Miro Samek

C/C++ Users Journal™
Advanced Solutions for Professional Developers

An Exception or a Bug?

From design by contract to pre-emptive programming: principles for writing better code

Is it an Exception or is it a Bug?

If you were to single out just one most effective programming technique for delivering high-quality code, what would you choose? Object-oriented programming? Generic programming? Design patterns? Software components? Frameworks? UML? CASE tools?

I don't know if these suggestions would top your list, but I suppose that not many programmers would pick assertions (or more scientifically Design by Contract) as *the* single most effective programming technique. Yet, if I were to search my soul, I'd have to admit that none of the aforementioned techniques (not even my favorite hierarchical state machines) has helped me as much as the consistent use of assertions.

In this installment of "The Embedded Angle", I try to explain the Design by Contract philosophy, what can it do for you, and why, I think, embedded systems programmers should care.

Errors versus Exceptional Conditions

In the inaugural installment of this column (February, 2003), I argued that while embedded systems come with their own set of complexities, they also offer many opportunities for simplifications compared to general-purpose computers. Dealing with errors and exceptional conditions provides perhaps the best case in point. Just think, how many times have you seen embedded software terribly convoluted by attempts to painstakingly propagate an error through many layers of code, just to end up doing something trivial with it, such as performing a system reset?

By *error* (known otherwise as a "bug"), I mean a persistent defect due to a design or implementation mistake (e.g., overrunning an array index or writing to a file before opening it). When your software has a bug, typically you cannot reasonably "handle" the situation. You should rather concentrate on detecting (and ultimately fixing) the root cause of the problem. This situation is in contrast to the *exceptional condition*, which is a specific circumstance that can legitimately arise during the system lifetime but is relatively rare and lies off the main execution path of your software. In contrast to an error, you need to design and implement a recovery strategy that handles the exceptional condition.

As an example, consider dynamic memory allocation. In any type of system, memory allocation with `malloc()` (or the C++ operator `new`) can fail. In a general-purpose computer, a failed `malloc()` merely indicates that, at this instant the operating system cannot supply the requested memory. This can happen easily in a highly dynamic general-purpose computing environment. When it happens, you have options to recover from the situation. One option might be for the application to free up some memory that it allocated and then retry the allocation. Another choice could be to prompt the user that the problem exists and encourage her to exit other applications so that the current application can gather more memory. Yet another option is to save data to the disk and exit. Whatever the choice, handling this situation requires some drastic actions, which are clearly off the mainstream behavior of your application. Nevertheless, you should design and implement such actions because in a desktop environment, a failed `malloc()` must be considered an exceptional condition.

In a typical embedded system, on the other hand, the same failed `malloc()` probably should be flagged as a bug. That's because embedded systems offer much fewer excuses to run out of memory, so when it happens, it's typically an indication of a flaw. You cannot really recover from it. Exiting other applications is not an option. Neither is saving data to a disk and exit. Whichever way you look at it, it's a bug no different really from overflowing the stack, dereferencing a `NULL` pointer, or overrunning an array index. Instead of bending over backwards in attempts to handle this condition in software (as you would on the desktop), you should concentrate first on finding the root cause and then fixing the problem (I would first look for a memory leak, wouldn't you?).

The main point here is that many situations traditionally handled as exceptional conditions in general-purpose computing are in fact bugs in embedded systems. In other words, the specifics of embedded systems (computers dedicated to a single, well-defined purpose) allow you to considerably *simplify* the embedded software by flagging many situations as bugs (that you don't need to handle) rather than exceptional conditions (that you do need to handle). The correct distinction between these two situations always depends on the context, so you should not blindly transfer the rules of thumb from other areas of programming to embedded real-time systems. Instead, I propose that you critically ask yourself the following two probing questions: "Can a given situation *legitimately* arise in this particular system?" and "If it happens, is there anything *specific* that needs to or can be done in the software?" If the answer to either of these questions is "yes," then you should handle the situation as an exceptional condition; otherwise, you should treat the situation as a bug.

The distinction between errors and exceptional conditions in any type of software (not just firmware) is important, because errors require the *exact opposite* programming strategy than exceptional conditions. The first priority in dealing with errors is to detect them as early as possible. Any attempt to handle a bug (as an exceptional condition) results in unnecessary complications of the code and either camouflages the bug or delays its manifestation. (In the worst case, it also introduces new bugs.) Either way, finding and fixing the bug will be harder.

