https://www.embedded.com/assert-yourself

Assert Yourself

May 1, 2001 Niall murphy

Niall continues to assert the usefulness of assertions, but this month, he shows you how to make your own.

Last month we discussed the use of the **assert()** macro. Now we're going to look at how you can define your own macro for handling assertions. Enough trade-offs are made in the design of this macro that a one-size-fits-all will most likely be an uncomfortable fit. To avoid confusion in this article and in your code, it is advisable to use the name **ASSERT** rather than the **assert** that is defined in the standard library.

There are two reasons to use macros for assertions, rather than a normal function call. The lesser reason is that we want to have the option of removing all assertions from the code if the cost in space and speed of having them is not affordable.

The more important reason is that we want the file name and line number of the assertion to be available. This information is accessible from the **__LINE__** and **__FILE__** macros. The preprocessor maintains these values as it processes each file. In fact, if you do not like the way it numbers the lines in the file, you can restart the line numbering with:

```
#line 2000
```

Sneak one of these into a colleague's code just before he starts compiling and watch the fun as all of the syntax errors are reported on the wrong line. Apart from this amusing distraction, the **#line** directive does not help us write assertions, but **__LINE__** does. We want to use the fact that the preprocessor knows what line it is on by passing that information to our function for handling assertion failures. The macro could look like this:

```
#define ASSERT(expr)
   if (!(expr))
    aFailed(__FILE__, __LINE__)
```

The signature of aFailed would then be:

```
void aFailed(char *file,
   int line);
```

We now have the job of somehow communicating the line and file name to the user, and then halting or resetting the system. Much of my work is in the graphics arena, so a display is usually available where the system can place its dying wishes:0

In some cases it is more important or convenient to keep the error information in some sort of persistent storage. This is where a string-which can be of arbitrary length-is inconvenient. For this reason I like to convert filenames to a numeric form.

https://www.embedded.com/assert-yourself

Getting rid of strings

This leads me to the other problem with strings. When the assert macro is expanded, the filename appears as a string. This may mean several bytes per assertion. Some compilers are clever enough to combine all occurrences of identical strings into one occurrence, and in some cases this optimization is enabled via a compiler command line switch. Even with this optimization, you still pay the price of one string per file, which penalizes projects that go to the trouble of breaking the program down into small manageable chunks.If a reasonable numbering system can be contrived for all files in the project, we can define a macro holding an index for the file:

#define F_NUM 1

A similar macro appears at the top of each C source file defining a number for that file. Obviously we want those numbers to be unique. I generally assume that no assertions appear in header files, though some C++ users may find a need for this.

I use the following macro to ensure that reuse of the same number for two different files will be detected:

```
#define FILENUM(num)
   enum { F_NUM=num };
   void _dummy##num(void) {}
```

The enumerated type declares the name **F_NUM** so that it can be used within the **ASSERT** macro, and we can't put a **#define** inside our macro. The **##** is a preprocessor operator that performs concatenation. The **_dummy** symbol is declared as the name **_dummy** concatenated with the num value passed in. If the macro is called twice with the same number, say 5, then **_dummy5()** will be declared twice, leading to a link time error. This will prevent you from running a system in which two files are using the same identifying number. Now the programmer does not use:

#define F_NUM 1

at the top of the source file. Instead the following line is used:

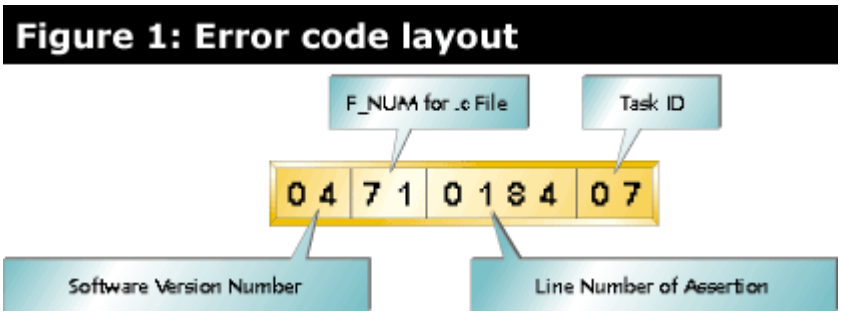
FILENUM(1)

Unique error codes

Now that we have replaced the string **__FILE__** with a number, **F_NUM** , we almost have a unique identifier for the error code. We want a unique identifier that can be reported from the field, easing the investigation of the problem.

