DAVID B. STEWART

<BACK

30 Pitfalls for Real-Time Software Developers, Part 1

The path to successful real-time software development is strewn with pitfalls along the way that can trap the unwary programmer. This month and next the author guides you past 30 of them.

Novices and experts alike, whether in a university or corporation, repeat the same mistakes over and over again when developing real-time software. I have observed this while reviewing and grading code in academic projects, and as a consultant involved in numerous design and code reviews for industry.

Most real-time software developers are not even aware that their favorite methods can be problematic. Quite often, experts are self-taught; hence they tend to have the same bad habits as when they first began, usually because they never witnessed better ways of programming their embedded systems. These experts then train novices, who subsequently acquire the same bad habits. The purpose of this article is to improve your awareness of common problems, and to provide a start towards eliminating mistakes and thus creating software that is both more reliable and easier to maintain.

This list first began as the 10 most common pitfalls, but there were just so many common mistakes and problems that the list grew. It expanded through 15 and 25, to its present number. This month, I'll present problems 30 through 16; the rest I'll lay out for you next month.

For each problem, I present the misconception or source of the problem. Then I offer possible solutions or alternatives that can help minimize or eliminate the mistakes. If you're not familiar with the details or terminology of the alternate solutions, then a quick library or Web search should yield additional literature on the topic. While there is usually agreement about most items being mistakes, some of the mistakes listed and the corresponding proposed solutions may be controversial. In such cases, simply

Many designers and programmers refuse to listen to the experiences of others, claiming that their applications are different, and of course, much more complicated.

highlighting that there is a disagreement as to what is the best way to alleviate these problems encourages designers to compare their methods to other approaches, and to reconsider if their methods are provably better.

Correcting just *one* of these mistakes within a project can lead to weeks or months of savings in manpower (especially during the maintenance phase of a software life cycle) or can result in a significant increase in quality and robustness of the application. If multiple mistakes are common and they are all fixed, potential company savings or additional profits can be in the thousands or millions of dollars. Thus I encourage you to review your current methods and policies, compare them to each of the reported mistakes and the proposed alternatives, and decide for yourself if potential savings exist for your company or project. Even if there are no direct savings, consider the potential for improved quality and robustness at no extra cost by modifying some of your current practices.

Here now are the first 15 of the 30 most common mistakes; problems that are higher on the list (where #30 is lowest and #1 is highest on list) are either more common and/or have the most impact on quality, development time, and software maintenance. Naturally, the order represents my opinion. It's not so important that one mistake is listed higher on the list than another. What is important is that both are listed, thus both may be significant in your specific environment.

#30 "My problem is different."

Many designers and programmers refuse to listen to the experiences of others, claiming that their applications are different, and of course, much more complicated. Designers should be more open-minded about the similarities in their work. Even what seems like the most different applications are probably nearly identical when you consider the nuts and bolts of the real-time infrastructure. For example, communications engineers will claim their applications have no similarities to systems designed by control engineers because of the high volume of data and the need for special processors such as digital signal processors (DSPs). In response, ask "What is different in the LCD display software in a cellular phone vs. one in a temperature controller? Are they really different?"

Comparing control and communication systems side-by-side, both are characterized by modules that have inputs and outputs, with a function that maps the input to the output. A 256 x 256 image processed by a DSP algorithm might not be that different from graphical code for an LCD dot matrix display of size 320 x 200. Furthermore, both use hardware with limited memory and processing power relative to the size of the application; both require development of software on a platform distinct from the target, and many of the issues in developing software for a DSP also apply to developing software for a microcontroller.

The timing and volume of data are different. But if the system is designed correctly, these are just variables in equations. Methods to analyze resources such as memory and processing time are the same—both may require similar real-time scheduling, and both may also have high-speed interrupt handlers that can cause priority inversion.

Perhaps if control systems and communication systems are similar, so are two different control applications or two different communication systems. Every application is unique, but more often than not the procedure to specify, design, and build the software is the same. Embedded software designers should learn as much as possible from the experiences of others and not shrug off experience just because it was acquired in a different application area.

#29 Tools choice driven by marketing hype, not by evaluation of technical needs

Software tools for embedded systems are often purchased based on the flashiness of the marketing, because a lot of other people are using them, or because of a feature that sounds appealing but really does not make a difference.

Flashiness. Just because one tool has a prettier graphical user interface than another does not make it better. It's important to consider the technical capabilities of each, relative to the needs of the application being built.

Number of users. Buying software from a vendor just because it's the biggest does not mean it's the best. Along with pitches that more people are using the software are probably hidden true stories that more people are paying for more than they really need, or that more people have unused versions of the tools sitting on the shelf after discovering the tools were not suited to their needs.

