown how to use the `gdb` debugger in order to debug code and discover defects (or potential defects). Finally, students will be shown how to use a `makefile` that will help better manage a development project during compilation, setup and installation.

## 0.2 Objectives

The objectives of the experiment is to learn the following:

- Give a quick overview of the GNU project.
- Show students on how to use the `gcc` compiler and the `ar` archive command.
- Show students on how to use the `gdb` debugger.
- Show students on how to use a `makefile` to manage a development project.

## 0.3 Managing projects with make

Compiling your source code files can be tedious, specially when you want to include several source files and have to type the compiling command everytime you want to do it[1]. In addition, real-life projects might end up being composed of tens (if not hundreds) of source files. Some of these files will be dependant on some other files. Thus, doing the compilation by hand for every individual source file and keeping track of dependancies is not just a hard task, but might be problem-generator.

In software development, `make` is a utility that automatically builds executable programs and libraries from source code by reading files called `makefiles` which specify how to derive the target program[2].

Makefiles are special format files that together with the `make` utility will help you to automatically build and manage your projects.

The advantages of `make` over scripts is that you can specify the relationships between the elements of your program to `make`, and it knows through these relationships and timestamps exactly what steps need to be redone to produce the desired program each time. Using this

[1] http://mrbook.org/tutorials/make/
[2] http://en.wikipedia.org/wiki/Make_(software)

information, `make` can also optimize the build process avoiding unnecessary steps[3].

`make` defines a language for describing the relationships between source code, intermediate files, and executables. It also provides features to manage alternate configurations, implement reusable libraries of specifications, and parameterize processes with user-defined macros. In short, `make` can be considered the center of the development process by providing a roadmap of an application's components and how they fit together.

### 0.3.1 Basic example

As a warm up example of using command `make` in association with a `makefile`, refer to the below example.

```
#include <stdio.h>

/* print the message Hello, World! on standard output */

int main()
{
  printf("Hello, World!\n");
  return(0);
}
```

Save the above code in a file called `hello.c`.

Type the following text in a file called `makefile`:

```
hello: hello.c
       gcc hello.c -o hello
```

In order to compile it the file `hello.c`, run the command `make`. Check that the executable `hello` has been generated. Run it and make sure you get the right output message.

Run the command `make` and note that no extra compilation takes place. In fact, you'll get the following message on the standard output:

```
make:  'hello' is up to date.
```

The sequence of actions that took place when `make` was invoked can be summarized as follows:

- The command `make` reads the text file `makefile`.
- `make` encounters the target `hello`.
- `make` notes that the target `hello` depends on the source file `hello.c`. `make` checks that the source file `hello.c` exists in the current folder.
- `make` executes the action that is associated with the target, namely `gcc hello.c -o hello`

### 0.3.2 Targets and Prerequisites

Essentially a `makefile` contains a set of rules used to build an application. The first rule seen by `make` is used as the default rule. A rule consists of three parts: the target, its prerequisites, and the command(s) to perform:

```
target: prereq1 prereq2
        commands
```

[3] **Managing Projects with GNU make, 3rd Edition** - By Robert Mecklenburg - O'Reilly

The `target` is the file or thing that must be made. The `prerequisites` or `dependents` are those files that must exist before the target can be successfully created. And the `commands` are those shell commands that will create the target from the prerequisites.

In the example above we saw a simplified form of a rule. The more complete (but still not quite complete) form of a rule is:

```
target1 target2 target3 : prerequisite1 prerequisite2
      command1
      command2
      command3
```

One or more targets appear to the left of the colon and zero or more prerequisites can appear to the right of the colon.

**Note 1:**

Each command must begin with a tab character. This (obscure) syntax tells `make` that the characters that follow the tab are to be passed to a subshell for execution. If you accidentally insert a tab as the first character of a noncommand line, `make` will interpret the following text as a command under most circumstances. In some cases, you might get the following error after running command `make`:

```
Makefile:6: *** commands commence before first target. Stop.
```

**Note 2:**

The comment character for `make` is the hash or pound sign (#). All text from the pound sign to the end of line is ignored.

**Note 3:**

Long lines can be continued using the standard Unix escape character backslash (\). It is common for commands to be continued in this way. It is also common for lists of prerequisites to be continued with backslash.

