The make Programming Discipline

make is a utility that guides the building of complex things out of one or more simpler things. The “complex thing” can be practically any sort of file that you create regularly, such as a report or a program.

Under COHERENT, **make** is most commonly used to control the building of complex C programs; and it is in this context that **make** shows its power most easily. This tutorial introduces the features of **make**, and discusses how to use it to help you build complex C programs easily and efficiently.

How Does make Work?

To understand how **make** works, it is first necessary to understand how a C program is built: how COHERENT takes you from the C source code that you write to the executable program that you can run on your computer.

The file of C source code that you write is called a *source module*. When COHERENT compiles a source module, it uses the C code in the source module, plus the code in the header files that the code calls to produce an *object module*. This object module is *not* executable by itself. To create an *executable file*, the object module generated from your source module must be handed to a linker, which links the code in the object module with the appropriate library routines that the object module calls, and adds the appropriate C runtime startup routine.

For example, consider the following C program, called **hello.c**:

```
main()
{
	printf("Hello, world\n");
}
```

When the C compiler compiles the file that contains C code shown above, it generates an object module called **hello.o**. This object module is not executable because it does not contain the code to execute the function **printf()**; that code is contained in a library. To create an executable program, you must hand **hello.o** to the linker **ld**, which copies the code for **printf** from a library and into your program, adds the appropriate C runtime startup routine, and writes the executable file called **hello**. This third file, **hello**, is what you can execute on your computer.

The term *dependency* describes the relationship of executable file to object module to source module. The executable program *depends* on the object module and one or more libraries. The object module, in turn, depends on the source module and its header files (if any).

A program like **hello** has a simple set of dependencies: the executable file is built from one object module, which in turn is compiled from one source module. If you changed the source module **hello.c**, creating an updated version of **hello** would be easy: you would simply compile **hello.c** to create **hello.o**, which you would link with the library and the runtime startup to create **hello**. COHERENT, in fact, does this for you automatically: all you need to do is type

```
cc hello.c
```

and the C compiler takes care of everything.

On the other hand, the dependencies of a large program can be very complex. For example, the executable file for the MicroEMACS screen editor is built from several dozen object modules, each of which is compiled from a source module plus one or more header files. Updating a program as large as MicroEMACS, even when you change only one source module, can be quite difficult. To rebuild its executable file by hand, you must remember the names of all of the source modules used, compile them, and link them into the executable file. Needless to say, it is very inefficient to recompile several dozen object modules to create an executable when you have changed only one of them.

make automatically rebuilds large programs for you. You prepare a file, called a **makefile**, that describes your program’s chain of dependencies. **make** then reads your **makefile**, checks to see which source modules have been updated, recompiles only the ones that have been changed, and then relinks all of the object modules to create a new executable file. **make** both saves you time, because it recompiles only the source modules that have changed, and spares you the drudgery of rebuilding your large program by hand.

Try make

To see how **make** works, try compiling a program called **factor**. It is built from the following files:

```
atod.c
factor.c
makefile
```

All three are kept in directory **/usr/src/sample**. To use them, copy the following files into your current directory. (By the way, first make sure that you do not already have a file named **makefile** in your current directory, or the following commands will overwrite it.)

```
cp /usr/src/sample/atod.c .
cp /usr/src/sample/factor.c .
cp /usr/src/sample/makefile .
```

Now, type **make**. **make** begins by reading **makefile**, which describes all of **factor**'s dependencies. It then uses the **makefile** description to create **factor**. The following appear on your screen:

```
cc -c factor.c
cc -c atod.c
cc -f -o factor factor.o atod.o -lm
```

Each of these messages describes an action that **make** has performed. The first shows that **make** is compiling **factor.c**, the second shows that it is compiling **atod.c**, and the third shows that it is linking the compiled object modules **atod.o** and **factor.o** to create the executable file **factor**.

