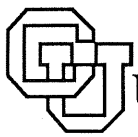
Elements of UNIX make

Carolyn J.C. Schauble

CU-CS-570-92 January 1992



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

Elements of UNIX *make*

Carolyn J. C. Schauble

CU-CS-570-92 January 1992

University of Colorado at Boulder

Technical Report CU-CS-570-92
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

Abstract

Building on fundamental *UNIX* concepts, this document is a tutorial intended to introduce students to the *UNIX* **make** utility. It provides the basic syntax of a **Makefile** and demonstrates the use of **make** from a simple *Fortran* compilation up through the development of a **Makefile** to compile and run modular *Fortran* programs with the additional facility of printing source files and output.

Elements of UNIX *make**

Carolyn J. C. Schauble

January 1992

1 Introduction

This tutorial is intended to introduce you to **make**, one of the most commonly used *UNIX*[1] utilities [Feldman 86]. The purpose of **make** is to help maintain files and keep them current. It can also help reduce keystrokes and eliminate redundant or repetitious commands.

You will learn first how to use **make** for a simple *Fortran* compilation. Later, a method for performing more complex modular compilation with **make** will be shown. Similar techniques can be applied for compiling *C* programs. Additional uses of **make** will also be discussed.

For the examples shown in this document, the commands that you, the reader, enter are displayed in *this font*, e.g.,

make myprog

while the computer responses are displayed in `this font`, e.g.,

```
f77 -O myprog.f -o myprog
```

These examples were run on a DECstation 5000/200 using *Ultrix 4.1*[2] The given responses to the commands may differ slightly on other systems; some of the defaults may be different as well. However, the basic concepts will be the same.

*This work has been supported by the National Science Foundation under an Educational Infrastructure grant, CDA-9017953.

[1]UNIX is a trademark of AT&T.

[2]Ultrix is a trademark of Digital Equipment Corporation.

2 An Example of using *make*

Suppose you have created a *Fortran* program named `myprog.f`. Also suppose that `myprog.f` is the only file in your current directory.

Naturally, the first thing you'll want to do is to compile your new program to check for errors. Normally, you would just type something like

f77 -o myprog myprog.f

and the linked object or executable file for the compiled program (if it compiled successfully) would be put in `myprog`. If you were using **make** instead, you would type

make myprog

and accomplish the same thing. The full dialogue with **make** as displayed on the screen would be as follows:

```
        ⋮
% make myprog
f77 -O myprog.f -o myprog
%
```

In other words, if you ask **make** to *make* a file named `myprog`, **make** is clever enough to know that it has to compile `myprog.f` to do it.

Why? What happens? By what magic does **make** know to compile your program?

2.1 Dependencies

The **make** utility is basically used for file maintenance; it is mainly concerned with the dependencies of one file on another. By one file being dependent upon another, we mean that the first file is generated from the other. For instance, a file with a `.o` extension is the result of a compilation and so depends on a file with the same rootname and an extension which refers to a computer language, such as, `.f` for *Fortran*, `.c` for *C*, `.p` for *Pascal*, or even `.s` for assembly languages. In our example, the linked executable file, `myprog`, is dependent upon the *Fortran* source file, `myprog.f`.

From these known dependencies, **make** generates a series of commands to form a given file, called the *target* file. These commands use the *components* of the target file, also called the *dependency files*, i.e., those files on which the *target* is dependent. This sequence of commands may be known implicitly by **make**. As mentioned above, **make** is aware that a file with a `.o` extension is the result of a compilation. Or the commands may be defined in a *description file*, called a *makefile*, which specifies the dependencies of a target and the commands or *rules* by which to form the *target*. Such a description file is usually named **Makefile** or **makefile**, and may describe rules which are unique to the files in your directory.

2.2 How the *make* Request Works

The command

make myprog

calls the **make** utility with the argument **myprog**. In this case, **myprog** is called the *target* for the given *make* command.

As the first step in the execution of this **make** request, your current directory is searched for the file named **myprog**. If the file were there, it might not need to be re*made*. However, in this example, no file of this name exists.

Next, **make** looks for a file named **Makefile** or **makefile**. Such a file might contain specific instructions to *make* **myprog**.

Since there is no *makefile* in the directory assumed for this example, **make** must use the implicit rules of dependency mentioned above. Thus, it looks for a file that can be used to create the desired file, starting with **myprog.o**. If there was a **myprog.o** file, then **make** would only need to link that file to create the new executable file, **myprog**. Again no such file exists.

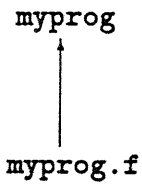


Figure 1: Derivation of **myprog** from **myprog.f**.

However, **make** finds the file named **myprog.f**. It knows that this file can be compiled, using the *Fortran* compiler, to create the file **myprog**. And this is what it does.

Hence, in trying to *make* the target file named **myprog**, **make** finds it must compile the only file in the directory with the correct rootname, **myprog.f**. From this compilation, it can derive the desired file, **myprog**. A picture of this derivation is shown in Fig. 1.

2.3 When *make* Fails

If no files in the directory had the correct rootname, **myprog**, this **make** command would fail. For instance, suppose you had erroneously stored the file as **myprg.f**. In response to the command

make myprog

the **make** utility would produce the error message:

```
Make:  Don't know how to make myprog.  Stop.
```

This is because it was unable to find anything it could logically convert to **myprog**; it was unable to find any files on which **myprog** could be dependent.

As another example, suppose there was a compilation error. Then, again, **make** would not be able to complete its assigned task. This time the error message would occur in a dialogue much like the following:

```
        ⋮
% make myprog
f77 -O myprog.f -o myprog
Error on line 1 of myprog.f:  illegal ....
Stop
%
```

Any time that **make** is unable to create the final desired target file, it will print an error message to that effect. Note that some intermediate files may need to be created in the process and will be present in your directory when **make** is done.

3 Some Advantages of *make*

So, why is the command

make myprog

better than the command

f77 myprog.f

Both commands contain only two words and use about the same number of keystrokes. At this point, there doesn't seem to be that much to recommend switching to **make**.