**Example:**

Consider the following example:

```
/*********************************************************************
 *                        File count_words.c                         *
 *********************************************************************/

#include <stdio.h>
#include <counter.h>

int main( int argc, char ** argv )
{
    int counts[4];
    counter( counts );
    printf( "%d %d %d %d\n", counts[0], counts[1], counts[2], counts[3] );
    exit( 0 );
}
```

Save the above code in a file called `count_words.c`.

```
/*********************************************************************
 *                           File counter.c                          *
 *********************************************************************/
```

```
#include "counter.h"

void counter( int counts[4] )
{
  counts[0] = VALUE_1;
  counts[1] = VALUE_2;
  counts[2] = VALUE_3;
  counts[3] = VALUE_4;
}
```

Save the above code in a file called `counter.c`.

```
/*********************************************************************
 *                            File counter.h                         *
 *********************************************************************/

#define VALUE_1 1
#define VALUE_2 2
#define VALUE_3 3
#define VALUE_4 4
```

Save the above code in a file called `counter.h`.

```
#####################################################################
#                 This is an example of a Makefile                  #
#####################################################################

# This is an example of a Makefile

# Indicate that the compiler is the gcc compiler
CC = gcc

# Indicate to the compiler to include header files in the local folder
CPPFLAGS = -I .

count_words: counter.o
count_words.o: counter.h
counter.o: counter.h
```

Save the above instructions in a file called `Makefile`.

Compile now using the command `make` and check that an executable called `count_words` get generated. Run the executable `count_words` and check on the results that you get.

Notice that if you call the command `make` again, the executable will not get re-generated since `make` knows that the source code was not modified.

Now modify the values included in file `counter.h` and run the command `make` again. You will notice that the executable `count_words` will be re-generated since it depends on the file `counter.h`.

Note that the variables `CC` and `CPPFLAGS` are special flags that the command `make` knows how to interpret. The line `CC = gcc` indicates to command `make` to use the `gcc` compiler instead of the default `cc` compiler while variable `CPPFLAGS` indicates to command `make` to include the current folder while searching for header files.

## 0.4 Rules

We've seen above some rules to compile and link a program. Each of those rules defines a `target`, that is, a file to be updated. Each target file depends on a set of `prerequisites`, which are also files. When asked to update a target, `make` will execute the command script of the rule if any of the prerequisite files has been modified more recently than the target. Since the `target` of one rule can be referenced as a `prerequisite` in another rule, the set of targets and prerequisites form a chain or graph of *dependencies* (short for "dependency graph"). Building and processing this dependency graph to update the requested target is what `make` is all about.

`make` can use the following list of rules:

- **Explicit rules**: These rules are like the ones we've seen above. They indicate a specific target to be updated if it is out of date with respect to any of its prerequisites. This is the most common type of rule you will be writing.
- **Pattern rules**: These rules use wildcards instead of explicit filenames. This allows `make` to apply the rule any time a target file matching the pattern needs to updated.
- **Implicit rules**: These rules are either pattern rules or suffix rules found in the rules database built-in to make. Having a built-in database of rules makes writing makefiles easier since for many common tasks make already knows the file types, suffixes, and programs for updating targets.
- **Static pattern rules**: These rules are like regular pattern rules except they apply only to a specific list of target files.

### 0.4.1 Explicit rules

An explicit rule can have more than one target. This means that each target has the same set of prerequisites as the others. If the targets are out of date, the same set of actions will be performed to update each one. As an example, consider the following content of a `makefile`:

```
vpath.o variable.o: make.h config.h getopt.h gettext.h dep.h
```

This indicates that both `vpath.o` and `variable.o` depend on the same set of C header files. This line has the same effect as:

```
vpath.o: make.h config.h getopt.h gettext.h dep.h
variable.o: make.h config.h getopt.h gettext.h dep.h
```

A rule does not have to be defined "all at once". Each time `make` sees a target file it adds the target and prerequisites to the dependency graph. If a target has already been seen and exists in the graph, any additional prerequisites are appended to the target file entry in `make`'s dependency graph. In the simple case, this is useful for breaking long lines naturally to improve the readability of the makefile. As an example, consider the following:

```
vpath.o: vpath.c make.h config.h getopt.h gettext.h dep.h
vpath.o: filedef.h hash.h job.h commands.h variable.h vpath.h
```

### 0.4.2 Variables

In a `makefile`, you can use variables in exactly the same way you have been defining and using environment variables. You can define a variable like:

```
COMPILER = gcc
```

In your `makefile`, you can refer to the content of variable `COMPILER` as:

```
$(COMPILER)
```

A `makefile` will typically define many variables, but there are also many special variables defined automatically by `make`. Some can be set by the user to control `make`'s behavior while others are set by `make` to communicate with the user's `makefile`.