Promises of compatibility. Managers are especially influenced by a product because of promises of compatibility. So what if software is 100% POSIX-compliant? What is its relevance? Is there a plan to change the operating system? Suppose there is a change to another POSIX-compliant operating system—what is there to gain? Absolutely nothing, unless "extensions" are used. But if such extensions are used, compatibility is lost, hence

the benefits are no longer there. Standards such as POSIX have not been proven to even be good for real-time systems, let alone the best. Therefore, don't assume that the product is better because of that promise. Portability and reusability can only be achieved if all the designers follow proven software engineering strategies for developing component-based software.[1,2]

When selecting tools, consider the needs of the application first; then investigate the dozens (or hundreds) of options available from a *technical* perspective, as they relate specifically to the application requirements. The best tools for a particular design or application are not necessarily the most popular.

#28 Large if-then-else and case statements

It's not uncommon to see large *if-else* statements or *case* statements in embedded code. These are problematic from three perspectives:

- Such statements are extremely difficult to debug, because code ends up having so many different paths. If statements are nested it becomes even more complicated
- The difference between best-case and worst-case execution time becomes significant. This leads to either under-utilizing the CPU, or the possibility of timing errors when the longest path is taken
- The difficulty of structured code coverage testing grows exponentially with the number of branches, so branches should be minimized

Computational methods can often provide an equivalent answer. Performing Boolean algebra, implementing a finite state machine as a jump table, or using lookup tables are alternatives that can reduce a 100-line if-else statement to less than 10 lines of code.

Here is a trivial example of converting an if statement to Boolean algebra:

```
if (x == 1)
  x=0;
else
  x=1
```

Instead, a Boolean algebra computation would be the following:

```
x = !x; // x = NOT x; can also use
        // x = 1-x
```

Despite the simplicity, many programmers still toggle a Boolean value with the if statement above.

#27 Delays implemented as empty loops

Real-time software often uses delays to ensure that data sent or received over an I/O port has time to propagate. These delays are frequently implemented by putting a few no-ops or empty loops (assuming `volatile` is used if the compiler performs optimizations). If this code is used on a different processor, or even the same processor running at a different rate (for example, a 25MHz vs. 33MHz CPU), the code may stop working on the faster processor. This is especially something to avoid, since it results in the kind of timing problem that is extremely difficult to track down and solve, because the symptoms of the problem are sporadic.

Instead, use a mechanism based on a timer. Some RTOSes provide these functions, but if not, one can still easily be built. Following are two possibilities to build a custom `delay(int usec)` function.

Most count-down timers allow the software to read a register to obtain the current count-down value. A system variable can be saved to store the rate of the timer, in units such as microseconds per tick. Suppose the value is 2µs per tick, and a delay of 10µs is required: the delay function busy-waits for five timer ticks. Suppose a different speed processor is used—the timer ticks are still the same. Or if the timer frequency changes, then the system variable would change, and the number of ticks to busy-wait would change, but the delay time would remain the same.

If the timer doesn't support reading intermediate count-down values, an alternative is to profile the speed of the processor during initialization. Execute an empty loop continuously and count how often it occurs between two timer interrupts. Since frequency of the timer interrupt is known, a value for the number of microseconds per iteration can be computed. This value is then used to dynamically determine how many iterations of the loop to perform for a specified delay time. In our custom RTOS with this implementation, the delay function was accurate within 10% of the desired time for any processor with which we tested it, without ever having to change the code.

#26 Interactive and incomplete test programs

Many embedded designers create a series of test programs, each program testing a separate feature. Test programs need to be executed one at a time, and in some cases require the user to provide input (say, through a keypad or switch) and observe the output response. The problem with this method is that programmers tend only to test what they are changing. Since there are often interactions between unrelated code due to the sharing of resources, every time a change is made, the entire system should undergo testing.

To accomplish this, avoid interactive test programs. Create a single test program that goes through as much self-testing as possible, so that any time even the smallest change is made, a complete test can easily and quickly be performed.

Unfortunately, this is more easily said than done. Some testing, especially of I/O devices, can only be done interactively. Nevertheless, the principle of automated testing should be at the forefront of any attempt to create test software, and not a side-thought with test code written only on an as-needed basis.

#25 Reusing code not designed for reuse

Code that is not designed for reuse will not be in the form of an abstract data type or object. The code may have interdependencies with other code, such that if all of it is taken, there is more code than needed. If only part is taken, it must be thoroughly dissected, which increases the risk of unknowingly cutting out something that is needed, or unexpectedly changing the functionality. If code isn't designed for reuse, it's better to analyze what the existing code does, then redesign and re-implement the code as well-structured reusable software components. From there on, the code can be reused. Rewriting this module will take less time than the development and debugging time needed to reuse the original code.