3.1 Implicit Rules

One interesting point is that the command

make myprog

would have worked just as well if the program in your directory were a *C* program (`myprog.c`), a *Pascal* program (`myprog.p`), or an assembly program (`myprog.s`). The **make** utility would have found a compilable program file from which to create `myprog`. Similarly, the command

make myprog.o

would create an object file named `myprog.o` from `myprog.f`, `myprog.c`, `myprog.p`, or `myprog.s`. The dialogue with **make** for this command would be similar to that below.

```
        ⋮
% make myprog.o
f77 -O -c myprog.f
%
```

3.2 Avoiding Redundant Work

Now suppose that you had not worked with `myprog.f` for some time and were not certain if the copy of `myprog` in the directory included the last changes to `myprog.f`. One way to find out would be to check the creation dates yourself.

```
        ⋮
% ls -l
total 80
-rwxrwxr-x 1 schauble  80516 Jul 2 11:42 myprog*
-rw-r--r-- 1 schauble     19 Jul 2 11:41 myprog.f
%
```

In this case, it appears that myprog was created after the latest changes to myprog.f. If this were not so, you would probably want to recompile the program before reusing it with the command:

f77 -o myprog myprog.f

Checking on the validity of myprog could also be done by using **make** instead. If you typed

```
        ⋮
% make myprog
'myprog' is up to date.
%
```

the effect would be the same as examining the `ls -l` output; you would know that myprog was more recently created than myprog.f, without needing to look at the creation dates yourself. Further, the recompilation would have been done automatically if the executable file was not up-to-date. In other words, **make** will only recompile the program if it is necessary.

This is another advantage of **make**. It not only checks to see if the target file already exists, but also that it is a valid version of that file; that is, **make** assures that the target file is current with respect to the file(s) on which it is dependent.

4 The Makefile

Using the implicit version of **make** has its limitations. For instance, the *Fortran* compilations done by **make** in the examples above used the `-O` optimization flag.[3] You might prefer a different optimization level. How can you override the implicit rules of **make**? By forming your own rules and putting them in a file named **Makefile** or **makefile**.

[3]The default compilation flags for **make** differ from system to system.

4.1 Format of a Sample Makefile

A simple description file or *makefile* for the derivation shown in Fig. 1 might look like the following:

```
#
# Simple Makefile
#    make myprog:  to compile myprog.f
#

myprog:  myprog.f
        f77 -O1 -o myprog myprog.f
```

This consists of some comments and a rule for *mak*ing the target, myprog. Let's look at the parts of this *makefile* in more detail.

The first few lines of the *makefile* are comments. It is useful to include in these comments all the functions a given *makefile* performs.

```
#
# Simple Makefile
#    make myprog:  to compile myprog.f
#
```

Notice that these lines all begin with the special character, '#'. Any characters following an '#' on any line in the *makefile* are assumed to be part of a comment. Blank lines are also ignored by **make**.

The remaining lines of this sample *makefile* define the rule that this *makefile* is providing.

```
myprog:  myprog.f
        f77 -O1 -o myprog myprog.f
```

The rules of a *makefile* have two parts: (i) a *dependency line* which describes the target file with its dependencies and (ii) one or more *command lines* which describe how to form the target file from the dependencies. All commands which form rules for **make** are executed in the *Bourne shell* (**/bin/sh**).

1. The first of the lines above

   ```
   myprog:  myprog.f
   ```

is the dependency line and defines **myprog** as being the *target* file and **myprog.f** as the file on which myprog is dependent. This means that when the command

make myprog

is given, **make** should form the desired file **myprog** according to the rule set in this *makefile*, first checking that the file **myprog.f** exists. If there should be no file named myprog.f in the directory, **make** will complain appropriately.

```
        ⋮
% make myprog
Make:  Don't know how to make myprog.f.  Stop.
%
```

Notice how this differs from the earlier example when **make** was unable to find a file with the correct rootname. Then it was just looking for any file that could be used to derive myprog. When it couldn't find one, it simply responded that it didn't know how to *make* **myprog**. In the current example, **make** is told that the particular file myprog.f is necessary for *mak*ing the target file. So, when it can't find myprog.f, it complains that that particular file, **myprog.f**, is missing.

2. The last line of this *makefile* provides the actual commands (in this case a single command) that **make** is to follow in creating the target file, **myprog**.

```
f77 -O1 -o myprog myprog.f
```

This *Bourne* shell command line tells **make** to use the *Fortran* f77 compiler on the dependency file, myprog.f, with the -O1 optimization flag, storing the resultant executable file as the target file.

> *Comment:* One of the odd rules of **make** is that the lines containing the commands for **make** rules, such as the line discussed above, must be indented by a *Tab*; spaces will not work. Using spaces instead of the *Tab* is a common error in *makefiles*.

4.2 Using a Makefile

If we now try the command

make myprog

with only the files **makefile** and **myprog.f** in the current directory, we get the following response:

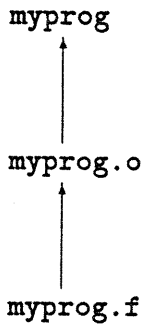


Figure 2: Two-Step Derivation of `myprog` from `myprog.f`.

```
      ⋮
% make myprog
f77 -O1 -o myprog myprog.f
%
```

Clearly, the `-O1` optimization flag is used, as specified in our *makefile* rule, in contrast to the earlier examples which used the implicit `make` rules. The new file `myprog` will be in the current directory.

4.3 Another Sample Makefile

An alternate version of the description file would separate the compile step from the link step, as in the following *makefile.*

```
# Simple 2-Step Makefile

myprog:  myprog.o
        f77 -o myprog myprog.o

myprog.o:  myprog.f
        f77 -O1 -c myprog.f
```

This *makefile* has two rules replacing the one of the previous *makefile.* The derivation tree for this description file is shown in Fig. 2.

Given the command

make myprog

make sees that myprog is dependent on myprog.o, which is not in the directory. But it finds that there is a rule for *mak*ing myprog.o. This rule is dependent on myprog.f which does exist. Hence, make would use that second rule to generate the file myprog.o, which uses the -c option of the *Fortran* f77 compiler. Once that had been accomplished, the target file, myprog, would be created using the first rule.

```
        ⋮
% make myprog
f77 -01 -c myprog.f
f77 -o myprog myprog.o
%
```

Notice that each rule is printed out as it is executed by make. After the successful completion of this invocation of make, the contents of the current working directory would include myprog.f, myprog.o, myprog, and makefile.

