

Computer Shopper

The World's Biggest Computer Magazine

August 1988 Vol. 8 No. 8 Issue 103

Program Maintenance: Touch And Make For MSDOS

By Dean C. Wills

When I first purchased MicroSoft's C Compiler 5.0 I was pleasantly surprised to find a "Make" utility bundled along with the package. Make, a UNIX programmer's utility that determines which files need to be recompiled when source code is modified, was an old friend and a welcome addition to my software library. At the same time, however, I was disappointed to notice that Make's sister utility, "Touch," was not provided. A quick glance through the software ads showed that this practice was not uncommon; Make was provided in a variety of software packages that neglected to include Touch. With this in mind I set out to write a version of Touch for public use, as well as explain Make and Touch, and their use together. Let me begin with Touch. (See Listing #1 on page 304 for the assembly language code for "Touch.")

Understanding Touch

Touch is a programmer's utility that may be used either independently or as a complement to Make. Touch updates a file's revision time to the present without modifying its contents. This is certainly useful in itself, although it might take a moment to understand why. On the other hand, anyone who's ever done a directory on a released disk from MicroSoft will understand why immediately. Inspection shows that all the files share the same revision time and date. This is desirable because it allows easy identification of a file with a particular version of the software. Touch was probably used to do this. After all they couldn't really have all been revised at the same time. (Or could they? *Headline-MicroSoft C Version 5.0 created: 10/15/87 5:00 a.m.: "It was a dark and stormy night. The keyboards were blazing as the elves completed their task." And so on.*)

You invoke Touch by entering "Touch" followed by an MSDOS file specification describing the desired files. Touch expands the file specification into separate file names using the MSDOS "First-Match" and "Next-Match" commands. Taking the preceding paragraph's scenario as an example, update the revision times of all the files in a directory by entering

```
Touch *.*
```

Applications for Touch range from the mundane to the ingenious. A friend of mine who loved to play games on company time was once cautioned because he had saved several games in progress during the day—a fact reflected in each saved gamefile's revision time. From then on, after each bout of gaming, he set his system clock to either the lunch hour or before or after working hours and "Touched" all of the saved file to conceal the day's shirking.

Touch is also powerful when used in conjunction with Make. Here it is capable of overriding Make to either force or prevent a certain action. With this in mind, onward to Make.

Understanding Make

Make takes care of the details of generating an executable program from program source. Using a file's time of last revision along with production rules which describe a program, Make can determine which files need to be compiled or linked after a source modification. It then selectively compiles and links the program where necessary to create an up-to-date executable program.

The production rules for Make are read from a file aptly known as a makefile. It basically lists all files in the program and how they are compiled; its structure will be presented later. More important than that now is an understanding of the workings of Make on a simple program. Assume that a makefile is set up for a program with a few source files which are: 1) compiled individually to make object files and then 2) linked together to produce an executable program. Assume also that all the latest modifications to the source files have been compiled and linked in. This program is said to be up-to-date. A program remains up-to-date so long as the revision time of each object file is more recent than the revision time of its corresponding source file, and the revision time of the executable file is more recent than the revision time of any of the object files. This seems complicated but isn't really. A determination of whether something is up-to-date is something most programmers do without worrying about the revision times. It reduces to knowing that compiles follow source modifications and not the other way around. Or even on a simpler level: I modified source file X. Time to recompile and relink.

Regardless of how programmers think about it, Make can use the revision times of files to determine whether a program is up-to-date and do something about it if it isn't. After the description I presented earlier, the question arises why use Make at all if it is something that programmer do instinctively? There are several reasons. First of all, Make saves typing. Enter:

Make "makefile"

And the entire compile and link process is done automatically. The makefile also contains the list of compilation or link options being used to create a program and generates them automatically. My current options of choice for MicroSoft C 5.0 are:

```
cc /c /Zp /Zi /Od /AL /d /u /G2 /G14 file.c
```

This is hard enough to type let alone remember.

Next, Make takes care of compiling and linking even when the process isn't instinctive. It really becomes powerful when programs get complex with multiple source files sharing incongruent subsets of header files and then being compiled together into several executable files. And what if several programmers are all working on different files at the same time? Who knows what files need to be recompiled? Make does. No matter what the file is made up of, the revision times will reflect modifications and a makefile can be created to show what to compile and how to do it.

This description of Make limits it to its most common usage: that of compilation and link manager. Although it is actually capable of directing any sort of revision time based actions, it is beyond the scope of this article to delve beyond Make as the program maintainer. With that in mind, let me explain how to use Make.

