https://www.embedded.com/assertiveness-training-for-programmers

Assertiveness Training for Programmers

March 15, 2001 Niall murphy

Concerned about the number of programmers who don't use or know about assert(), Niall explains the macro and how to use it.

The **assert()** macro is one of those simple tools that would not seem to merit an entire column, but recently I have come across an alarming number of engineers who have not heard of it or do not use it. Hopefully this month's column will help bolster the number who make good use of this feature. This month, we will look at appropriate use of assertions, and next month we will examine how we can write the **assert()** macro ourselves.

One of the properties of a good software project is that more lines of code are written for test and debug purposes than for implementing basic requirements. An engineer in the telecommunications industry told me that, in his estimation, 50% of the code in their switches was dedicated to checking that the other 50% was doing its job right. It is important to have a consistent policy for inserting self-checking code. Otherwise it contributes to the overall complexity of the source.

The importance of self-checking software became apparent to Neil Armstrong when he made his final descent to the moon in the Apollo 11 lander.[1] The spacecraft's computer was capable of reporting error codes to the pilot if a sanity check failed. Neil Armstrong read the error code 1202 from a control panel as he made his final approach. After the error code was reported, the computer reset and started doing its job again. Mission control deduced that this particular software problem would not lead to an aborted mission. Engineers at ground control knew that the error code was not critical unless the alert was continuous. Later investigation revealed that the CPU was running one extra task that was taking up processor time that should have been available for other tasks. The lander software was written in assembler and did not have the benefit of the C pre-processor, but the principle remains the same. Accurately pin-pointing a problem often saves the day.

If you are going to add self-checking code, it should have certain properties. You want to be able to turn checking on and off, since the final product may not have enough CPU cycles to do this sort of checking. You do not want to pollute the code with a multitude of if-then statements that are not part

of the basic algorithm you are trying to implement. You also want the self-checking statements to be easy to write. If it involves a lot of work, programmers will add fewer checks-that's just human nature.

The **assert()** macro is used to check expressions that ought to be true as long as the program is running correctly. It is a convenient way to insert sanity checks. If you write a piece of code that computes the day of the month, then the following check may be useful:

```
assert (dayOfMonth < 32);
```

If the preceding algorithm has generated an invalid value, it's likely that the condition in the assertion will not be true. Typical behavior when the assert fails is for the program to exit.

The assert macro is defined in the **assert.h** file, which is part of the standard library. The exact behavior of the macro varies with implementation, but in general it is as follows: if the expression passed to the macro is false, output an error message that includes the file name and line number, and then exit. Some Unix implementations save a core file. Other environments allow you to jump straight to a debugger, if one is installed. On embedded systems, you almost always have to define your implementation to suit the tools you are using, or the error response that fits your system. We need to consider how to write our own version of **assert()** , but first I will examine how it should be used, regardless of how it is defined.

Using assertions

A number of philosophies can be employed when deciding where to use an **assert()** macro. I prefer to assume that assertions only serve the purposes of catching bugs and helping documentation. If you remember those two goals, you will not go far wrong.

Helping to document the code means that the statements inside the assertion tell the reader something he might not already know. For example:

```
void set(int v)
{
   assert(v > 0);
   assert(v < 10);
```

It is now obvious to the reader that the valid range of the parameter **v** is 1 to 9. I am not suggesting that this replaces good comments, but a comment may become invalid when a programmer updates the code but forgets to update the comment. An assertion has the advantage of being validated at runtime.

At this point I will give you a quick peek at the topic of next month's discussion and show you how the **assert()** macro could be defined:

https://www.embedded.com/assertiveness-training-for-programmers

```
#define assert(expr)
   if (!(expr))
   assertFailed();
```

If the expression evaluates to 0, the function **assertFailed()** will be called, typically halting the program in some way. This version of the macro is very weak, but for the moment it is enough to understand that the assert is a macro, and not a function call.

Another factor to consider when choosing where to use an assertion is that they are meant to catch bugs. If any valid way exists for the statement to be false in the running of the program, you should not use an assertion. Issues such as user errors and expected failures of a communications channel should not be handled with assertions.