Observe the order of the two rules in this *makefile*. The dependency file of the first rule is the target of the second rule. On a few *UNIX* systems, make appears to do only one pass through the *makefile*. Thus, if the order of the rules above were reversed, make would again look for the dependency file, myprog.o. But when the file wasn't found, make would not know how to create it, since the rule for doing so was higher up in the *makefile* and could not be found either.

5 Further Examples

Now consider a more complex program consisting of several modules: myprog.f, sub1.f, sub2.f, and mytime.f. Here the convenience of make becomes more apparent. The *makefile* for the directory containing these modules might contain the following information:

```
# Simple Makefile for Several Modules

myprog:  myprog.f sub1.f sub2.f mytime.f
        f77 -01 -o myprog myprog.f sub1.f sub2.f mytime.f
```

Notice that all four *Fortran* modules are components of the target, myprog, and are listed as dependency files. This is perfectly legal; the executable file indeed depends upon all of them as shown in Fig. 3. However, this *makefile* requires that all four *Fortran* files be recompiled whenever myprog is to be *made*, since the *.f version of each module is listed as part of the f77 command line.

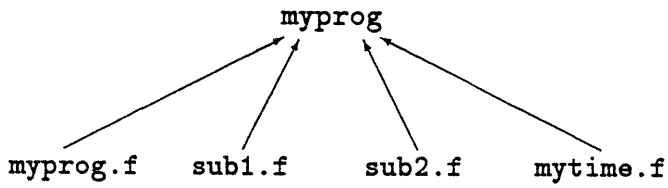


Figure 3: Derivation of Multi-Modular myprog.

A version taking better advantage of **make** would separate the compile steps and have the myprog target dependent on the object versions or *.o files for the given modules.

```
# Better Makefile for Several Modules
myprog:  myprog.o sub1.o sub2.o mytime.o
        f77 -o myprog myprog.o sub1.o sub2.o mytime.o
myprog.o:  myprog.f
        f77 -O1 -c myprog.f
sub1.o:  sub1.f
        f77 -O1 -c sub1.f
sub2.o:  sub2.f
        f77 -O1 -c sub2.f
mytime.o:  mytime.f
        f77 -O1 -c mytime.f
```

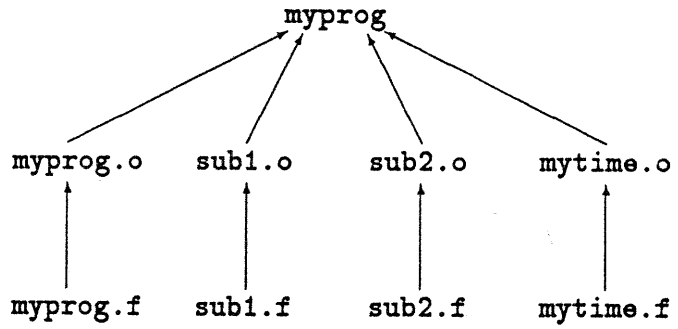


Figure 4: Alternate Derivation of Multi-Modular myprog.

In this way, not all the *Fortran* files would be recompiled to *make* myprog. Instead only those *Fortran* modules which had been modified since the last *make* would be recompiled. The derivation for this description file is shown in Fig. 4.

Suppose you had these four *Fortran* modules in your current directory along with the *makefile*.

```
        ⋮
% ls -l
total 4
-rw-rw-r-- 1 schauble    293 Jul 5 09:16 makefile
-rw-r--r-- 1 schauble     19 Jul 2 11:41 myprog.f
-rw-rw-r-- 1 schauble     31 Jul 5 09:17 mytime.f
-rw-rw-r-- 1 schauble     29 Jul 5 09:17 sub1.f
-rw-rw-r-- 1 schauble     29 Jul 5 09:17 sub2.f
%
```

The first time you use make to compile these modules together you might get the following response.

```
        ⋮
% make myprog
f77 -O1 -c myprog.f
f77 -O1 -c sub1.f
f77 -O1 -c sub2.f
f77 -O1 -c mytime.f
f77 -o myprog myprog.o sub1.o sub2.o mytime.o
%
```

Again, each rule is printed by make as it is executed. After the command has completed, the directory will contain the *.o files as well as the final target, myprog.

```
        ⋮
% ls -l
total 94
-rw-rw-r-- 1 schauble    293 Jul 5 09:16 makefile
-rwxrwxr-x 1 schauble  81188 Jul 5 09:20 myprog*
-rw-r--r-- 1 schauble     19 Jul 2 11:41 myprog.f
-rw-rw-r-- 1 schauble    648 Jul 5 09:19 myprog.o
-rw-rw-r-- 1 schauble     31 Jul 5 09:17 mytime.f
-rw-rw-r-- 1 schauble    456 Jul 5 09:20 mytime.o
```

```
-rw-rw-r-- 1 schauble     29 Jul 5 09:17 sub1.f
-rw-rw-r-- 1 schauble    452 Jul 5 09:19 sub1.o
-rw-rw-r-- 1 schauble     29 Jul 5 09:17 sub2.f
-rw-rw-r-- 1 schauble    452 Jul 5 09:19 sub2.o
%
```

If changes were now made to one module, say `sub2.f`, then the command

make myprog

would merely require that `sub2.f` be recompiled into `sub2.o`. Then all the `*.o` files, including those unchanged since the last *make*, would be linked together to form the new `myprog`.

```
        ⋮
% make myprog
f77 -O1 -c sub2.f
f77 -o myprog myprog.o sub1.o sub2.o mytime.o
% ls -l
total 94
-rw-rw-r-- 1 schauble    293 Jul 5 09:16 makefile
-rwxrwxr-x 1 schauble  81188 Jul 5 09:20 myprog*
-rw-r--r-- 1 schauble     19 Jul 2 11:41 myprog.f
-rw-rw-r-- 1 schauble    648 Jul 5 09:19 myprog.o
-rw-rw-r-- 1 schauble     31 Jul 5 09:17 mytime.f
-rw-rw-r-- 1 schauble    456 Jul 5 09:20 mytime.o
-rw-rw-r-- 1 schauble     29 Jul 5 09:17 sub1.f
-rw-rw-r-- 1 schauble    452 Jul 5 09:19 sub1.o
-rw-rw-r-- 1 schauble     30 Jul 5 09:20 sub2.f
-rw-rw-r-- 1 schauble    452 Jul 5 09:20 sub2.o
%
```

This is because **make** checks the creation dates of the `*.o` files against their corresponding `*.f` files (the dependency files), only recreating the out-of-date files (in this case, `sub2.o`).