Here is a short-list table of variables used as names of programs in built-in rules:

| | |
|---|---|
| `AR` | Archive-maintaining program; default `ar` |
| `AS` | Program for doing assembly; default `as` |
| `CC` | Program for compiling C programs; default `cc` |
| `CXX` | Program for compiling C++ programs; default `g++` |
| `CO` | Program for extracting a file from RCS; default `co` |
| `CPP` | Program for running the C preprocessor, with results to standard output; default `$(CC) -E` |
| `LEX` | Program to use to turn Lex grammars into C programs or Ratfor programs; default `lex` |
| `YACC` | Program to use to turn Yacc grammars into C programs; default `yacc` |
| `RM` | Command to remove a file; default `rm -f` |

Below is a short-list of variables whose values are additional arguments for the programs above. The default values for all of these is the empty string, unless otherwise noted.

| | |
|---|---|
| `ARFLAGS` | Flags to give the archive-maintaining program; default `rv` |
| `ASFLAGS` | Extra flags to give to the assembler (when explicitly invoked on a `.s` or `.S` file) |
| `CFLAGS` | Extra flags to give to the C compiler |
| `CXXFLAGS` | Extra flags to give to the C++ compiler |
| `COFLAGS` | Extra flags to give to the RCS `co` program |
| `CPPFLAGS` | Extra flags to give to the C preprocessor and programs that use it (the C and Fortran compilers) |
| `FFLAGS` | Extra flags to give to the Fortran compiler |
| `GFLAGS` | Extra flags to give to the SCCS `get` program |
| `LDFLAGS` | Extra flags to give to compilers when they are supposed to invoke the linker, `ld` |
| `LFLAGS` | Extra flags to give to Lex |
| `YFLAGS` | Extra flags to give to Yacc |

### 0.4.3 Automatic Variables

*Automatic variables* are set by `make` after a rule is matched. They provide access to elements from the target and prerequisite lists so you don't have to explicitly specify any filenames. They are very useful for avoiding code duplication.

There are six "core" automatic variables:

- `$@` The filename representing the target.

- `$%` The filename element of an archive member specification.
- `$<` The filename of the first prerequisite.
- `$?` The names of all prerequisites that are newer than the target, separated by spaces.
- `$^` The filenames of all the prerequisites, separated by spaces. This list has duplicate filenames removed since for most uses, such as compiling, copying, etc., duplicates are not wanted.
- `$+` Similar to $^, this is the names of all the prerequisites separated by spaces, except that $+ includes duplicates.

**Example:**

We will consider below an example similar to the one seen above. Execute the following steps:

- Create a folder called `ex2` and execute the command `cd ex2`.
- Under folder `ex2`, create the folders `src` and `include`.
- Under folder `src`, copy the previous C-files `counter.c` and `count_words.c`.
- Under folder `include`, copy the previous header file `counter.h`.
- Save the below text in a file called `Makefile`:

```
#########################################################################
#          This is a Makefile that uses automatic variables             #
#########################################################################

# Indicate that the compiler is the gcc compiler
CC = gcc

# Indicate to the compiler to include header files in folder include
CPPFLAGS = -I include

# Indicate to the compiler to look for source files and header files
# under folders "src" and "include"
VPATH   = src include

count_words: count_words.o counter.o
$(CC) $^ -o $@

count_words.o: count_words.c counter.h
$(CC) $(CPPFLAGS) -c $< -o $@

counter.o: counter.c counter.h
$(CC) $(CPPFLAGS) -c $< -o $@
```

- Run the command `make` and make sure that the executable `count_words` gets generated.
- Run the executable `count_words` and make sure you get the same output as in the previous example.

### 0.4.4 Pattern rules

The makefiles we've seen above are a bit simple. For a small program of a dozen files or less we may not care, but for programs with hundreds or thousands of files, specifying each target, prerequisite, and command script becomes unworkable.