Embedding Design by Contract

And here is where the Design by Contract (DbC) philosophy comes in. DbC, pioneered by Bertrand Meyer (e.g., see *Object Oriented Software Construction*, 2nd ed. Prentice Hall, 1997), views a software system as a set of components whose collaboration is based on precisely defined specifications of mutual obligations — the contracts. The central idea of this method is to inherently embed the contracts in the code and validate them automatically at runtime. Doing so consistently has two major benefits: (1) It automatically helps *detect* bugs (as opposed to "handling" them) and (2) It is one of the best ways to document code.

You can implement the most important aspects of DbC (the contracts) in C or C++ with assertions. The standard C-library macro `assert()` is rarely applicable to embedded systems, however, because its default behavior (when the integer expression passed to the macro evaluates to 0) is to print an error message and exit. Neither of these actions makes much sense for most embedded systems, which rarely have a screen to print to and cannot really exit either (at least not in the same sense as a desktop application can). Therefore, in an embedded environment, you usually have to define your own assertions that suit your tools and allow you to customize the error response. I'd suggest, however, that you think twice before you go about "enhancing" assertions, because a large part of their power derives from their relative simplicity.

Listing 1 shows the simple embedded systems-friendly assertions that I've found adequate for a wide range of embedded projects (see also the `qassert.h` header file in the code accompanying the June installment of this column). Listing 1 is similar to the standard `<assert.h>` (`<cassert>` in C++), except: (1) it allows customizing the error response, (2) it conserves memory by avoiding proliferation of multiple copies of the filename string, and (3) it provides additional macros for testing and documenting preconditions (`REQUIRE`), postconditions (`ENSURE`), and invariants (`INVARIANT`). (The names of the three last macros are a direct loan from Eiffel, the programming language that natively supports DbC.)

```
 1: #ifndef qassert_h
 2: #define qassert_h
 3:
 4: /** NASSERT macro disables all contract validations
 5:  * (assertions, preconditions, postconditions, and invariants).
 6:  */
 7: #ifdef NASSERT                    /* NASSERT defined--DbC disabled */
 8:
 9: #define DEFINE_THIS_FILE
10: #define ASSERT(ignore_)  ((void)0)
11: #define ALLEGE(test_)    ((void)(test_))
12:
13: #else                          /* NASSERT not defined--DbC enabled */
14:
15: #ifdef __cplusplus
16: extern "C" {
17: #endif
18:                   /* callback invoked in case of assertion failure */
19: void onAssert__(char const *file, unsigned line);
20:
21: #ifdef __cplusplus
22: }
23: #endif
24:
25: #define DEFINE_THIS_FILE \
26:    static char const THIS_FILE__[] = __FILE__
27:
28: #define ASSERT(test_) \
29:    ((test_) ? (void)0 : onAssert__(THIS_FILE__, __LINE__))
30:
31: #define ALLEGE(test_)    ASSERT(test_)
32:
33: #endif                                                /* NASSERT */
34:
35: #define REQUIRE(test_)   ASSERT(test_)
36: #define ENSURE(test_)    ASSERT(test_)
37: #define INVARIANT(test_) ASSERT(test_)
38:
39: #endif                                              /* qassert_h */
```

Listing 1 Embedded systems-friendly assertions

The all-purpose `ASSERT()` macro (lines 28-29) is very similar to the standard `assert()`. If the argument passed to this macro evaluates to 0 (`false`) and if additionally the macro `NASSERT` is not defined, then `ASSERT()` will invoke a global callback `onAssert__()`. The function `onAssert__()` gives the clients the opportunity to customize the error response when the assertion fails. In embedded systems, `onAssert__()` typically first monopolizes the CPU (by disabling interrupts), then possibly attempts to put the system in a fail-safe mode, and eventually triggers a system reset. (Many embedded systems come out of reset in a fail-safe mode, so putting them in this mode before reset is often unnecessary.) If possible, the function should also leave a trail of bread crumbs from the cause, perhaps by storing the filename and line number in a non-volatile memory. (The entry to `onAssert__()` is also an ideal place to set a breakpoint if you work with a debugger. TIP: Consult your debugger manual how you can hard-code a *permanent* breakpoint in `onAssert__()`.)