When **make** has finished, the shell prompt returns. To see how your newly compiled program works, type

```
factor 100
```

factor calculates the prime factors of its argument **100**, and print them on the screen:

```
2 2 5 5
```

To see what happens if you try to re-make your file, type **make** again. **make** will run quietly for a moment, and then exit. **make** checked the dates and times of the object modules and their corresponding source modules and saw that the object modules had a time later than that of the source modules. Because no source module changed, there was no need to recompile an object module or relink the executable file, so **make** quietly exited.

To see what happens when one of the source modules changes, try the following. Use the MicroEMACS screen editor to open the file **factor.c** for editing. Insert the following line into the comments at the top, immediately following the **/***:

```
* This comment is for test purposes only.
```

Now type **<ctrl-Z>** to save the file and exit. Type **make** once again. This time, you will see the following on your screen:

```
cc -O -c factor.c
cc -o factor factor.o atod.o -f -lm
```

Because you altered the source module **factor.c**, its time was later than that of its corresponding object module, **factor.o**. When **make** compared the times of **factor.c** and **factor.o**, it noted that **factor.c** had been altered. It then recompiled **factor.c** and relinked **factor.o** and **atod.o** to re-create the executable file **factor**. **make** did not touch the source module **atod.c** because **atod.c** had not been changed since the last time it was compiled.

As you can see, **make** simplifies the construction of a C program that uses more than one source module.

Essential make

Although **make** is a powerful program, its basic features are easy to master. This section will show you how to construct elementary **makefile**s.

The makefile

When you invoke **make**, it searches the directories named in the environmental variable **PATH** for a file called **makefile** or **Makefile**. (You can tell **make** to read a file other than **makefile** or **Makefile**; see the description of **make**'s **-f** option, below.) As noted earlier, the **makefile** is a text file that describes a program's dependencies. It also describes the type of program you wish to build, and the commands for building it.

A **makefile** has three basic parts.

First, the **makefile** describes the executable file's dependencies. That is, it lists the object modules needed to create the executable file. The name of the executable file is always followed by a colon ':' and then by the names of files from which the target file is generated.

For example, if the program **feud** is built from the object modules **hatfield.o** and **mccoy.o**, you would type:

```
feud:    hatfield.o mccoy.o
```

If the files **hatfield.o** and **mccoy.o** do not exist, **make** knows to create them from the source modules **hatfield.c** and **mccoy.c**.

Second, the **makefile** holds one or more *command* lines. The command line gives the command to compile the program in question. The only difference between a **makefile** command line and an ordinary **cc** command is that a **makefile** command line *must* begin with a space or a tab character.

For example, the **makefile** to generate the program **feud** must contain the following command line:

```
        cc -o feud hatfield.o mccoy.o
```

For a detailed description of the **cc** command and its options, refer to the entry for **cc** in the Lexicon.

Third, the **makefile** lists all of the header files that your program uses. (If you don't know what a header file is, see the entry for **#include** in the Lexicon.) These files are given so that **make** can check if any had been modified since your program was last compiled. For example, if the program **hatfield.c** used the header file **shotgun.h** and **mccoy.c** used the header files **rifle.h** and **pistol.h**, the **makefile** to generate **feud** would include the following lines:

```
hatfield.o: shotgun.h
mccoy.o: rifle.h pistol.h
```

Thus, the entire **makefile** to generate the program **feud** is as follows:

```
feud: hatfield.o mccoy.o
        cc -o feud hatfield.o mccoy.o

hatfield.o: shotgun.h
mccoy.o: rifle.h pistol.h
```

A **makefile** can also contain *macro definitions* and *comments*. These are described below.

Building a Simple makefile

The program **factor** is built from two source modules, **factor.c** and **atod.c**. No header files are used. The **makefile** contains the following two lines:

```
factor: factor.o atod.o
        cc -o factor factor.o atod.o -f -lm
```

The first line describes the dependency for the executable file **factor** by naming the two object modules needed to build it. The second line gives the command needed to build **factor**. The option **-lm** at the end of the command line tells **cc** that this program needs the mathematics library **libm** when the program is linked. No header file dependencies are described because these programs use no special header files.