In this last example, it becomes more obvious how **make** can save work. It knows exactly which modules need to be recompiled, and the simple command

make myprog

performs only those compilations necessary. Observe that entering a single command (only two words in length) is all that is needed to do the job, while typing in the corresponding f77 commands would be considerably longer (and more prone to typing errors). Of course, the *makefile* is not short, but it can be used over and over again.

The *makefile* can also be used as a record of the compilation. By looking at the rules and macro settings in the *makefile*, you will know how the executable files in the directory were compiled. For this reason, it is best to use different target names for executable files compiled under different flags. For instance, **debug** can be used for an executable file compiled under the **-g** option to be used by the **dbx** utility, as shown in Section 9.2.

6 Dynamic Macros

The main problem with this particular *makefile* is that it is long and repetitious. It seems that there must be a simpler way to accomplish this. There is, as demonstrated below.

6.1 $@ and $<

Consider the following version of our *makefile*:

```
# Shorter Makefile for Several Modules using Macros
myprog:  myprog.o sub1.o sub2.o mytime.o
        f77 -o $@ myprog.o sub1.o sub2.o mytime.o
.f.o :
        f77 -O1 -c $<
```

There are a few things to notice.

1. First of all, the rule for *mak*ing **myprog** now contains the characters, '**$@**', instead of the filename, **myprog**. For the **make** utility, these characters, '**$@**', form a macro symbol which is equivalent to the name of the target. This provides a shorthand method to repeat the target name. In fact, it would be correct to modify the rule above to read as follows:

```
myprog:  $$@.o sub1.o sub2.o mytime.o
        f77 -o $@ $@.o sub1.o sub2.o mytime.o
```

Comment: All **make** macros begin with a dollar sign (**$**). When a dynamic macro, such as '**$@**', is used in the dependency list, it must be preceded by an extra dollar sign.

2. The biggest difference in the new and shorter *makefile* above is that all the `.f`-to-`.o` rules are combined into a single rule. This rule will produce a `.o` file from any `.f` file which exists in the working directory. This is what the `.f.o` target means; namely, if a `.o` file is desired, create it from the corresponding `.f` file (the file with the same rootname) according to the given rule. This is called a **make** *implicit rule format* and will override the default rule.

3. The rule itself contains the symbol, '**$<**', where normally the *Fortran* filename would be. This **make** *macro*, '**$<**', should only be used in implicit rules. It refers to the file which caused the current rule to be invoked. In other words, if **make** is looking for the file **mytime.o** and finds instead the file **mytime.f**, '**$<**' is interpreted as **mytime.f**, the file which invokes the rule to *make* **mytime.o**.

Suppose we have modified the **sub1.f** and **mytime.f** files. Then typing the command

make myprog

with this new *makefile* will show the following response:

```
        ⋮
% make myprog
f77 -O1 -c sub1.f
f77 -O1 -c mytime.f
f77 -o myprog myprog.o sub1.o sub2.o mytime.o
%
```

If our *makefile* had not redefined the implicit rule, the command above would invoke the default compilation rule instead. Then the *makefile* would contain just one rule, much like the following:

```
# Makefile for Several Modules using Implicit Compilation

myprog:  myprog.o sub1.o sub2.o mytime.o
        f77 -o $@ myprog.o sub1.o sub2.o mytime.o
```

And the previous dialogue with **make** would change to show that the default `-O` optimization flag was used in compiling the individual modules.

```
        ⋮
% make myprog
f77 -O -c sub1.f
f77 -O -c mytime.f
f77 -o myprog myprog.o sub1.o sub2.o mytime.o
%
```

Replacing the default implicit rule allows us to define things to be done the way we want.

6.2 $? and $*

Two other **make** special dynamic macros are the characters, '**$?**' and '**$***'. The first, '**$?**', refers to a list of filenames which **make** considers to have been created earlier than the target. The other, '**$***', is only used in implicit rules; it refers to the common rootname of the target and dependent filenames. All the macros are set to the appropriate values when **make** first begins to work on a given rule and target.

To illustrate this, consider this modified version of our *makefile*:

```
# Makefile for Several Modules echoing Macros

OBJECTS = myprog.o sub1.o sub2.o mytime.o
CFLAGS = -O1

myprog:  $(OBJECTS)
        f77 $(CFLAGS) -o $@ $(OBJECTS)
        echo "The target file is " $@
        echo "The earlier files are " $?  *

.f.o :
        f77 $(CFLAGS) -c $<
        echo "The invoking file is " $<
        echo "The common rootname is " $*
        echo " "
```

Executing this version of the *makefile* will cause the values of all four dynamic macros, including '**$?**' and '**$***', to be written to the screen:

```
        ⋮
% make -s myprog
The invoking file is myprog.f
The common rootname is myprog
```

```
The invoking file is sub1.f
The common rootname is sub1

The invoking file is sub2.f
The common rootname is sub2

The invoking file is mytime.f
The common rootname is mytime

The target file is myprog
The earlier files are myprog.o sub1.o sub2.o mytime.o
%
```

Observe that the `-s` option flag has been used in this command. Normally, `make` echos each command of a *makefile* rule as it executes it; this flag turns the echo off. Otherwise, many more lines would have been printed out, even the '`echo`' commands. For more information on this and other options, see Section 8.

7 User-Defined Macros

It is often convenient for the programmer to create his own macros in a *makefile*.

7.1 A *Makefile* with User-Defined Macros

Consider the following modifications to our *makefile*.

```
# Makefile for Several Modules using Macros

OBJECTS = myprog.o sub1.o sub2.o mytime.o
CFLAGS = -O1 -c

myprog:  $(OBJECTS)
        f77 -o $@ $(OBJECTS)

.f.o :
        f77 $(CFLAGS) $<
```

This performs the same as the earlier *makefile* using the `-O1` optimization. However, two user-defined macros, `$(OBJECTS)` and `$(CFLAGS)`, have been added.