Compared to the standard `assert()`, the macro `ASSERT()` conserves memory (typically ROM) by passing `THIS_FILE__` (Listing 1, line 26) as the first argument to `onAssert__()`, rather than the standard preprocessor macro `__FILE__`. This avoids proliferation of the multiple copies of the `__FILE__` string, but requires invoking macro `DEFINE_THIS_FILE` (line 25), preferably at the top of every C/C++ file. (Steve Maguire describes this technique in *Writing Solid Code*, Microsoft Press, 1993.)

Defining the macro `NASSERT`(Listing 1, line 7) disables checking the assertions. When disabled, the assertion macros don't generate any code (line 10, and 35-37); in particular, they don't test the expressions passed as arguments, so you should be careful to avoid any side-effects (required for normal program operation) inside the expressions tested in assertions. The notable exception is the `ALLEGE()` macro (lines 11 and 31), which always tests the expression, although when assertions are disabled, it does not invoke the `onAssert__()` callback. `ALLEGE()` is useful in situations where avoiding side-effects of the test would require introducing temporaries, which involves pushing additional registers onto the stack—something you often want to minimize in embedded systems.

The DbC Philosophy

The most important point to understand about software contracts (assertions in C/C++) is that they neither handle nor prevent errors, in the same way as contracts between people do not prevent fraud. For example, asserting successful memory allocation: `ALLEGE((foo = new Foo) != NULL)`, might give you a warm and fuzzy feeling that you have handled or prevented a bug, when in fact, you haven't. You did establish a contract, however, in which you spelled out that the inability to dynamically allocate object `Foo` at this spot in the code is an error. From that point on the contract will be checked automatically and sure enough the program will brutally abort if the contract fails. At first you might think that this must be backwards. Contracts not only do nothing to handle (let alone fix) bugs, but they actually make things worse by turning every asserted condition, however benign, into a fatal error! However, recall from the previous discussion that the first priority when dealing with bugs is to detect them, not to handle them. To this end, a bug that causes a loud crash (and identifies exactly which contract was violated) is much easier to find than a subtle one that manifests itself intermittently millions of machine instructions downstream from the spot where you could have easily detected it.

Assertions in software are in many respects like fuses in electrical circuits. Electric engineers insert fuses in various places of their circuits to instill a controlled damage (burning a fuse) in case the circuit fails or is mishandled. Any non-trivial circuit, such as the electrical system of a car, has a multitude of differently rated fuses (a 20A fuse is appropriate for the headlights, but it's way too big for the turn signals) to better help localize problems and to more effectively prevent expensive damage. On the other hand, a fuse can neither prevent nor fix a problem, so replacing a burned fuse doesn't help until the root cause of the problem is removed. Just like with assertions, the main power of fuses derives from their simplicity.

Due to the simplicity, however, assertions are sometimes viewed as too primitive error checking mechanism—something that's perhaps good enough for smaller programs, but must be replaced with a "real" error handling in the industry-strength software. This view is inconsistent with the DbC philosophy, which regards contracts (assertions in C/C++) as the integral part of the software design. Contracts embody important design decisions, namely declaring certain conditions as errors rather than exceptional conditions, and therefore embedding them in large-scale, industry-strength software is even more important than in quick-and-dirty solutions. Imagine building a large industrial electrical circuit (say, a power plant) without fuses…

Defensive or Preemptive Programming?

The term "defensive programming" seems to have two complementary meanings. In the first meaning, the term is used to describe a programming style based on assertions, where you explicitly assert any assumptions that should hold true as long as the software operates correctly (e.g., see Chapter 2 of *Thinking in C++, Volume 2*, 2nd ed. by Bruce Eckel and Chuck Allison, available online from <*www.mindview.net/Books/TICPP/-ThinkingInCPP2e.html*>). In this sense, "defensive programming" is essentially synonymous with DbC.

In the other meaning, however, "defensive programming" denotes a programming style that aims at making operations more robust to errors, by accepting a wider range of inputs or allowing order of operations not necessarily consistent with the object's state. In this sense, "defensive programming" is *complementary* to DbC. For example, consider the following hypothetical output port class:

```
class Port {
   bool open_;
public:
   Port() : open_(false) {}
   void open() {
      if (!open_) {
         // open the port ...
         open_ = true;
      }
   }
   void transmit(unsigned char const *buffer, unsigned nBytes) {
      if (open_ && buffer != NULL && nBytes > 0) {
         // transmit nBytes from the buffer ...
      }
   }
   void close() {
      if (!open_) {
         open_ = false;
         // close the port ...
      }
   }
   // . . .
};
```

This class is programmed defensively (in the second meaning of the term), because it *silently* accepts invoking operations out of order (e.g., `transmit()` before `open()`) with invalid parameters (e.g., `transmit(NULL, 0)`).