While a number of pieces of information can be optionally added onto the error code, I always insist on getting the version number inserted. When the error is reported, it can be notoriously difficult to get people to report information that may require further action on the machine, such as discovering the version of software. It may also be possible that the machine is no longer operational, and therefore no way exists with which to find out this information, even if the user tries. The version number is vital, since the line number reported is only valid for that version.

https://www.embedded.com/assert-yourself



It is also useful to report the number of the task that was running when **aFailed()** was called. This leads to an error code with the layout shown in Figure 1. The user need not be aware of the meaning of each part of the number, but it can be deciphered once the report reaches the engineer who is troubleshooting the failure.

A number is easier to fill in on a report form and easier to store in NVRAM. It is also international, leading to fewer ambiguities if the same error is being reported in different spoken languages.

An objection may be raised at this point that the shipped product should never fire any assertions, and so this error code mechanism will be of no use. I will be glad to listen to such objections from any readers who have shipped products that never fail in the field.

Failure actions

Once the **aFailed()** function has stored or reported the error, it must decide on the system's next action. One option is to halt, forcing human intervention, such as a power cycle, before normal operation can restart. Another option is to trigger a software reset and attempt to run the system again. The policy you choose will be largely determined by the nature of your product.

During the development stage, though, policy may be very different. For example, some RTOSes have a debug task that allows the user to access the system via a serial port. If we enable that task and disable all others from the aFailed() function, the engineer will have access to RTOS-specific information, such as the number of ready tasks and the length of system queues. An alternative to an RTOS debug task is a machine code monitor that will allow a user to view values in memory from the serial port, or through the user interface.

If such a debug mode is not available, the **aFailed()** function can be made to print out the values of a set of important global variables to the serial port or report on the state of the system in another way.

If the system is running with a source level debugger, place a breakpoint in the **aFailed()** function. If you hit an assertion while running, code execution will halt, and you can examine stack traces and the like to track down the fault. I set up the initialization file of my debugger to set up this breakpoint automatically every time I start the debugger, so that I never run without it.

https://www.embedded.com/assert-yourself

Redundant information

Some compilers supply an **assert()** macro that passes the condition as a string. A **#** in the middle of a macro declaration causes the following argument to be converted to a string, by placing it in quotation marks:

```
#define ASSERT(expr)
   if (!(expr))
      aFailed(F_NUM, __LINE__, #expr)
```

This macro passes a third argument, which, in this case, is a string. If we have the following assertion on line 50 of foo.c:

```
assert(x>5);
```

it will expand to:

```
if (!(x>5))
   aFailed(1, 50, "x>5");
```

It may be handy to print the condition at debug time, but it will be meaningless in the field. It also adds more constant strings to the code, and these strings can't be optimized away like the file name because they will be different for every invocation of the macro. Since the condition can be seen in the source code, I generally leave out this argument.

Matching else

As defined so far, the macro has a problem. If a call to the macro is immediately followed by an **else** statement, the **else** will match the **if** in the macro, though the programmer almost certainly intended it to match the previous **if** . So we need to adapt our macro to:

```
#define ASSERT(expr)
   if (expr)
      {}
   else
      aFailed(F_NUM, __LINE__)
```

The semicolon after the assert where the macro is called will still match the semicolon required by the final line of the macro.

Some implementations use the properties of the || (or) operator to implement the macro as follows:

```
#define ASSERT(expr)
   ((expr) ||
      aFailed(F_NUM, __LINE__))
```

If **expr** evaluates to false, the second part of the expression will be evaluated, causing **aFailed()** to be called. Which route to choose is a matter of taste, so long as the dangling else issue is resolved.

https://www.embedded.com/assert-yourself

Ship with assertions?

Last month I discussed the controversial method of defining **NDEBUG** in order to make the expression in an assertion disappear at compile time. This topic deserves a little more attention. I will start by expressing the opinion of C.A.R. Hoare on this subject. He considered it to be like using a lifebelt during practice, but then not bothering with it for the real thing.

I recently read a news posting in which an engineer claimed that one of his projects was shipped without assertions because the delivered product crashed more often with the assertions in place. Whenever the product crashed, the failure was automatically escalated to red alert status and the number of red alerts reflected badly on the team. If we assume that these red alerts were raised because the organization really cared about their customer, then they should have cared enough to not allow the customer to run a system which might fail silently and process bad data until the next reboot.