Comments and Macros

make ignores all lines that begin with a pound sign '#'. This lets you embed comments within a **makefile**, to "document" the file so that whoever reads it will know what it is for. For example, you may wish to include the following comments in your **makefile** for **factor**:

```
# This makefile generates the program "factor".
# "factor" consists of the source modules "factor.c" and
# "atod.c".  It uses the standard mathematics library
# "libm", but it requires no special header files.
# "-f" lets you use printf for floating-point numbers.

        factor: factor.o atod.o
                cc -f -o factor factor.o atod.o -lm
```

Anyone who reads this file will know immediately what it is for by looking at the comments.

make also lets you define macros within your **makefile**. A *macro* is a symbol that represents a string of text. Usually, a macro is defined at the beginning of the **makefile** using a *macro definition statement*. This statement uses the following syntax:

```
        SYMBOL = string of text
```

Thereafter, when you use the symbol in your **makefile**, it must begin with a dollar sign '$' and be enclosed within parentheses. (If the macro name is only one character long, the parentheses are not required.) A macro name can use both upper-case and lower-case characters.

Macros eliminate the chore of retyping long strings of file names. For example, with the **makefile** for the program **factor**, you may wish to use a macro to substitute for the names of the object modules out of which it is built. This is done as follows:

```
# This makefile generates the program "factor".
# "factor" consists of the source modules "factor.c" and
# "atod.c".  It uses the standard mathematics library
# "libm", but it requires no special header files.
# "-f" lets you use printf for floating-point numbers.


OBJ = factor.o atod.o
factor: $(OBJ)
        cc -o factor $(OBJ) -f -lm
```

The macro **OBJ** is used in this **makefile**. If you use a macro that has not been defined, **make** substitutes an empty string for it. The use of a macro makes sense when generating large files out of a dozen or more source modules. You avoid retyping the source module names, and potential errors are avoided.

Note that you can define macros in the **makefile**, in the environment, or as a command-line argument. A macro defined as a command-line argument always overrides a definition of the same macro in the environment or in the **makefile**. Normally, a definition in a **makefile** overrides a definition of the same macro name in the environment; however, the **-e** option to **make** forces definitions in the environment to override those in the **makefile**.

Setting the Time

As noted above, **make** checks to see which source modules have been modified before it regenerates your C program. This is done to avoid wasteful recompiling of source modules that have not been updated.

make determines that a source module has been altered by comparing its date against that of the target program. For example, if the object module **factor.o** was generated on March 16, 1992, 10:52:47 A.M., and the source module **factor.c** was modified on March 20, 1992, at 11:19:06 A.M., **make** will know that **factor.c** needs to be recompiled because it is *younger* than **factor.o**.

Building a Large Program

As shown earlier, **make** can ease the task of generating a large program. The following gives a **makefile** that can be used to generate the screen editor MicroEMACS:

```
# makefile for "MicroEMACS"

CFLAGS = -O
LFLAGS = /usr/lib/libterm.a
OBJ=ansi.o basic.o buffer.o display.o file.o \
        fileio.o line.o main.o random.o region.o \
        search.o spawn.o termio.o vt52.o window.o \
        word.o tcap.o

me: $(OBJ)
        cc -o me $(OBJ) $(LFLAGS)

$(OBJ): ed.h
```

Note that this **makefile** has been simplified for the purposes of this tutorial; the actual **makefile** that builds the COHERENT edition of MicroEMACS is considerably more complex.

The first line in the above **makefile** gives commentary that describes the file does. The next five lines define macros that are used on the target and command lines. The first macro will be discussed in the following section. The second macro substitutes for the name of a special library that is needed to create this program. The third macro, which is three lines long, stands for the names of the source modules that produce MicroEMACS. A backslash '\' must be used to tell **make** that the macro's definition extends onto the next line.

The next line names the target file (**me**) and the files used to construct it, here represented by the macro **OBJ**.

Next comes the command line, which gives the compilation to be performed. This line *must* begin with a space or a tab.

The last line lists the header file **ed.h**, which is required by all of the files used to generate MicroEMACS.

Command-Line Options

Although **make** is controlled by your **makefile**, you can also control **make** by using command-line options. These allow you to alter **make**'s activity without editing your **makefile**.