1. The first of these is just a list of all the object files (`*.o` files) upon which the target is dependent. This macro provides a handle for that list of files, which is used twice in this *makefile*. We need only type the list once in the definition of

the new macro, so we may prevent typing errors. This also makes it easier to add another object file to the list.

2. The second macro is `$(CFLAGS)`. This provides the compile flags we wish to use. If we wish to change optimization levels at some time, we could change the definition of `$(CFLAGS)` in the *makefile*

   ```
   CFLAGS = -O2 -c
   ```

   or we could call **make** with the specific parameter:

   make myprog "CFLAGS=-O2 -c"

Using the second method gives more flexibility. The dialogue with **make** would be as follows, provided all the old `*.o` files have been deleted first:

```
        ⋮
% make myprog "CFLAGS=-O2 -c"
f77 -O2 -c myprog.f
f77 -O2 -c sub1.f
f77 -O2 -c sub2.f
f77 -O2 -c mytime.f
f77 -o myprog myprog.o sub1.o sub2.o mytime.o
%
```

So, by using different parameters for the macro `CFLAGS` with the same *makefile*, we can compile the program under different optimization levels. The value of any user-defined macro in a *makefile* may be changed in this manner for a specific run.

The main problem with this method is that you have lost the record of how the target file was created. If you look at the directory after some weeks, you probably will not remember whether or not `myprog` was *made* with the flags given in the *makefile* or with some parameter change. It might be wise to rename the executable file in this case, `myprog2`, to indicate that it is not the standard version of the target file. Or better yet, include a **make** rule for `myprog2` which uses the `-O2` flag.

7.2 Another *Makefile* using User-Defined Macros

As another example, we can consider using a different version of the *Fortran* compiler. First, we will need to create a macro to define the compiler and add it to the rules, replacing all the occurrences of `f77`:

```
        ⋮
FCOMPLR = f77
        ⋮
myprog:  $(OBJECTS)
        $(FCOMPLR) -o $@ $(OBJECTS)
.f.o :
        $(FCOMPLR) $(CFLAGS) $<
```

Now if we remove the old `*.o` files and remake myprog with a parameter specifying a different compiler, the dialogue will look like this:

```
        ⋮
% make myprog "FCOMPLR=f772.1"
f772.1 -01 -c myprog.f
f772.1 -01 -c sub1.f
f772.1 -01 -c sub2.f
f772.1 -01 -c mytime.f
f772.1 -o myprog myprog.o sub1.o sub2.o mytime.o
%
```

Again, it might be best to use different target names for executable files that are compiled under different compilers. In this way, the *makefile* would provide a record of how each executable file was compiled.

7.3 Using User-Defined Macros

There are a few points to be observed concerning the macros.

1. Macro names are strings consisting of letters and digits and are similar to *C* identifiers. They can be of almost any length, subject to the system default. By convention, the letters used in macro names are usually capitalized.

2. User-defined macros should be given a default definition at the beginning of the *makefile*. This is to assure that they are defined before they are used.

3. Macros can be redefined by calling **make** with parameters. More than one parameter can be included in the **make** command. For instance, the command

   make myprog "FCOMPLR=f772.1" "CFLAGS=-O2 -c"

   is perfectly legal and will redefine both the `FCOMPLR` and `CFLAGS` macros.

4. When used with a **make** rule, the macro names must be contained within parentheses and preceded by a dollar sign, .e.g., `$(CFLAGS)`. The exception to this rule is that macro names of a single letter need not be put inside parentheses, e.g., `$A`. So, any single character or parenthesized string which is preceded by a dollar sign is assumed to be a **make** macro.

8 Additional Features

We have described above only a few of the features of **make** in order to get you started. Many more features are available. There are several option flags that can be used with the **make** command. And there are some special (fake) target names and command prefixes which add flexibility. More of of these features are discussed below; check the **man** pages to get a full listing.

8.1 Silent Running

The **make** utility usually prints out each command (or rule) before it executes it. This allows the user to see how the process is proceeding. However, it is possible that you might not wish to have all this output.

There are three ways to reduce this output.

1. The `-s` option flag requests a *silent* interaction with **make**. The commands (or rules) will *not* be printed out as they are executed. The dialogue with **make** simply becomes as follows:

```
        ⋮
% make -s myprog
%
```

   Only the command line for **make**, which you typed in, appears on the screen. An example of using this option is shown in Section 6.2.

2. If you just want certain rules not printed by **make**, insert the character, '@', at the beginning of the line. Rules which begin with an '@' will not be printed. For instance, consider the following *makefile* rule for a target named **help** which describes the functions of the given *makefile*:

```
help :
        @echo "This makefile supports the following:"
        @echo "make run - runs the program"
        @echo "make myprog - creates executable program"
```

Typing the command to *make* help will give the following response:

```
        ⋮
% make help
This makefile supports the following:
make run - runs the program
make myprog - creates executable program
%
```

If the command lines had not begun with the character '@', the response would have been as follows:

```
        ⋮
% make help
echo "This makefile supports the following:"
This makefile supports the following:
echo "make run - runs the program"
make run - runs the program
echo "make myprog - creates executable program"
make myprog - creates executable program
%
```

With each echo command printed as it is executed, the output becomes confused and is not as helpful as the target name would suggest.

3. If you always want to run in a silent mode, you should add the special (or fake) target name, .SILENT, to your *makefile*. For example, consider the following *makefile*:

```
# Makefile with silent responses
OBJECTS = myprog.o sub1.o sub2.o mytime.o
CFLAGS = -O1
.SILENT:
myprog:  $(OBJECTS)
        f77 $(CFLAGS) -o $@ $(OBJECTS)
.f.o :
        f77 $(CFLAGS) -c $<
```

Notice that `.SILENT` is given as a target with no dependencies and no rules. If we use this *makefile* to produce the executable file, **myprog**, the dialogue will not show the actions of **make** while they are being performed. The contents of the directory are shown before and after the **make** command; otherwise, you might wonder if anything was indeed done.

```
        ⋮
% ls
makefile      mytime.f      sub1.f
myprog.f      specprog.f    sub2.f
%
% make myprog
%
% ls
makefile      myprog.f      mytime.f
myprog*       myprog.o      mytime.o
specprog.f    sub1.o        sub2.o
sub1.f        sub2.f
%
```

8.2 Error Handling

If any rule results in an error while **make** is executing, the error is printed and the **make** command stopped. Sometimes, this is not appropriate for the given *makefile*. Consider, for instance, a rule which might remove some files from the working directory to clean up after running some process. If one of more of the files to be removed does not exist, the rule will return an error and **make** will stop.

