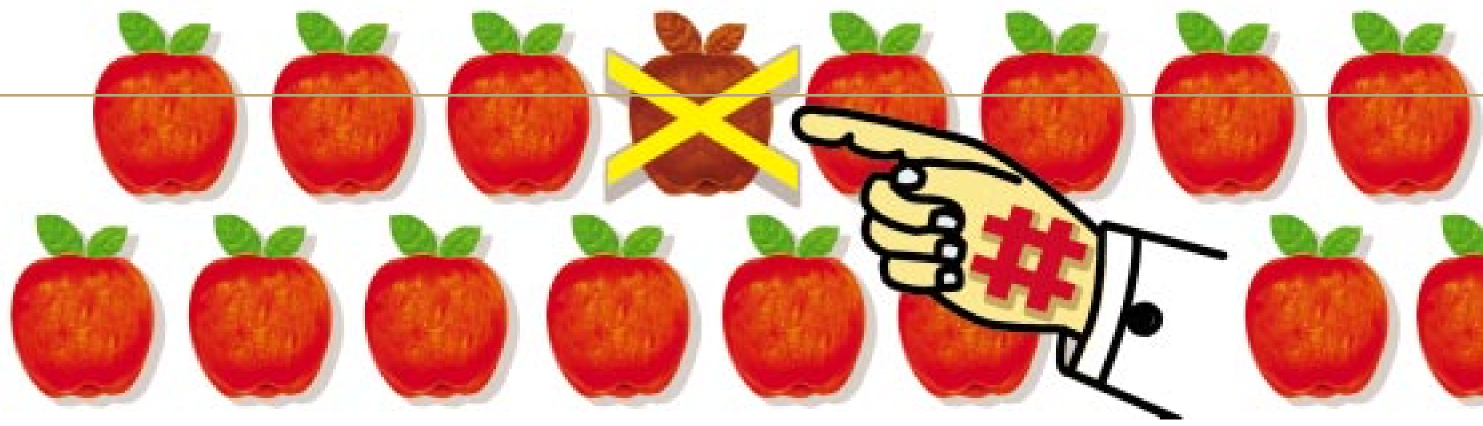
RETURN

NIGEL JONES



In Praise of the #error Directive

The #error directive may not be a widely used preprocessor directive, but according to the author, it can make your life easier. Here's his case.

One of the least used but potentially most useful preprocessor directives is `#error`. I would guess that only a small percentage of the readers of this article know what `#error` is, and an even smaller percentage have actually used it. This is a shame, because `#error` has a couple of uses that are invaluable for making your life a lot easier.

So what exactly is the `#error` directive and what does it do? It's an ANSI-C specified preprocessor directive. Its syntax is very straightforward, as you can see:

```
#error <writer supplied error mes-
sage>
```

The `<writer supplied error message>` is optional (although it never really makes sense to omit it) and can consist of any printable text. You don't even have to enclose the text in quotes.

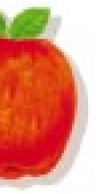
When the preprocessor encounters a `#error` statement, it causes compilation to terminate and the writer-supplied error message to be printed to `stderr`. A typical error message from a compiler looks like this:

```
Filename(line_number): Error!
Ennnn: writer supplied error message
```

where `Filename` is the source file name, `line_number` is the line number where the `#error` statement is located, and `Ennnn` is a compiler-specific error number. Thus, the error message is basically indistinguishable from ordinary compiler error messages.

"Wait a minute," you might say. "I spend enough time trying to get code to compile and now he wants me to do something that causes more compiler errors?" Absolutely! The essential point is that code that compiles but is incorrect is *worse* than useless. I've found three general areas in which this

"Wait a minute," you might say. "I spend enough time trying to get code to compile and now he wants me to do something that causes more compiler errors?" Absolutely!



problem can arise and `#error` can help. Read on and see if you agree with me.

Protecting a work in progress

I tend to code using a step-wise refinement approach, so it isn't unusual during development for me to have functions that do nothing, `for` loops that lack a body, and so forth. Consequently, I often have files that are compilable but lack some essential functionality. Working this way is fine, until I'm pulled off to work on something else (an occupational hazard of being in the consulting business). Because these distractions can occasionally run into weeks, I sometimes return to the job with my memory a little hazy about what I haven't completed. In the worst-case scenario (which has occurred), I perform a `make`, which runs happily, and then I attempt to use the code. The program, of course, crashes and burns, and I'm left wondering where to start.