Options must follow the command name on the command line and begin with a hyphen, '-', using the following format. The square brackets merely indicate that you can select any of these options; do *not* type the brackets when you use the **make** command:

```
make [ -deinpqrst ] [ -f filename ]
```

Each option is described below.

-d Debug option: **make** describes all of its decisions. You can use this to debug your **makefile**.

-e Environment option: force definitions in the environment to override those in the **makefile**. For example, if the **makefile** defines

```
foo=makefoo
```

and the environment defines

```
foo=envfoo
```

then **$(foo)** expands to **makefoo** if you use the command **make** but expands to **envfoo** if you use the command **make -e**.

-f *filename*
File option: Tell **make** that its commands are in a file other than **makefile**. For example, the command

```
make -f smith
```

tells **make** to use the file **smith** rather than **makefile**. If you do not use this option, **make** searches the directories named in the environmental variable **PATH**, and then the current directory for a file entitled **makefile** or **Makefile** to execute.

-i Ignore errors: **make** ignores error returns from commands and continues processing. Normally, **make** exits if a command returns an error status.

-n No execution: **make** tests dependencies and modification times but does not execute commands. This option is especially helpful when constructing or debugging a **makefile**.

-p Print: **make** prints all macro definitions and target descriptions.

-q Quit option: Return a zero exit status if the targets are up to date. Do not execute any commands.

-r Rules option: **make** does not use the default macros and commands from **/usr/lib/makemacros** and **/usr/lib/makeactions**. These files will be described below.

-s Silence: **make** does not print each command line as it is executed.

-t Touch: **make** changes the modification time of each executable file and object module to the current time. This suppresses recreation of the executable file, and recompilation of the object modules. Although this option is used typically after a purely cosmetic change to a source module or after adding a definition to a header file, it must be used with great caution.

Other Command Line Features

In addition to the options listed above, you may include other information on your command line.

First, you can define macros on the command line. A macro definition must *follow* all command-line options. Arguments, including spaces, must be surrounded by quotation marks, as spaces are significant to the shell. For example, the command line

```
make -n -f smith "OPT=-DTEST"
```

tells **make** to run in the *no execution* mode, read the file **smith** instead of **makefile**, and define the macro **OPT** to mean **-DTEST**.

The ability to define macros on the command line means that you can create a **makefile** using macros that are not yet defined; this greatly increases **make**'s flexibility and makes it even more helpful in creating and debugging large programs. In the above example, you can define a command line as follows:

```
cc $(OPT) example.c
```

When you define the macro **OPT** on the command line, then the program is compiled using the **-DTEST** option, which defines the preprocessor variable **TEST**.

As noted above, a macro defined on the command line always overrides an identically named macro defined either in the environment or in the **makefile**.

Another command-line feature lets you change the name of the *target file* on the command line. Normally, the target file is the executable file that you wish to create, although, as will be seen, it does not have to be. As will be discussed below, a **makefile** can name more than one target file. **make** normally assumes that the target is the first target file named in **makefile**. However, the command line may name one or more target files at the end of the line, after any options and any macro definitions.

To see how this works, recall the program **factor** described above. **factor** is generated out of the source modules **factor.c** and **atod.c**. The command

```
make atod.o
```

with the **makefile** outlined above would produce the following **cc** command line:

```
cc -c atod.c
```

if the object module **atod.o** did not exist or were outdated. Here, **make** compiles **atod.c** to create the target specified in the **make** command line, that is, **atod.o**, but it does not create **factor**. This feature allows you to apply your **makefile** to only a portion of your program.

The use of special, or *alternative*, target files is discussed below.

Advanced make

This section describes some of **make**'s advanced features. For most of your work, you will not need these features; however, if you create an extremely complex program, you will find them most helpful.

Default Rules

The operation of **make** is governed by a set of *default rules*. These rules were designed to simplify the compilation of a typical program; however, unusual tasks may require that you bypass or alter the default rules.