There are three ways to avoid this.

1. The first is the use of the special character, '-', at the beginning of a rule. If an error is encountered while executing such a rule, it is ignored and the **make** continues.

2. The second method is to include the fake target, `.IGNORE`, as part of the *makefile*. This is similar to using the `.SILENT` fake target in Section 8.2 and instructs **make** to ignore *ALL* the errors found during its execution.

3. The final method is to call **make** with the `-i` option flag, which tells **make** to *ignore* all error codes for this particular invocation.

8.3 -n Option Flag

Suppose you want to **make myprog**, but first you want to know what files will be made (and which are already up-to-date) as well as the commands that will be executed. Of course, you could study the *makefile* and look at the creation dates of all the files you believe will be used. But an easier way would be to use the **-n** option flag for **make**, as follows:

```
        ⋮
% make -n myprog
f77 -O1 -c myprog.f
f77 -O1 -c sub2.f
f77 -o myprog myprog.o sub1.o sub2.o mytime.o
%
```

This appears to act just like the **make** command without the **-n** option flag. However, the **n** stands for *no execute mode*; so the commands that normally would be executed are merely listed without any of them being done. The example above shows that the object files, **myprog.o** and **sub2.o**, must be missing from the directory or out-of-date and so need to be created before the whole program can be linked together. In a like fashion, we can see that the object files, **sub1.o** and **mytime.o**, must be present and up-to-date.

8.4 -t Option Flag

The **-t** option flag requests that certain files be *touched*. When a file is *touched*, its creation date is changed to the current date and time. This may be used to avoid re*mak*ing files dependent upon files which have have been changed in a way which should not affect the dependent files.

For example, suppose we make a minor change to the comments in the file, **sub2.f**. This will cause the current version of the object file, **sub2.o**, out-of-date. Consequently, the executable file for the whole program, **myprog**, will also be out-of date. Touching those two files will update their creation dates. Meanwhile, the **make** utility knows that the other object files are still current.

```
        ⋮
% ls -l
total 89
-rw-rw-r-- 1 schauble   192 Jul 5  17:03 makefile
```

```
-rwxrwxr-x 1 schauble 81188 Jul 31 16:38 myprog*
-rw-r--r-- 1 schauble    19 Jul 2  11:41 myprog.f
-rw-rw-r-- 1 schauble   648 Jul 31 16:38 myprog.o
-rw-rw-r-- 1 schauble    31 Jul 5  09:17 mytime.f
-rw-rw-r-- 1 schauble   456 Jul 31 15:40 mytime.o
-rw-rw-r-- 1 schauble    29 Jul 5  09:17 sub1.f
-rw-rw-r-- 1 schauble   452 Jul 31 15:40 sub1.o
-rw-rw-r-- 1 schauble    30 Jul 31 16:46 sub2.f
-rw-rw-r-- 1 schauble   452 Jul 31 16:38 sub2.o
% make -t mytime.o
'mytime.o' is up to date.
% make -t sub1.o
'sub1.o' is up to date.
% make -t myprog
touch(sub2.o)
touch(myprog)
% ls -l
total 89
-rw-rw-r-- 1 schauble   192 Jul 5  17:03 makefile
-rwxrwxr-x 1 schauble 81188 Jul 31 16:47 myprog*
-rw-r--r-- 1 schauble    19 Jul 2  11:41 myprog.f
-rw-rw-r-- 1 schauble   648 Jul 31 16:38 myprog.o
-rw-rw-r-- 1 schauble    31 Jul 5  09:17 mytime.f
-rw-rw-r-- 1 schauble   456 Jul 31 15:40 mytime.o
-rw-rw-r-- 1 schauble    29 Jul 5  09:17 sub1.f
-rw-rw-r-- 1 schauble   452 Jul 31 15:40 sub1.o
-rw-rw-r-- 1 schauble    30 Jul 31 16:46 sub2.f
-rw-rw-r-- 1 schauble   452 Jul 31 16:47 sub2.o
%
```

Notice that touching the final executable file, **myprog**, causes the out-of-date object file, **sub2.o**, to be touched as well. This is because **sub2.o** is one of the files on which **myprog** is dependent.

8.5 -d Option Flag

The **-d** option flag permits d*ebug*ging of the *makefile* by generating more detailed information on each command (or rule) that is executed. For instance, typing the command

```
% make -d myprog
```

results in a great deal of output, only some of which is included here.

```
        ⋮
% make -d myprog
setvar:  = noreset = 0 envflg = 0 Mflags = 040101
Reading "=" type args on command line.
Reading internal rules.
setvar:  MACHINE = mips noreset = 0 envflg = 0
     Mflags = 040101
        ⋮
Reading makefile
setvar:  OBJECTS = myprog.o sub1.o sub2.o mytime.o
     noreset = 0 envflg = 0 Mflags = 040001
setvar:  CFLAGS = -O1 -c noreset = 0 envflg = 0
     Mflags = 040001
Warning:  CFLAGS changed after being used
doname(myprog,0)
TIME(myprog)=681000472
        ⋮
TIME(mytime.o)=680996426
look for implicit rules.  0
'myprog' is up to date.
        ⋮
OBJECTS = myprog.o sub1.o sub2.o mytime.o
        ⋮
MACHINE = mips
$ = $
MAKEFLAGS = bd
mytime.f done=2
sub2.f done=2
sub1.f done=2
myprog.f done=2
mytime.o done=2
sub2.o done=2
sub1.o done=2
myprog.o done=2
done=2 (MAIN NAME)
depends on:myprog:  myprog.o sub1.o sub2.o mytime.o
commands:
$(FCOMPLR) -o $@ $(OBJECTS)
```

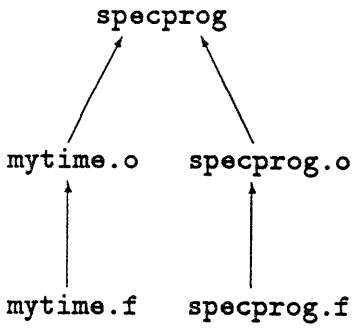


Figure 5: Derivation of specprog.

```
markfile done=0
        ⋮
%
```

As you can see, this option provides a lot of information. It lists all the environment settings used by make, the steps to determine the date of each target, and the rules, implicit or explicit, to *make* each target.