To use Make, create a script called a "makefile" which lists all files in the program, how they are dependent upon each other, and what actions to take if certain files are out-of-date with respect to others. Invoke Make by entering:

```
Make make_file_name
```

Here "make_file_name" is the name of the makefile. Each instruction for Make in this file has the following format:

```
target_file:one_or_more_dependent_files action_to_take_if_any_dependent_is_more_recent_than_target
```

Typical makefile script for a C program (line numbers added)

```
1: program.obj: program.c program.h header.h
2: cc <compile_options> program.c
3:
4: utility.obj: utility.c utility.h header.h
5: cc <compile_options> utility.c
6:
7: program.exe: program.obj utility.obj
8: link <link_options> program utility;
```

In this sample, the program is divided into two source files, program.c and utility.c. These are compiled separately then linked together to create the executable program.exe. They have individual header files, program.h and utility.h, and share a common header file, header.h. The first line of the script is interpreted as program.obj, the target, depends on program.c, program.h and header.h. If any of these last three have a revision field more recent than program.obj, signal the operation in the following line which is:

```
cc <compile_options> program.c
```

Make repeats this action for each instruction in the makefile.

For simple program maintenance, Make alone suffices. The makefile is created and designed once, and from then on, all program maintenance is completed through an invocation of Make. But there are a few situations where a compile is or isn't necessary and Make concludes the opposite. This is where Touch comes in handy. I will present a few of those situations here.

1) Using Touch to Force a Compile:

Changing compilation and link options. As mentioned earlier the makefile describes how the particular compiler or linker is invoked and which command-line options need to be used. But a problem arises when the command-line options need to be changed. Take for example the case of program.c earlier. Suppose I wanted to prepare this module for special debugging using CodeView with MicroSoft C 5.0. This requires that the option "/Zi" be added to the command line. The proper way to go about this of course is to change the "/Zi" to the line that indicates how program.c is compiled (line 2) and invoke Make to recompile the program. However, Make is smarter than we would prefer at this point. Make will examine the revision time of program.c and notice that it hasn't changed since it was last compiled. This is correct as only the makefile was changed. Nevertheless, a recompile with the new options is necessary.

To recompile program.c we must either invoke the compiler directly or modify program.c so that Make will compile it automatically. The disadvantages of the former option already explained earlier, it is best to choose the latter and use Make. To do this we first modify the makefile as described. Then we need to update the revision time so Make will detect that a recompilation is necessary.

We can either edit program.c using a standard word processor, which is slow because it will read and write the entire file in addition to updating the revision time, or we can update the revision times of one or more files quickly and easily using Touch. Enter:

touch program.c

This command will update the revision time of program.c to the present. To Make, this looks like a source modification. Now when Make is invoked it will think the source has been modified and will generate a compilation using the new options.

2) Using Touch to Avoid a Compile:

Adding non-program code to source files. Someday computers will be so swift that compile time will become insignificant. Until then, avoiding an unnecessary compile is wonderful when possible. Normally, changing the source file indicates that a compile is necessary. There are modifications to source code however which really don't require a recompilation of the program-adding comments for example. A recompilation is unnecessary because the compiler will still generate the same object file regardless of comments or other non computer-language text. Caveat-source modifications confuse symbolic debuggers like CodeView if new lines were entered.

Touch may be used to trick Make into skipping the compile although it's a little more complicated. First, execute Make on the program as usual so it is up-to-date. Then make any non computer-language modifications to the file, program.c for example.

Then enter:

```
touch program.obj  
touch program.exe
```

This command will update the revision time of program.obj and then program.exe to the present. To Make, this looks like the program has just been compiled and then linked. When Make is invoked it will decide that no compilation or link is necessary. As for the Make in advance, that is necessary to include any previous modifications; this procedure assumes that the program was up-to-date and thus the only modifications that were made were the ones unimportant to the object file.

These two examples show both sides of the picture with respect to Touch and Make. More sophisticated techniques can be developed combining these methods.

The source for my version of Touch is provided as an addendum, and its explanation left as an exercise. At this point I must remark that I used Make in Touch's development and have been using them both in tandem ever since. Hopefully this article will lead the way to more efficient software development and recruit more Make and Touch converts. If there is a strong enough following, I'm sure we'll find Touch as well as Make provided in the standard OS/2 version of MicroSoft C.

Source For Touch-TOUCH.ASM –available by request.

To create TOUCH.EXE execute:

```
masm touch;  
link touch;
```