Many programs that read one file type and output another conform to standard conventions. For instance, all C compilers assume that files that have a .c suffix contain C source code and that the object filename can be derived by replacing the .c suffix with .o (or .obj for some Windows compilers).

*A pattern rule* looks like the normal rules you have already seen except the stem of the file (the portion before the suffix) is represented by a % character. The following makefile specifies how to compile a .o file from a .c file:

```
%.o: %.c
     $(COMPILE) $(OUTPUT_OPTION) $<
```

In the following example, the rule specifies how to create a C-file from a flex file:

```
%.c: %.l
     $(LEX.l) $< > $@
```

Finally, there is a special rule to generate a file with no suffix (always an executable) from a .c file:

```
%: %.c
   $(LINK) $^ $(LOADLIBES) $(LDLIBS) -o $@
```

**Example:**

We will consider the example that we've seen above and modify its `makefile` to include pattern rules. Execute the following steps:

- Create a folder called `ex3` and execute the command `cd ex3`.
- Copy the content of folder `ex2` seen above by issuing the command:
  `cp -r ../ex2/* .`
- Remove the content of the `makefile` that was just copied from folder `ex2` and replace it by the following text:

```
#########################################################################
#             This is a Makefile that uses pattern rules                #
#########################################################################

# Indicate that the compiler is the gcc compiler
CC = gcc

# Indicate to the compiler to include header files in folder include
CPPFLAGS = -I include

# Indicate to the compiler to look for source files and header files
# under folders "src" and "include"
VPATH   = src include

OBJECTS := count_words.o counter.o

count_words: $(OBJECTS)
```

```
$(CC) $^ -o $@

%.o: %.c %.h
$(CC) $(CPPFLAGS) -c $<
```

- Run the command `make` and make sure that the executable `count_words` gets generated.
- Run the executable `count_words` and make sure you get the same output as in the previous example.

### 0.4.5 Implicit rules

`make` has around 90 built-in implicit rules. There are built-in pattern rules for C, C++, Pascal, FORTRAN and many other programming languages. In addition, there are rules for supporting programs for these languages, such as `cpp`, `as`, `yacc`, `lex`, and dvi tools.

The built-in implicit rules are applied whenever a target is being considered and there is no explicit rule to update it. So using an implicit rule is easy: simply do not specify a command script when adding your target to the makefile. This causes make to search its built-in database to satisfy the target.

**Example**

Suppose you have a makefile that looks like the following:

```
foo : foo.o bar.o
            cc -o foo foo.o bar.o
```

Note that you mention the object file `foo.o` but do not give a rule for it. `make` will automatically look for an implicit rule that tells how to update it. This happens whether or not the file `foo.o` currently exists.

## 0.5 Special Targets

A *special target* is a built-in target used to change `make`'s default behavior. There are twelve special targets. They fall into three categories: as we've just said many are used to alter the behavior of `make` when updating a target, another set act simply as global flags to `make` and ignore their targets, finally the `.SUFFIXES` special target is used when specifying old-fashioned suffix rules.

Some of the special targets are:

- `.PHONY`: It declares that its prerequisite does not refer to an actual file and should always be considered out of date. The .PHONY target is the most common special target you will see.

  **Example:**

  We will re-consider the example that included the files `count_words.c`, `counter.c` and `counter.h` that we've seen above with a minor modification in the `makefile`. Execute the following steps:

  - Create a folder called `ex4` and execute the command `cd ex4`.
  - Copy the content of folder `ex3` seen above by issuing the command:
    `cp -r ../ex3/* .`
  - Add the following line:

    ```
    .PHONY: count_words
    ```

    at the end of the `makefile`. The `makefile` will look like:

```
#########################################################################
#           This is a Makefile example with .PHONY target               #
#########################################################################

# Indicate that the compiler is the gcc compiler
CC = gcc

# Indicate to the compiler to include header files in folder include
CPPFLAGS = -I include

# Indicate to the compiler to look for source files and header files
# under folders "src" and "include"
VPATH   = src include

OBJECTS := count_words.o counter.o

count_words: $(OBJECTS)
$(CC) $^ -o $@

%.o: %.c %.h
$(CC) $(CPPFLAGS) -c $<

.PHONY: count_words
```

  - Run the command `make` and make sure that the executable `count_words` gets generated. Notice that every time you call the command `make`, the executable `count_words` gets re-generated.
  - Run the executable `count_words` and make sure you get the same output as in the previous example.