8.6 -f Option Flag

The -f option flag for the make utility allows a *file* named something other than makefile or Makefile to act as a *makefile.*

Suppose we have an additional program in our current directory, specprog.f. And assume this program also uses one of the subroutines in the directory named mytime.f. The derivation for specprog is shown in Fig. 5. We can create an additional *makefile* which will *make* the executable file, specprog, as follows:

```
# A Special Makefile
OBJECTS = specprog.o mytime.o
CFLAGS = -O1 -c
FCOMPLR = f77
specprog:  $(OBJECTS)
        $(FCOMPLR) -o $@ $(OBJECTS)
.f.o :
        $(FCOMPLR) $(CFLAGS) $<
```

We can name this new *makefile* any legal filename; in this case, we will call it `special`. To *make* the executable file, `specprog`, type the following:

make -f special specprog

and the dialogue with `make` will appear as below:

```
        ⋮
% make -f special specprog
f77 -O1 -c specprog.f
f77 -o specprog specprog.o mytime.o
%
```

The `-f special` portion of the command tells `make` to use the file named `special` to find the rules for the target `specprog`. Thus it is possible to keep more than one *makefile* in your directory. Notice that the object file `mytime.o` had been compiled previously and was up-to-date, probably from being used with the other program, `myprog`, and so did not need to be recreated.

Of course, we could have included all the rules for both targets in a single *makefile*,

```
# Combined Makefile

OBJECTS = myprog.o sub1.o sub2.o mytime.o
SOBJECTS = specprog.o mytime.o
CFLAGS = -O1 -c
FCOMPLR = f77

myprog:  $(OBJECTS)
        $(FCOMPLR) -o $@ $(OBJECTS)

specprog:  $(SOBJECTS)
        $(FCOMPLR) -o $@ $(SOBJECTS)

.f.o :
        $(FCOMPLR) $(CFLAGS) $<
```

simply by renaming the `OBJECTS` macro used by one of the target rules. The combined derivation graph for this more complex *makefile* is shown in Fig. 6. Observe that the module, `mytime.o`, is used by both main targets.

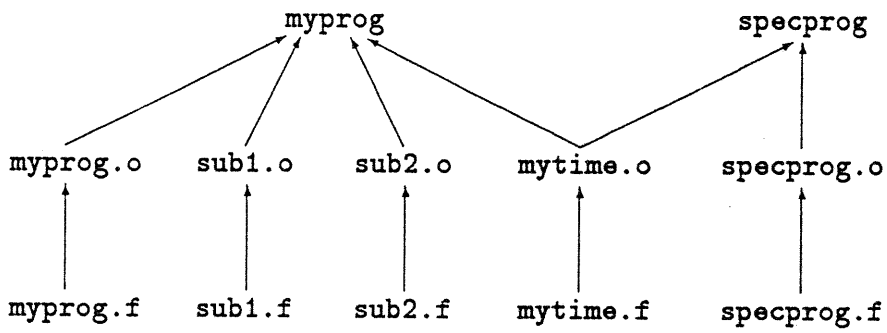


Figure 6: Alternate Derivation of specprog.

9 Other Examples

The examples we have used above all relate to compiling modular *Fortran* programs. But there are a number of other things can be done with make as well. It is not restricted to program compilation.

9.1 Program Execution

It is easy to include a run target in your *makefile*. This will execute a program, compiling and linking modules, if necessary.

```
run:  myprog
        myprog > myprog.output
```

In this case, typing the command

make run

will execute the program, myprog, causing the output to be redirected to the file named myprog.output. Observe two points about this *makefile* rule:

1. The target, run, is not a file; that is, the execution of this rule does not create a file named run. Since the target run is never created, the command

   make run

will *always* execute. This is legal and at times desirable.

It would have been quite logical to have the target be `myprog.output`, the name of the output file for the program, instead of `run`. However, this output file, `myprog.output`, would need to be removed or renamed before each execution.

2. The target is dependent on the executable file, `myprog`. If `myprog` is unavailable or out-of-date with respect to `myprog.f` or `myprog.o`, `make` will perform whatever steps are necessary to create a current version of `myprog` before executing it.

9.2 Compiling for *dbx*

If you are having trouble getting your program to run, you may wish to recompile it with the `-g` option flag; this will provide an executable file which can be used with the `dbx` debugging utility. To do so, you should include the following statements in your *makefile*:

```
        ⋮
SOURCES = myprog.f sub1.f sub2.f mytime.f
        ⋮
debug:  $(SOURCES)
        $(FCOMPLR) -g -o debug $(SOURCES)
```

Here we have added a macro named `SOURCES` which lists all the source files involved with the program, `myprog.f`. We cannot use the macro `OBJECTS` because that contains the `*.o` files which were compiled without the `-g` option. This new rule recompiles all the modules together with the `-g` option and creates an executable file named `debug`, (so named to avoid confusion with the regular executable file, `myprog`). Since the `-g` option turns off all optimizations, there is no need to include `$(CFLAGS)` in this rule.

9.3 Printing Files

The *makefile* can include rules to print out the source files as well as the output.

```
prints:  $(SOURCES)
         enscript -2rG $(SOURCES)
printo:  myprog.output
         lpr myprog.output
```

By referring to the macro SOURCES, all the modules related to myprog.f will be printed. Again, no target file is created when either of these rules is executed.

Notice that the enscript command in rotated, two-column format to reduce paper usage, since the source listings are usually used for debugging and then thrown away. Replacing 'enscript -2rG' by 'lpr' would produce the source listings in the more usual format.