A common misconception is that because software is defined in separate modules, it is naturally reusable. This is a separate mistake on its own, related to creating software with too many dependencies. See more details in mistake #18.

#24 Generalizations based on a single architecture

Embedded software designers may have the need to develop software that is intended to run on a variety of processors and platforms. In such a case, it's not uncommon for the programmer to begin writing software for one of the platforms, but generalize anything and everything in preparation for porting the code at a later time.

Unfortunately, doing so usually causes more harm than good. The design will tend to over-generalize items that are very similar on very different architectures, while not generalizing some items that are different, but that the designer did not foresee as different.

A better strategy is to design and develop the code simultaneously on multiple architectures, generalizing only those parts that are different in the different architectures. Intentionally choose three or four processors that are very different (for example, from different manufacturers and using different architectures).

#23 One big loop

When real-time software is designed as a single big loop, we have no flexibility to modify the execution time of various parts of the code independently. Few real-time systems need to operate everything at the same rate. If the CPU is overloaded, one of the methods to reduce utilization is to selectively slow down only the less critical parts of the code. This approach works, however, only if the multitasking features of an RTOS are used, or the code was developed based on a flexible custom or commercial real-time executive.

#22 No analysis of hardware peculiarities before starting software design

How long does it take to add two eight-bit numbers? What about two 16-bit or 32-bit numbers? What about two floats? What if an eight-bit number is added to a float? A software designer who cannot answer these questions off the top of his or her head for the target processor isn't adequately prepared to design and code real-time software.

Here are sample answers to the above measurements for a 6MHz Z180 (in microseconds): 7, 12, 28, 137, and 308. Note that it takes 250% more time to do float plus byte than float plus float, due to the long conversion time from byte to float. Such anomalies are often the source of code that overloads the processor.

In another example, a special purpose floating-point accelerator did floating-point addition/multiplication 10 times faster than a 33MHz 68882, but `sin()` and `cos()` took the same amount of time. This is because the 68882 has the trigonometric functions built into its hardware, while the floating point accelerator did those particular functions in software.

When code is implemented for a real-time system, being aware of the timing implications of every single line of code is important. Understand the capabilities and limitations of the target processor(s), and redesign an application that makes excessive use of slow instructions. For example, for the Z180, doing everything in float is better than having only some variables float and lots of mixed-type arithmetic.

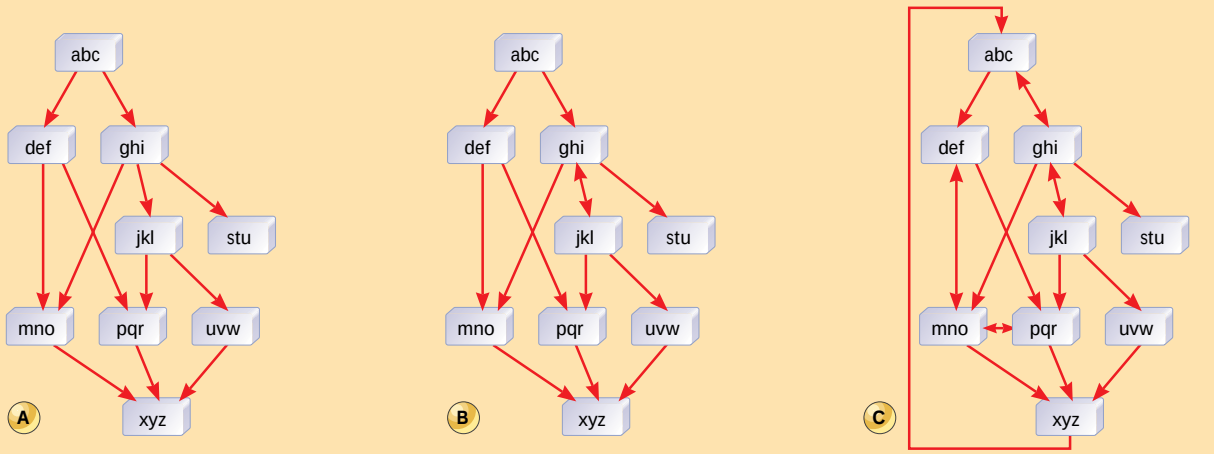
#21 Over-designing the system

If the processor and memory utilization are less than 90% on average and less than 100% peak, then the system has probably been over-designed. Writing programs for a processor with more than enough resources is a luxury for a software developer. In some cases, however, this luxury is so costly that it can make the difference between a profit and bankruptcy! Contributing towards minimizing the price and power consumption of an embedded system is a software engineer's duty. If the CPU is only 45% utilized, you can use a processor that operates at half the speed instead, thus saving as much as four times the power and possibly one or more dollars per processor.