Defensive programming is often advertised as a better coding style, but unfortunately, it often hides bugs. Is it really a good program that calls `port.transmit()` before `port.open()`? Is it really OK to invoke `transmit()` with un-initialized transmit buffer? I'd argue that a correctly designed and implemented code should not do such things, and when it happens it's a sure indication of a larger problem.

In comparison, the `Port` class coded according to the DbC philosophy would use preconditions:

```
class Port {
   bool open_;
public:
   Port() : open_(false) {}
   void open() {
      REQUIRE(!open_);
      // open the port ...
      open = true;
   }
   void transmit(unsigned char const *buffer, unsigned nBytes) {
      REQUIRE(open_ && buffer != NULL && nBytes > 0);
      // transmit n-bytes from the buffer ...
   }
   void close() {
      REQUIRE(open_);
      open_ = false;
      // close the port ...
   }
   // . . .
};
```

This implementation is intentionally less flexible, but unlike the defensive version, it's hard to use this one incorrectly. Additionally, (although difficult to convincingly demonstrate with a toy example like this) assertions tend to eliminate a lot of code that you would have to invent to handle the wider range of inputs allowed in the defensive code.

But there is more, much more to DbC than just complementing defensive programming. The key to unveiling DbC's full potential is to *preemptively* look for conditions to assert. The most effective assertions are discovered by asking two simple questions: "What are the implicit assumptions for this code and the client code to execute correctly?", and "How can I explicitly and most effectively test these assumptions?". By asking these questions for every piece of code you write, you'll discover valuable conditions that you wouldn't test otherwise. This way of thinking about assertions leads to a paradigm shift from "defensive" to "preemptive" programming, in which you preemptively look for situations that have even a potential of breeding bugs.

To this end, embedded systems are particularly suitable for implementing such a "preemptive doctrine". Embedded CPUs are surrounded by specialized peripherals that just beg to be used for validating correct program execution. For example, a serial communication channel (say, a 16550-type UART) might set a bit in Line Status Register when the input FIFO overflows. A typical serial driver ignores this bit (after all, the driver cannot recover bytes that already have been lost), but your driver can *assert* that the FIFO overrun bit is never set. By doing so, you'll know when your hardware has lost bytes (perhaps due to an intermittent delay in servicing the UART), which is otherwise almost impossible to detect or reproduce at will. Needless to say, with this information you'll not waste your time on debugging the protocol stack, but instead you'll concentrate on finding and fixing a timing glitch. You can use timer/counters in a similar way to build *real-time assertions* that check if you miss your deadlines. In my company, we've used a special register of a GPS correlator chip to assert that every GPS channel is always serviced within its C/A code epoch (around every 1 ms)—yet another form of a real-time assertion. The main point of these examples is that the information available almost for free from the hardware wouldn't be used if not for the "preemptive" assertions. Yet, the information is invaluable, because it's often the only way to directly validate the time-domain performance of your code.

Assertions and Testing

At GE Medical Systems, I once got involved in developing an automatic testing suite for diagnostics X-ray machines, which we called "cycler". The cycler was essentially a random monkey program that emulated activating soft-keys on our touch screen and depressing the foot-switch to initiate X-ray exposures. The idea was to let the cycler exercise the system at night and on weekends. Indeed, the cycler helped us to catch quite a few problems, mostly those that left entries in the error log. However, because our software was programmed mostly defensively, in absence of errors in the log we didn't know if the "cycler" run was truly successful, or perhaps the code just wondered around all weekend long silently "taking care" of various problems.

In contrast, every successful test run of code peppered with assertions builds much more confidence in the software. I don't know exactly what the critical density of assertions must be, but at some point the tests stop producing undefined behavior, segmentation faults, or system hangs—*all* bugs manifest themselves as assertion failures. This effect of DbC is truly amazing. The integrity checks embodied in assertions prevent the code from "wondering around" and even broken builds don't crash-and-burn but rather end up hitting an assertion.

Testing code developed according to DbC principles has an immense psychological impact on programmers. Because assertions escalate every asserted condition to a fatal error, all bugs require attention. DbC makes it so much harder to dismiss an intermittent bug as a "glitch"—after all, you have a record in form of the filename and the line number where the assertion fired. Once you know where in the code to start your investigations, most bugs are more transparent.