9.4 Cleaning up

The *makefile* might include a rule to clean up the directory, deleting any core or *.o files.

```
clean:
        - rm core debug *.o
```

This version also removes the executable file, **debug**, which was created for use with dbx and is probably no longer needed. You might also wish to remove the file myprog, as that can be generated whenever needed by the *makefile.*

Notice that the rule begins with the special character, '-'. If no core file happened to exist in the directory when the command **make clean** was done, the rule would stop when it couldn't find a file named core. But because of the special *ignore errors* character, '-', at the front of the rule, **make** will continue onto the next file after printing out an error message.

```
        ⋮
% make clean
rm core
rm:  core nonexistent
** Error code 1 (ignored)
rm debug
rm:  debug nonexistent
** Error code 1 (ignored)
rm *.o
%
```

Here you can see that the files listed on the **rm** statement are handled individually. And the error messages for non-existent files are printed but ignored. Also, the commands for a rule can include wild card characters, as in *.o.

9.5 A Complete Makefile

A full *makefile*, containing all the rules in this section, is given in Appendix A. You may wish to use it as a model. Eventually, you will develop your own style. For instance, you might use `myprog.x` as the target name of the executable file in place of the filename `myprog`. Also, `myprog.out` might have replaced the target `run`.

10 A Makefile for *C*

Designing a *makefile* for program maintenance in other languages should be straightforward, based on the previous example. For instance, with *C* programs, you may want to add a list of the header (`*.h`) files to the dependencies. An example of such a *makefile* is given in Appendix B.

There are a few things to notice in this example:

1. The macro, `LIBS`, is used to include the `math.h` library in the compile-and-link command. This library is commonly used by *C* programs. Other library names can be added or substituted here.

2. The command

   make help

   will result in a listing of all the functions of this *makefile*. This is a very useful and recommended procedure, as one often forgets what one has done some months ago.

   This is the very first target in the make file. So if you should just type

   make

   the list will still appear. In other words, the `make` command with no target will attempt to *make* the *first* target in the makefile.

3. The `echo` commands under the target `help` are preceeded by the character '`@`'. This is to prevent the echo command from being printed out in addition to the string it is printing.

4. References to header `*.h` files have been added.

5. Targets entitled by macro names will substitute any of the files defined by that macro as the target file. For instance,

make mainprog

will activate the `$(EXECFIL)` rule.

6. Target need have no commands in the rule. The target

   `$(FILES) : $(HEADERS)`

   merely declares that `$(HEADERS)` are dependency files for `$(FILES)`. This rule has no other purpose.

11 Creating Your Own Makefile

Try creating a simple *makefile* in one of your directories. As you begin to use **make**, you will find more and more things that you will want to do with it. As a result, your *makefile* will begin to grow.

You may need a separate *makefile* in each directory, but this means that each one can be individualized to meet the requirements of its own directory.

Some points to remember when creating your own *makefiles*:

1. Remember that the rules in a *makefile* are interpreted as Bourne shell commands. Among other things, this means that it is best to use absolute pathnames when referring to other directories; the value of the tilde character, '~', is not recognized by the Bourne shell.

2. The environment variables, such as `$HOME` and `$HOST`, are known by **make** as it executes and are treated as macros of the same names. In the same way, *make* assumes a macro named `$SHELL` which is equivalent to `/bin/sh` by default. On some of the newer *UNIX* operating systems, it is possible to change this, by redefining the macro variable:

   ```
   SHELL = /bin/csh
   ```

12 Further Information

For more information on **make**, you may refer to the Feldman paper [Feldman 86] or the **man** pages. This utility is discussed in some *C* and *UNIX* textbooks, e.g., [Kay & Kummerfeld 88] and [Sobell 84]. And books, such as *Managing Projects with Make* [Talbott 90], provide a more thorough discussion of the use of **make** with examples in other areas.

References

[Feldman 86] FELDMAN, S. I. [Nov 1986]. Make – a program for maintaining computer programs. In GROUP, COMPUTER SYSTEMS RESEARCH, editor, *Unix Programmer's Manual Supplementary Documents 1*, pages PS1:12–1 to PS1:12–9. USENIX Association.

[Kay & Kummerfeld 88] KAY, JUDY AND BOB KUMMERFELD. [1988]. *C Programming in a UNIX Environment.* Addison-Wesley Publishing Company.

[Sobell 84] SOBELL, MARK G. [1984]. *A Practical Guide to the UNIX System.* Benjamin/Cummings Publishing Company, Inc.

[Talbott 90] TALBOTT, STEVE. [1990]. *Managing Projects with Make.* Nutshell Handbooks. O'Reilly & Associates, Inc., Sebastopol, CA.

Appendix A: A Makefile for *Fortran* Modules

```
# Final Complete Makefile

#          to initialize user-defined macros
SOURCES = myprog.f sub1.f sub2.f mytime.f
OBJECTS = myprog.o sub1.o sub2.o mytime.o
CFLAGS = -O1 -c
FCOMPLR = f77

#          to execute myprog
run:  myprog
        myprog > myprog.output

#          to link and compile myprog
myprog:  $(OBJECTS)
        $(FCOMPLR) -o $@ $(OBJECTS)

#          to compile fortran modules
.f.o :
        $(FCOMPLR) $(CFLAGS) $<

#          to compile myprog for dbx
debug:  $(SOURCES)
        $(FCOMPLR) -g -o debug $(SOURCES)

#          to print myprog source files
prints:  $(SOURCES)
        enscript -2rG $(SOURCES)

#          to print myprog output file
printo:  myprog.output
        lpr myprog.output

#          to remove unnecessary files
clean:
        - rm core debug *.o
```

Appendix B: A Makefile for *C* Modules

```
#
# Makefile for C modules
#
FILES = mainprog.c submod.c
HEADERS = mainprog.h submod.h
OBJECTS = mainprog.o submod.o
EXECFIL = mainprog
LIBS = -lm
FLAGS =

help :
        @echo "This makefile supports the following:"
        @echo "make run - runs the program"
        @echo "make mainprog - creates executable program"
        @echo "make lint - run lint on the program"
        @echo "make debug - provides dbx executable"
        @echo "make clean - deletes *.o, debug, core files"

run :  $(EXECFIL)
        $(EXECFIL)

$(EXECFIL): $(OBJECTS)
        cc $(FLAGS) $(OBJECTS) -o $(EXECFIL) $(LIBS)

.c.o :  $$@.c $$@.h
        cc $(FLAGS) -c $<

lint :  $(FILES)
        lint $(FILES)

debug :  $(FILES)
        cc -g $(FLAGS) -o debug $(FILES)

$(FILES): $(HEADERS)

clean :
        - rm *.o core debug
```