To begin, **make** uses information from the files **/usr/lib/makemacros** and **/usr/lib/makeactions** to define default macros and compilation commands. **make** uses the commands in **makemacros** and **makeactions** whenever the **makefile** specifies no explicit regeneration commands. The command line option **-r** tells **make** not to use the macros and actions defined in **makemacros** and **makeactions**.

As shown in earlier examples, **make** knows by default to generate the object module **atod.o** from the source module **atod.c** with the command

```
cc -c atod.c
```

The macro **.SUFFIXES** defines the suffixes **make** knows about by default. Its definition in **makemacros** includes both the **.o** and **.c** suffixes.

make's files **makemacros** and **makeactions** use pre-defined macros to increase their scope and flexibility. These are as follows:

$< This stands for the name of the file or files that cause the action of a default rule. For example, if you altered the file **atod.c** and then invoked **make** to rebuild the executable file **factor**, **$<** would then stand for **atod.c**.

$*** This stands for the name of the target of a default rule with its suffix removed. If it had been used in the above example, **$* would have stood for **atod**.

$<** and **$* work *only* with default rules; these macros will not work in a **makefile**.

$? This stands for the names of the files that cause the action and that are younger than the target file.

$@ This stands for the target name.

You can use the macros **$?** and **$@** in a **makefile**. For example, the following rule updates the archive **libx.a** with the objects defined by macro **$(OBJ)** that are out of date:

```
libx.a: $(OBJ)
        ar rv libx.a $?
```

For more information on archives, see the Lexicon entry for the command **ar**.

makemacros also contains default commands that describe how to build additional kinds of files:

- **AS** and **ASFLAGS** call the assembler **as** to assemble **.o** files out of files with the suffix **.s**, which **make** assumes hold assembly language.
- **YACC** and **YFLAGS** call **yacc** to build **.o** or **.c** files from files with the suffix **.y**, which **make** assumes hold **yacc** source code.
- **LEX** and **LFLAGS** call **lex** to build **.o** or **.c** files from files with the suffix **.l**, which **make** assume hold **lex** source code.

You can change the default rules of **make** by changing them in **makeactions** and changing the definition of any of the macros as given in **makemacros**.

Source File Path

If a file is not specified with an absolute path name beginning with '/', **make** first looks for the file in the current directory. If the file is not found in the current directory, **make** searches for it in the list of directories specified by the macro **$(SRCPATH)**. This allows you to compile a program in an object directory separate from the source path.

For example

```
export SRCPATH=/usr/src/local/me
make
```

or alternatively

```
make SRCPATH=/usr/src/local/me
```

builds objects in the current directory as specified by the **makefile** from sources kept in directory **/usr/src/local/me**. To test changes to a program built from several source files, copy only the files you wish to change to the current directory; **make** will use the local sources and find the other sources on the **$(SRCPATH)**.

Note that **$(SRCPATH)** can be a single directory, as in the above example, or a list of directories. In the latter case, each entry in the list must be separated by a colon ':', as described in the Lexicon entry for the function **path()**.

Double-Colon Target Lines

An alternative form of target line simplifies the task of maintaining archives. This form uses the double colon "::" instead of a single colon ':' to separate the name of the target from those of the files on which it depends.

A target name can appear on only one single-colon target line, whereas it can appear on several double-colon target lines. The advantage of using the double-colon target lines is that **make** remakes the target by executing the commands (or its default commands) for the *first* such target line for which the target is older than a file on which it depends.

For example, for the program **factor** described earlier, assume that two versions of the source modules **factor.c** and **atod.c** exist: fBfactora.c plus **atoda.c**, and **factorb.c** plus **atodb.c** The **makefile** would appear as follows:

```
OBJ1 = factora.o atoda.o
OBJ2 = factorb.o atodb.o
```

```
factor:: $(OBJ1)
        cc -c $(OBJ1) -lm

factor:: $(OBJ2)
        cc -c $(OBJ2) -lm
```

This **makefile** tells **make** to do the following: (1) Check if either **factora.o** or **atoda.o** is younger than **factor**. (2) If either one is, regenerate **factor** using this version of these files. (3) If neither **factora.o** nor **atoda.o** is younger than **factor**, then check to see if either **factorb.o** or **atodb.o** is younger than **factor**. (4) If either of them is, then regenerate **factor** using the youngest version of these files.