If the product is mass-produced, saving $1 on the processor could save a million dollars over the production span of the item. If the product is battery-powered, it will allow the battery to last much longer, thus increasing the marketing appeal of the product. As an extreme example of power consumption of computers, consider a laptop. Most have less than three hours of power when using a heavy battery. A watch, however, has a lightweight, cheap battery that can last three years. Although software isn't usually associated with power consumption, it does have a major role.

Fast processors and more memory than necessary tend to also lead to laziness in thinking about the design. Start embedded development with slower processors with less memory, and move up to the next level of processor only on an as-needed basis. Software that uses hardware more effi-

FIGURE 1 Examples of dependency graphs, with and without cycles. An objective in developing good software is to decompose code into modules to minimize or eliminate circular dependencies. a) Dependency graph with no cycles. This is desirable. b) Dependency graph with cycle between *ghi* and *jkl*. c) Dependency graph with many circular dependencies, including a major circular dependency.



ciently is more likely to evolve from this approach than from later trying to cut corners to bring down the cost of the system.

#20 Fine-grain optimizing during first implementation

The converse to problem #21 is also a common mistake. Some programmers foresee anomalies (some are real, some are mythical). An example of a mythical anomaly is that multiplication takes much longer than addition. Many designers would implement 3**x* as *x*+*x*+*x*. On many embedded processors, however, multiplication is less than twice as long as addition, so *x*+*x*+*x* would be slower than 3**x*.

A programmer who foresees all the anomalies may implement the first version of the code in an unreadable manner so as to optimize the code; this is before knowing if optimization is even needed. As a general rule, don't perform fine-grained optimizations during implementation. Only optimize segments of code later if it proves necessary to get better performance. If optimization is unnecessary, then keep the more readable code. If the CPU is overloaded, it's nice to know that a variety of places remain in the code where simple, straightforward optimizations can be performed quickly.

#19 "It's just a glitch."

Some programmers use the same workarounds over and over again because the system has a glitch. A programmer's typical response is that it always executes well if the workaround is used.

Unfortunately, the same errors that force a workaround are likely to resurrect themselves later in a different form. Anytime there is any "glitch," it means something is wrong! Make sure appropriate steps are taken to understand the problem. A workaround may be valuable to ensure that a product is shipped on time, but immediately after the deadline, take a bit of extra time to identify the problem, to ensure it does not show up again—such as during the next big demo.

#18 Too many inter-module dependencies

The dependencies between modules in a good software design can be drawn as a tree, as shown in Figure 1a. A dependency diagram consists of nodes and arrows, such that each node represents a module (such as one source code file), and the arrows show dependencies between that node and other modules. Modules on the bottom-most row are not dependent on any other software module. To maximize software reusability, arrows should always point downwards, and not upwards or bidirectionally. For example, module *abc* depends on module *def* if it has a `#include` "*def.h*" in the code, or an `extern` declaration in the file *abc.c* to a variable or function defined in module *def.c*.

The dependency graph is a valuable software engineering aid. Given such a diagram, it's easy to identify what parts of the software can be reused, create a strategy for incremental testing of modules, and develop a method to limit error propagation through the entire system.

Each circular dependency (a cycle in the graph) reduces the ability to reuse the software module. Testing can only occur for the combined set of dependent modules, and errors will be difficult to isolate to a single module. If the graph has too many

cycles, or a major cycle exists where a module at the bottom-most level of the graph is dependent on the top-most module, then not a single module is reusable.

Figures 1b and 1c both include circular dependencies. If a circular dependency is inevitable, Figure 1b is much preferred over Figure 1c, since in 1b reusing some of the modules is still possible. The restriction in Figure 1b is that modules *pqr* and *xyz* can only be reused together. In Figure 1c, however, reusing any subset of modules isn't possible, as too many dependencies exist between modules. Furthermore, a major circular dependency exists, where module *xyz*—which should not be dependent on anything because it is at the bottom of the graph—is dependent on *abc*. Only one such major cycle is required to make the entire application non-reusable. Unfortunately, most existing applications are more similar to Figure 1c than to Figure 1a or Figure 1b, hence the difficulty in reusing software from existing applications.