Even if you make the choice that all assertions will be left in the code when the product ships, the software should still function without them. For example:

```
assert (buffer = malloc(10));
```

This might seem like a reasonable way to check that an allocation succeeded, but it is an incorrect use of the assert macro. If all of the assertions in the program were removed, then the buffer would no longer be allocated and we would have changed the behavior of the program. A better way to check the same thing is to write:

```
buffer = malloc(10);
assert (buffer != NULL);
```

So, in short, an assert should never have a side effect. There are two good reasons for this.
The **assert()** macro is just that: a macro. The argument passed to a macro may appear more than once in the macro definition causing the expression argument to be evaluated more than once. In the case above, two calls to **malloc** would be made-one of which would be a memory leak.

The second reason that it is unhealthy to use an assertion with a side effect that is part of the main algorithm is that different projects will apply different policies to their use of assertions. On your current project, assertions may stay in the shipped version of the code, but you may move to another project where that is not the case. You may think that removing the assertions is a bad idea, but it may not be your decision. You want to have good programming habits that will work on both projects.

When I say that the assertions are removed, I do not actually mean that the code is edited to remove them, but the macro is defined in such a way that the expression in the assert is never evaluated, saving processor cycles and code space, but, of course, losing the benefit of the check. When using the standard **assert.h** , you can disable all asserts by defining **NDEBUG** . This will result in all assertions being defined as an empty expression.

https://www.embedded.com/assertiveness-training-for-programmers

Size matters

I often find myself passing structures from one task or processor to another via a buffer. Maybe the buffer is being passed on a queue, or across a serial link. The size of the buffer may be set at compile time or as part of the configuration of the RTOS. You want to be sure that the buffer is big enough to hold the structure. Assume that the size of the buffer is defined in **SIZE** . Now:

```
assert (sizeof(struct myData) >
   SIZE);
```

will check that the structure fits in the buffer. It is not enough to check this once by hand, since you may add a field to **struct myData** later and forget to recheck the size. With an assertion, it will get checked on every run.

It is better to put this sort of a check in an initialization routine because executing it in a piece of code that is run regularly is a waste of CPU cycles.

Ideally, the check shown here would be performed at compile time so it won't take up any space in the final executable. You would think that the following would do the trick:

```
#if (sizeof(struct myData) > SIZE)
#error myData is too big
```

Logically, you are trying to do the right thing. Unfortunately, the **sizeof()** operator is not going to work. The **sizeof()** operator is evaluated by the compiler, but the **#if** is evaluated by the pre-processor. So at the point in time when the value of **sizeof(struct myData)** is required, it has not yet been calculated.

If you are determined to use this check without spending any CPU cycles on the target processor, I know a trick that you can use. The basis of the trick is to get the compiler to do the comparison, rather than the pre-processor. If the expression **(sizeof(struct myData) > SIZE)** appears in the code, the compiler will evaluate it to 0 or 1. We want to ensure that the compiler stops compiling and gives an error message if the expression evaluates to 0. One place where 0 is illegal is as the size of an array. So the following declaration will not compile if the expression is false:

```
static char dummy[(sizeof(struct
   myData) > SIZE)];
```

The dummy array will either have a size of 1 or 0. If it's 1, we waste one byte of RAM by creating this array. If the size of the array is 0, any ANSI-compliant compiler will refuse to compile this line.

We could go a little further and make the array **const** . This would consume one byte of ROM rather than a byte of RAM. It is up to you to decide which is more valuable. Maybe you could use compiler extensions to specify the location of this array and place it in an area of memory that will never be

used. At some point, however, you have to ask yourself how much work you want to do to save a single byte.

I will caution you that this trick does not work with the GNU C compiler, which is not ANSI conformant, though you can get it to generate a warning for a zero-sized array if you use the -ansi and -pedantic switches.

While this trick is interesting, it is usually the case that if you can afford to use assertions at all, you can afford to use them for cases such as these, where the expression could have been done at compile time rather than run time.

Forced assert

In some cases you know a bug exists when you reach a certain point in the code. A common example is a switch statement where the programmer expects exactly one of the branches to execute. If the default of the switch is reached, the value used to choose the branch had an illegal value:

```
switch (i)
{
case 1:
   doActionOne();
   break;
case 2:
   doActionTwo();
   break;
default:
   assert(0);
}
```

By passing 0, or false, to the assert, we guarantee the assert will call its error handler if that line of code is reached. If you make a habit of coding all of your switch statements in this way, you will catch a surprising number of illegal values, and you will immediately know which value is at fault.