Having said that, I know of one situation in which using assertions in the debug version and not using them in the shipped version is acceptable. That is if the resources are so tight that the alternative is to not use assertions at all. If you simply have to ship with so little ROM that you cannot add this extra code, then you should still use assertions to your benefit during the debugging part of the lifecycle.

Assuming we have decided to ship with assertions, the programmer will still be reluctant sometimes to do certain checks, if the price in CPU cycles is high. Consider a string that should never exceed 10 characters. The following check may be applied:

ASSERT(strlen(string) <=>

If this check is inside a tight loop, the **strlen()** call may add considerable cost.

More extreme cases exist. One option is to follow the lead of the EPOC C++ coding standard which defines **__ASSERT_ALWAYS** and **__ASSERT_DEBUG** . The former will be compiled into all versions of the product and the latter will be removed for the shipped version. I like this idea, but I would be inclined to pick shorter names for the macros, since they should appear very often and the argument to the macro may also be long. If the length of the macro name causes the assertion to be broken up over multiple lines, it will be less readable.

Assertions and malloc

It was pointed out to me that my use of an assertion to check the result of a **malloc()** may not be a wise course, because a failed **malloc()** is not an indication of a bug, but an indication that the computer has run out of one particular resource: heap space. Steve Maguire takes this line in *Solid Code* (Microsoft Press, 1993). I believe that the circumstances for an embedded system are different from those for a desktop application.

With a desktop application you have two choices when you run out of heap space. One is for the application to free up some of the resources that it allocated itself, and then retry the operation. The second is to prompt the user that the problem exists, and encourage him to exit another application so that the current application may gather more resources. The second option is

obviously not feasible in an embedded system. And the first option of freeing up some already allocated resource is only rarely a possibility. On the desktop, information that is not stored in RAM can be stored on disk. Placing more of the data on disk (possibly using virtual memory) requires more time, but the end result is the same. On an embedded system, a disk is not typically available, so freeing up RAM in another part of the application is likely to result in the failure of some feature.

Embedded engineers also have fewer excuses for running out of RAM. During design you know exactly how much RAM will be available. On the desktop, that number will vary with the platform and with the other applications running on the system.

All of this leads to the conclusion that a failed **malloc()** represents a software failure for most embedded systems. It means that the system has a leak. An allocation failure should never happen simply because the user made too many requests, as can happen on the desktop.

Final assertion

The most impressive property of using assertions, once used widely, is that the majority of bug reports from testing and from the field are reports that an assertion failed. The number of times bugs manifest themselves as strange behavior, or a hung system, drops dramatically. Once you know where in the code to start your investigations, most bugs are more transparent. This is particularly useful for bugs that are hard to reproduce.

Translation strings revisited

A number of readers responded to my March 2001 column, the title of which appeared in Kanji characters and translated to "Do You Speak Japanese?" (p. 43). In that column, I discussed representating characters from non-ASCII character sets. In that issue I proposed closing and reopening strings in order to ensure that octal escape sequences in strings did not accidentally merge with a following digit. For example we might intend "**415"** to be stored with the same two bytes as "**!5"** , since octal 41 is the exclamation mark in the ASCII table. However the C compiler will not interpret this as 41 followed by the character 5. Instead it will try to interpret the octal value 415. The solution I proposed was to break the string, by closing the double quotes and reopening them. The compiler will interpret "**41" "5"** as "**!5"** . I am using the explanation mark for the example, but in a real application we would be dealing with unprintable characters, which the programmer's editor could not enter.

It was pointed out to me independently by Daron Smith, Todd Litwin, and John Langenbach that by ensuring that there were three octal digits, I could get the desired result and it would not look as untidy as breaking the string in two. By using leading zeros I can increase the number of digits for the first character to three. So we could use "**415"** to get "**!5"** . This is a far better solution. The simplest rule is to always ensure that your octal character constants are always exactly three digits long.

It is worth pointing out the origin of my mistake. While an octal escape sequence is at most three digits long, a hexadecimal escape sequence will continue until a non-hexadecimal character is found. So if you are using hexadecimal values in your string you will have to close and reopen the

string. This is a rather devious difference between the way you use hexadecimal character constants and the way you use octal character constants, so don't let yourself get caught out like I did.

Niall Murphy has been writing software for user interfaces and medical systems for 10 years. He is the author of *Front Panel: Designing Software for Embedded User Interfaces.* Murphy's training and consulting business is based in Galway, Ireland.