To best use dependency graphs to analyze the reusability and maintainability of software, write code that makes it easy to generate the graph. That is, all `extern` declarations for exported variables in functions in a module *xxx* should be defined in file *xxx.h*. In module *yyy*, simply looking at what files are `#include`'d allows determination of that module's dependencies. If this convention is not followed, and an `extern` declaration is embedded in *yyy.c* instead of `#include`ing the appropriate file, then the dependency graph will be erroneous and an attempt to reuse code that appears to be independent of the other module will be difficult.

#17 "I don't have time to take a break."

Many programmers struggle nonstop for hours on a problem, only to hit dead end after dead end. They continue because they face a deadline. Many hours could be saved if the person simply took a break after not making any progress for an hour. Relax, take a walk around a lake, go for a beer, take a nap—anything.

With a clear mind that results from a bit of mental relaxation, analyzing what is happening is much easier, and you can more quickly converge to a solution. A two-hour break—even with a deadline looming—might save a day of work. A 10-minute coffee break away from the computer can sometimes save an hour of work.

#16 Using message passing as primary inter-process communication

When software is developed as functional blocks, the first thought is to implement inputs and outputs as messages. Although this works well in non-real-time environments—such as for distributed networking—it's problematic in a real-time system.

Three major problems arise when using message passing in a real-time system:

- Message passing requires synchronization, a primary source of unpredictability to real-time scheduling. Functional blocks end up executing synchronously, and thus analysis of the system's timing is difficult, if not impossible
- In systems with bi-directional communication between processes or any kind of feedback loop, deadlock is a possibility
- Message passing incurs significantly more overhead as compared to shared memory. While messages may be required for communication across networks and serial lines, it's often inefficient when random-access to the data is possible, as is the case for interprocess communication on a single processor

State-based communication is preferred in embedded systems to provide higher assurability. A state-based system uses structured shared memory, such that communication has less overhead. The most recent data is always available to a process when the process needs it. Steenstrup and Arbib developed the port-automation theory to formally prove that a stable and reliable control system can be created by only reading the most recent data.[3] Costly blocking is eliminated by creating local copies of shared data, to ensure that every process has mutually exclusive access to the information it needs.[2] Using states instead of messages also provides robustness if the possibility of lost messages exists, if code does not all execute at the same rate, and if implementing with shared memory generates less operating system overhead.

Converting control systems from message-based communication to state-based communication is generally straightforward. For example, an intelligent train control system has independent control of every brake to maximize train handling. To minimize stopping distance when coming to a full stop, all the brakes on the train must be applied together. The I/O logic for each brake is handled by a separate process; the control module must inform each brake module to turn on the brakes. When using a message-based system, the controlling unit sends a message, "apply brake," to every brake process. This approach has high communication overhead, potential loss of messages if tasks execute at different frequencies, nondeterministic blocking, a separate copy of the message for every process, and the possibility of deadlock. Due to the dependencies among processes, it creates a real-time system that is difficult to analyze and is not suitable for reconfigurable systems. In contrast, in a state-based communication mechanism, each brake module executes periodically and monitors the brake variable to update the state of its own brake I/O. For example, instead of the "apply brake" message, revise the state of the brake variable so that it says, "the brake should be on." Since processes are periodic, a schedulability analysis is easier. Processes only need to bind to a single element in the

state table, thus eliminating direct dependencies between processes. Communication through shared memory also incurs less overhead when compared to a message-passing system.

When transferring a stream of data between objects, a producer/consumer-type buffer should be created in shared memory, such that the maximum amount of data that is processed during each periodic cycle is software-controlled.

And so you have the first half of my list of 30 pitfalls. Next month, I'll present the top 15. **esp**

Dr. David B. Stewart is an assistant professor at University of Maryland. He earned his PhD in computer engineering from Carnegie Mellon University. He teaches and consults in the area of real-time embedded software. He has led large student projects, including the Pinball Machine Project that was demonstrated in Las Vegas, and the Computer-Controlled Electric Train project that received an honorable mention in the 1998 Motorola University Design Contest. His research focuses on next generation RTOS and middleware technology to support the rapid design and analysis of dynamically reconfigurable real-time software. He may be contacted through his home page at www.ece.umd.edu/~dstewart.

References

1. D.B. Stewart, "Designing Software Components for Real-Time Applications," in Proceedings of Embedded Systems Conference, San Jose, CA, September 1999.
2. D.B. Stewart, R.A. Volpe, and P.K. Khosla, "Design of dynamically reconfigurable real-time software using port-based objects," IEEE Trans. on Software Engineering, v. 23, n. 12, Dec. 1997.
3. M. Steenstrup, M. Arbib, and E.G. Manes. "Port Automata and the Algebra of Concurrent Processes," *Journal of Computer and System Sciences*, v. 27, n.1, pp. 29-50, Jan. 1983.