Compilers and other tools, such as **lint** , sometimes generate warnings in cases where a function does not return a value from some paths. For example:

```
int foo(int i)
{
   switch(i)
   {
   case 1:
```

```
      return 10;
   case 2:
      return 20;
   default:
      assert(0);
   }
}
```

may generate warnings that some paths do not return any value. Most good compilers will allow you to define certain calls that do not return. Other examples are **exit()** and **abort()** . Once you define the **assert()** call as one such operation, the warnings should be suppressed.

Checking hardware

On a desktop system, anything that is generated from a hardware failure could be legitimately caught by an assertion. On an embedded system this is not necessarily true, since an operating system may not be present to take responsibility for the hardware. Hardware will fail at the end of life and the product may need to tolerate this to some extent. Faulty hardware does not represent a bug.

The response to failed hardware may also be different. If a bug causes an assert to fire, it's often likely that the system is capable of resetting itself and running again. For a hardware error, you may want to shutdown completely, or perhaps do some extra tests on the hardware before using it again.

The nature of error reporting may also be very different for a hardware fault. If an assert fires, a design flaw exists. Such failures may need to be raised at a higher level in the field so that designers are given the opportunity to investigate whether the software needs to be fixed.

Because of these differences I never use assertions to check readings from hardware, or to check if hardware is still functioning correctly.

Solid code

One of the most widely cited references on the topic of assertions is *Writing Solid Code* .[2] I recommend it highly for its discussion of assertions as well as much of the other advice it offers.

Maguire is sometimes criticized for suggesting that assertions should not be compiled into the shipped version of the code (using the **NDEBUG** switch discussed above). Considering that he works for Microsoft, and he is writing software for desktop applications, I would say that shipping without the assertions may be the wrong engineering decision, but it is most definitely the correct business decision. Remember, shipping with the assertions enabled will not make the software any more robust. In fact, it can have the opposite effect. In many cases a program will exit when it hits an assert, but had it continued running, the effect of the bug might not have been noticed by the user. You don't

want this to happen in a nuclear power plant, but someone running a word processor may not care that one line was misaligned during a screen refresh routine. Perception of the product will be based far more on the number of crashes than any other inconsistencies. You must also allow for the small percentage of assertions that will actually be checking the wrong expression. The condition they check may just be a bit stricter than was necessary. They may not have fired in testing, but they might fire in the field where a broader range of input is present.

The other thing to consider is what will be done with the assertion information once it is displayed to the user. The people at Microsoft have a lot of bugs to fix, they have plenty of data on where those bugs are, and, like any other desktop software company, they will choose to fix some and not others, putting any surplus effort into developing new functionality. The extra information gathered in the field from assertions firing would probably never be put to much use, and certainly not enough to compensate for the increase in size and decrease in speed of the product.

Away from desktop office applications, the parameters are different. In some cases, a company will have a small number of high-value customers, and it may be vital to fully investigate any failures that occur at those customer's sites.

Many embedded systems are capable of restarting themselves, which is often a reasonable option when an assertion fails. Like most developers working with desktop systems, it was probably not an option for Maguire.

Another reason for keeping asserts in the ship version of an embedded product is that turning the asserts off will change the timing characteristics of the program. On a desktop application, this rarely leads to a different end result. In real-time applications, removing the asserts may lead to misbehavior that did not arise before-and the assertions will not be in place to detect the situation. It's also worth bearing in mind that on the desktop, more speed is always better, while on embedded products, extra speed is often not an advantage, once you meet your hard real-time targets.

Next month we will look at how the **assert()** macro is written and which parts of it you may want to tune to fit the needs of your own project. If you are bug-hunting in the meantime, stop and consider whether a well-placed assertion could have saved you some time.

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. He welcomes feedback and can be reached at . Reader feedback to this column can be found at .

References:

1. Gray, Mike. *Angle of Attack: Harrison Storms and the Race to the Moon.* NY: Penguin Books, 1992.

https://www.embedded.com/assertiveness-training-for-programmers

2. Maguire, Steve. *Writing Solid Code.* Redmond, WA: Microsoft Press, 1993.