- `.INTERMEDIATE`: Prerequisites of this special target are treated as intermediate files. If `make` creates the file while updating another target, the file will be deleted automatically when `make` exits. If the file already exists when `make` considers updating the file, the file will not be deleted.

  **Example:**

  We will re-consider the example that included the files `count_words.c`, `counter.c` and `counter.h` that we've seen above with a minor modification in the `makefile`. Execute the following steps:

  - Create a folder called `ex5` and execute the command `cd ex5`.
  - Copy the content of folder `ex4` seen above by issuing the command:
    `cp -r ../ex4/* .`
  - Remove the line:

    `.PHONY: count_words`

    and add the following line:

    `.INTERMEDIATE: $(OBJECTS)`

    at the end of the `makefile`. The `makefile` will look like:

```
#########################################################################
#        This is a Makefile example with .INTERMEDIATE target          #
#########################################################################
```

```
# Indicate that the compiler is the gcc compiler
CC = gcc

# Indicate to the compiler to include header files in folder include
CPPFLAGS = -I include

# Indicate to the compiler to look for source files and header files
# under folders "src" and "include"
VPATH    = src include

OBJECTS := count_words.o counter.o

count_words: $(OBJECTS)
        $(CC) $^ -o $@

%.o: %.c %.h
        $(CC) $(CPPFLAGS) -c $<

.INTERMEDIATE: $(OBJECTS)
```

  - Run the command `make` and make sure that the executable `count_words` gets generated. Notice that before `make` finishes its job, the 2 object files `count_words.o` and `counter.o` are removed since they are declared as intermediate files.
  - Run the executable `count_words` and make sure you get the same output as in the previous example.

- `.SECONDARY`: Prerequisites of this special target are treated as intermediate files but are never automatically deleted. The most common use of `.SECONDARY` is to mark object files stored in libraries. Normally these object files will be deleted as soon as they are added to an archive. Sometimes it is more convenient during development to keep these object files, but still use the `make` support for updating archives.

- `.PRECIOUS`: When `make` is interrupted during execution, it may delete the target file it is updating if the file was modified since make started. This is so `make` doesn't leave a partially constructed (possibly corrupt) file laying around in the build tree. There are times when you don't want this behavior, particularly if the file is large and computationally expensive to create. If you mark the file as precious, `make` will never delete the file if interrupted.

- `.DELETE_ON_ERROR`: This is sort of the opposite of `.PRECIOUS`. Marking a target as `.DELETE_ON_ERROR` says that `make` should delete the target if any of the commands associated with the rule generates an error.

### 0.5.1 Searching Directories for Prerequisites

For large systems, it is often desirable to put sources in a separate directory from the binaries. The *directory search* features of `make` facilitate this by searching several directories automatically to find a prerequisite. When you redistribute the files among directories, you do not need to change the individual rules, just the search paths[4].

#### `VPATH`: Search Path for All Prerequisites

The value of the `make` variable `VPATH` specifies a list of directories that `make` should search. Most often, the directories are expected to contain prerequisite files that are not in the current

[4] http://www.gnu.org/software/make/manual/make.html

directory; however, `make` uses `VPATH` as a search list for both prerequisites and targets of rules.

Thus, if a file that is listed as a target or prerequisite does not exist in the current directory, `make` searches the directories listed in `VPATH` for a file with that name. If a file is found in one of them, that file may become the prerequisite. Rules may then specify the names of files in the prerequisite list as if they all existed in the current directory.

For example, if the makefile contained the following statement:

```
VPATH = src:../headers
```

that means that the path contains 2 directories: `src` and `../headers` which `make` searches in tha order.

### The `vpath` Directive

Similar to the `VPATH` variable, but more selective, is the `vpath` directive (note lower case), which allows you to specify a search path for a particular class of file names: those that match a particular pattern. Thus you can supply certain search directories for one class of file names and other directories (or none) for other file names.

There are three forms of the `vpath` directive:

- `vpath` *pattern directories*: Specify the search path *directories* for file names that match *pattern*.