This technique allows you to maintain multiple versions of source files in the same directory and selectively recompile the most recently updated version without having to edit your **makefile** or otherwise trick the system.

You cannot target a file in both a single-colon and a double-colon target line.

Special Targets

A few target names have special meanings to **make**. The name of each special target begins with '.' and contains upper-case letters.

The target name **.DEFAULT** defines the default commands **make** uses if it cannot find any other way to build a target. The special target **.IGNORE** in a **makefile** has the same effect as the **-i** command line option. Similarly, **.SILENT** has the same effect as the **-s** command line option.

Errors

make prints "*command* exited with status *n*" and exits if an executed *command* returns an error status. However, it ignores the error status and continues processing if the **makefile** command line begins with a hyphen '**-**' or if the **make** command line specifies the **-i** option.

make reports an error status and exits if the user interrupts it. It prints "**can't open** *file*" if it cannot find the specification *file*. It prints "**Target** *file* **is not defined**" or "**Don't know how to make** *target*" if it cannot find an appropriate *file* or commands to generate *target*. Other possible errors include syntax errors in the specification file, macro definition errors, and running out of space. The error messages **make** prints are generally self-explanatory. The section **Error Messages**, at the end of this manual, lists **make's** error messages and describes them briefly.

Exit Status

make normally returns a status of zero if it succeeds, and of one if an error occurs. With the **-q** option (described above), **make** returns zero if all files are up to date and two if they are not up to date.

Alternative Uses

make is a program that helps you construct complex things from a number of simpler things.

make usually is used to build complex C programs: the executable file is made from object modules, which are made from source modules and header files. However, you can also use **make** to build any file that is constructed from one or more source modules. For example, an accountant can use **make** to generate monthly reports from daily inventories: all the accountant has to do is prepare a **makefile** that describes the dependencies (that is, the name of the monthly report he wishes to create and the names of the daily inventories from which it is created), and the command required to generate the monthly report. Thereafter, to recreate the report, all the accountant has to do to generate a monthly report is type **make**.

In another example, the **makefile** can trigger program-maintenance commands. For example, the target name **backup** might define commands to copy source modules to another directory; typing **make backup** saves a copy of the source modules. Similar uses include removing temporary files, building archives, executing test suites, and printing listings. A **makefile** is a convenient place to keep all the commands used to maintain a program.

The following example shows a **makefile** that defines two special target files, **printall** and **printnew**, to be used with the source files for the program **factor**.

```
# This makefile generates the program "factor".
# "factor" consists of the source modules "factor.c" and
# "atod.c".  It uses the standard mathematics library
# libm, but it requires no special header files.

OBJ = factor.o atod.o
SRC = factor.c atod.c

factor: $(OBJ)
	cc -o factor $(OBJ) -lm

# program to print all the updated source modules
# used to generate the program "factor"

printall:
	pr $(SRC) | lpr
	>printnew

printnew: $(SRC)
	pr $? | lpr
	>printnew
```

In this instance, typing the command

```
make printall
```

forces **make** to generate the target **printall** rather than the target **factor**, which is the default as it appears first in the **makefile**. The **pr** and **lpr** commands are then used to print a listing of all files defined by **SRC**. The macro **OBJ** cannot be used with these commands because it would trigger the printing of the object files, which would not be of much use. It also creates an empty file **printnew**. This new file serves only to record the time the listing is printed. This tactic is performed in order to record the time that the listing was last generated so that **make** will know what files have been updated when you next use **printnew**.

Typing the command

```
make printnew
```

forces **make** to generate the target **printnew** rather than the default target **factor**. **printnew** prints only the files named in the macro **SRC** that have changed since any files were last printed.

Where To Go From Here

The Lexicon article on **make** summarizes **make**'s options and features. The source code included with the COHERENT system and with the COHware packages include **makefile**s. Studying them will show you how **make** has been used to control the building of large, real-world applications.