Shipping with Assertions

The standard practice is to use assertions during development and testing, but to disable them in the final product by defining the `NDEBUG` macro. In Listing 1, I have replaced this macro with `NASSERT`, because many development environments define `NDEBUG` automatically when you switch to the production version, and I wanted to decouple the decision of disabling assertions from the version of software that you build. That's because I truly believe that leaving assertions enabled, *especially* in the ship-version of the product, is a good idea.

The often-quoted opinion in this matter comes from C.A.R. Hoare, who considered disabling assertions in the final product like using a lifebelt during practice, but then not bothering with it for the real thing. I find the comparison of assertions to fuses more compelling. Would you design a prototype board with carefully rated fuses, but then replace them all with 0Ω resistors (chunky pieces of wire) for a production run?

The question of shipping with assertions really boils down to two issues. First is the overhead that assertions add to your code. Obviously, if the overhead is too big, you have no choice. (But then I must ask how have you built and tested your firmware?) However, assertions should be considered an integral part of the firmware and properly sized hardware should accommodate them. As the prices of hardware rapidly drop and its capabilities skyrocket, it just makes sense to trade a small fraction of the raw CPU horsepower and memory resources for a better system integrity. In addition, as I mentioned earlier, assertions often pay for themselves by eliminating reams of defensive code.

The other issue is the correct system response when an assertion fires in the field. As it turns out, a simple system reset is for most embedded devices the least inconvenient action from the user's perspective—certainly less inconvenient than locking-up a device and denying service indefinitely. That's exactly what happened the other day, when my wife's cellular phone froze and the only way of bringing it back to life was to pull out the battery (I don't know how she does it, but since then she managed to hang her phone again more than once, along with our VCR and even the TV). The question that comes to my mind is whether the firmware in those products used assertions (or whether the assertions have been enabled)—apparently not, because otherwise the firmware would have reset automatically.

End Around Check

Assertions have been a recurring subject of many articles (and rightly so). For example, two recent articles in the *CUJ Experts Forum* by Andrei Alexandrescu describe how you can unleash templates and exceptions to build truly smart assertions (see: "Generic<Programming>: Assertions", <www.cuj.com/experts/2104/-alexandr.htm>, April, 2003, and "Generic <Programming>: Enforcements", <www.cuj.com/experts/2106/-alexandr.htm>, June 2003). More specifically to embedded systems, I greatly enjoyed two successive installments of Niall Murphy's column in the *Embedded Systems Programming* magazine devoted to assertions (see "Assertiveness Training for Programmers", *ESP*, April, 2001, and "Assert Yourself", *ESP*, May, 2001, both available online from <www.panelsoft.com/murphyslaw>). By the way, the analogy between assertions in software and fuses in electrical systems was Niall's original idea, which came up when we talked about assertions at the latest Embedded Systems Conference in San Francisco.

"When?", and "How should I use assertions?". The main goal of this article is to convince you that the DbC philosophy can fundamentally change the way you design, implement, test, and deploy your software. A good starting point to learn more about DbC is the Eiffel Software website at <www.eiffel.com> (in particular, you can find there an interesting interpretation if the infamous Ariane 5 software failure.)

In the embedded systems domain, the days of logic analyzers or in-circuit emulators having direct access to all of the CPU's state information are long gone. Even if you had access to all the CPU's address and data signals (which you typically don't, because there are simply not enough pins to go around), the multi-stage pipelines and cache memories make it impossible to figure out what's going on in there. The solution requires the testing instrumentation (assertions) integrated directly into the system's firmware. You can no longer design a system without accounting for testing overhead right from the start. Assuming that all the CPU cycles, the RAM, and all the ROM will be devoted strictly to the job at hand simply won't get the job done. ❑

Miro Samek is the author of *Practical Statecharts in C/C++: Quantum Programming for Embedded Systems*, CMP Books, 2002. He is the lead software architect at IntegriNautics Corporation (Menlo Park, CA) and a consultant to industry. Miro previously worked at GE Medical Systems, where he has developed safety-critical, real-time software for diagnostic imaging X-ray machines. He earned his Ph. D. in nuclear physics at GSI (Darmstadt, Germany) where he conducted heavy-ion experiments. Miro welcomes feedback and can be reached at `miro@quantum-leaps.com`.