  The search path, *directories*, is a list of directories to be searched, separated by colons (semi-colons on MS-DOS and MS-Windows) or blanks, just like the search path used in the `VPATH` variable.

- `vpath` *pattern*: Clear out the search path associated with *pattern*.

- `vpath`: Clear all search paths previously specified with `vpath` directives.

A `vpath` pattern is a string containing a % character. The string must match the file name of a prerequisite that is being searched for, the % character matching any sequence of zero or more characters (as in pattern rules).

When a prerequisite fails to exist in the current directory, if the pattern in a `vpath` directive matches the name of the prerequisite file, then the directories in that directive are searched just like (and before) the directories in the `VPATH` variable.

For example,

```
vpath %.h ../headers
```

tells `make` to look for any prerequisite whose name ends in `.h` in the directory `../headers` if the file is not found in the current directory.

If several `vpath` patterns match the prerequisite file's name, then `make` processes each matching `vpath` directive one by one, searching all the directories mentioned in each directive. `make` handles multiple `vpath` directives in the order in which they appear in the makefile; multiple directives with the same pattern are independent of each other.

Thus,

```
vpath %.c foo
vpath %.c blish
vpath %.c bar
```

will look for a file ending in .c in `foo`, then `blish`, then `bar`, while

```
vpath %.c foo:bar
vpath %.c blish
```

will look for a file ending in .c in `foo`, then `bar`, then `blish`.

### 0.5.2 Building a library in a makefile

We explored what libraries are and how to use them in a previous experiment using the `ar` command. We'll see in the below example how to include building a library in a `makefile`.

Execute the following steps:

- Create a folder called `ex6` and execute the command `cd ex6`.
- Copy the content of folder `ex5` seen above by issuing the command:
  `cp -r ../ex5/* .`
- Remove the content of the `makefile` that was just copied from folder `ex5` and replace it by the following text:

```
######################################################################
#              This is a Makefile that builds a library              #
######################################################################

# Indicate that the compiler is the gcc compiler
CC = gcc

# Indicate to the compiler to include header files in folder include
CPPFLAGS = -I include

# Indicate to the compiler to look for source files and header files
# under folders "src" and "include"
VPATH   = src include

count_words: libcounter.a

libcounter.a: libcounter.a(counter.o)

libcounter.a(counter.o): counter.o
$(AR) $(ARFLAGS) $@ $<

count_words.o: counter.h
counter.o: counter.h
```

- Run the command `make` and make sure that the executable `count_words` gets generated as well as the library `libcounter.a`. Note that the executable `count_words` depends on that library.
- Run the executable `count_words` and make sure you get the same output as in the previous example.

## 0.6 Conditional Parts of Makefiles

A **conditional** causes part of a `makefile` to be ignored depending on the values of variables. Conditionals can compare the value of one variable to another, or the value of a variable to a constant string. Conditionals control what command `make` actually "sees" in the `makefile`

The basic syntax of the conditional directive is:

```
if-condition

    text if the condition is true
```

```
endif
```

or

```
if-condition

    text if the condition is true

else

    text if the condition is false

endif
```

The `if-condition` can be one of the following:

```
ifdef variable-name

ifndef variable-name

ifeq test

ifneq test
```

**Example:**

As an example, consider the following `makefile`:

```
#######################################################################
#               Example of a conditional Makefile                     #
#######################################################################

libs_for_gcc = -lgnu
normal_libs =

foo: $(objects)
ifeq ($(CC),gcc)
        $(CC) -o foo $(objects) $(libs_for_gcc)
else
        $(CC) -o foo $(objects) $(normal_libs)
endif
```

In the above `makefile`, the conditional tells `make` to use one set of libraries if the `CC` variable is `gcc`, and a different set of libraries otherwise. It works by controlling which of two command lines will be used as the command for a rule.

**Example:**

As an additional example on using conditionals with makefiles, consider the following `makefile`:

```
#######################################################################
#            Another example of a conditional Makefile                #
#######################################################################

# Variable COMSPEC is defined only on Windows OS.

ifdef COMSPEC
```

```
  PATH_SEP := ;
  EXE_EXT := .exe
else
  PATH_SEP := :
  EXE_EXT :=
endif
```

In the above `makefile`, the first branch of the conditional is selected if the variable `COMSPEC` is defined.