In the past, I'd comment the file to note what had been done and what was still needed. However, I found this approach to be rather weak because I then had to read all my comments (and I comment heavily) in order to find what I was looking for. Now I simply enter something like the following in an appropriate place in the file:

```
#error *** Nigel - Function foo
is incomplete. Fix before using
***
```

Thus, if I forget that I haven't done the necessary work, an inadvertent attempt to use the file will result in just about the most meaningful compiler message I'll ever receive (after all, I wrote it). Furthermore, it saves me from having to wade through pages of comments, trying to find what work I haven't finished.

Protecting compiler-dependent code

As much as I strive to write portable code, I often find myself having to trade off performance for portability—and in the embedded world, performance tends to win. However, what happens if a few years later I reuse some code without remembering that the code has compiler-specific peculiarities? The result is a much longer debug session than is necessary. But a judicious `#error` statement can prevent a lot of grief. A couple of examples may help.

Example 1

Some floating-point code requires at least 12 digits of resolution to return the correct results. Accordingly, the various variables are defined as type `long double`. But ANSI only requires that a `long double` have 10 digits of resolution. Thus on certain machines, a `long double` may be inadequate to do the job. To protect against this, I would include the following:

```
#include <float.h>
#if (LDBL_DIG < 12)
#error  *** long doubles must
 have at least 12 digit resolu-
 tion. Do not use this compiler!
 ***
#endif
```

This approach works by examining the value of an ANSI-mandated constant found in float.h. Incidentally, this is one of the few uses I've found for the various ANSI-mandated constants.

Example 2

An amazing amount of code makes invalid assumptions about the underlying size of the various integer types. If you have code that has to use an `int` (as opposed to a user-specified data type such as `INT16`), and the code assumes that an `int` is 16 bits, you can do the following:

```
#include <limits.h>
#if (INT_MAX != 32767)
#error *** This file will only
 work with 16 bit integers. Do
 not use this compiler! ***
#endif
```

Again, this works by checking the value of an ANSI-mandated constant. This time the constant is found in the file limits.h.

This approach is a lot more useful than putting these limitations inside a big comment that someone may or may not read. After all, you have to read the compiler error messages.

Properly handling conditionally compiled code

Since conditionally compiled code seems to be a necessary evil in life, it's common to find code sequences such as the following:

```
#if defined OPT_1
/* Do option_1 */
#else
/* Do option_2 */
#endif
```

As it is written, this code means the following: if and only if `OPT_1` is defined, we will do `option_1`; otherwise we'll do `option_2`. The problem with this code is obvious. A user of the code doesn't know (without explicitly examining the code) that `OPT_1` is a valid compiler switch. Instead, the naïve user will simply compile the code without defining `OPT_1` and get the alternate implementation, irre-

BEN FISHMAN

I know that if this procedure had been adopted universally, I would have saved a lot of time over the years trying to reuse other people's code.

spective of whether that is what's required or not. A more considerate coder might be aware of this problem, and instead do the following:

```
#if defined OPT_1
   /* Do option 1 */
#elif defined OPT_2
   /* Do option 2*/
#endif
```

In this case, failure to define either OPT_1 or OPT_2 will typically result in an obscure compiler error at a point later in the code. The user of this code will then be stuck with trying to work out what must be done to get the module to compile. This is where #error comes in. Consider the following code sequence:

```
#if defined  OPT_1
/* Do option_1 */
#elif defined OPT_2
/* Do option_2 */
#else
#error *** You must define OPT_1
or OPT_2 to compile this file ***
#endif
```

Now the compilation fails, but at least it tells the user explicitly what to do to make the module compile. I know that if this procedure had been adopted universally, I would have saved a lot of time over the years trying to reuse other people's code.

So there you have it. Now tell me, don't you agree that #error is a really useful part of the preprocessor, worthy of your frequent use—and occasional praise? **esp**

Nigel Jones is a consultant on all aspects of embedded development. He particularly likes working on small, distributed systems. Reach him at NAJones@compuserve.com.