Five top causes of nasty embedded software bugs

Michael Barr

4/1/2010 12:00 AM EDT

Too often engineers give up trying to discover the cause of infrequent anomalies not easily reproduced in the lab. Yet these ghosts in the machine live on.

Editor's note: See the next five bugs in part 2.

Finding and killing latent bugs in embedded software is a difficult business. Heroic efforts and expensive tools are often required to trace backward from an observed crash, hang, or other unplanned run-time behavior to the root cause. In the worst cases, the root cause damages the code or data in a way that the system still appears to work fine or mostly fine-at least for a while.

Too often engineers give up trying to discover the cause of infrequent anomalies that cannot be easily reproduced in the lab- dismissing them as user errors or "glitches." Yet these ghosts in the machine live on. Here's a guide to the most frequent root causes of difficult to reproduce bugs. Look for these top five bugs whenever you are reading firmware source code. And follow the recommended best practices to prevent them from happening to you again.

Bug 1: Race condition
A race condition is any situation in which the combined outcome of two or more threads of execution (which can be either RTOS tasks or **`main()`** and an interrupt handler) varies depending on the precise order in which the interleaved instructions of each are executed on the processor.

For example, suppose you have two threads of execution in which one regularly increments a global variable (**`g_counter += 1;`**) and the other very occasionally zeroes it (**`g_counter = 0;`**). There is a race condition here if the increment cannot always be executed atomically (in other words, in a single instruction cycle). Think of the tasks as cars approaching the same intersection, as illustrated in **Figure 1**. A collision between the two updates of the counter variable may never or only very rarely occur. But when it does, the counter will not actually be zeroed in memory that time; its value is corrupt at least until the next zeroing. The effect of this may have serious consequences for the system, although perhaps not until a long time after the actual collision.

Each shared resource is an accident waiting to happen.

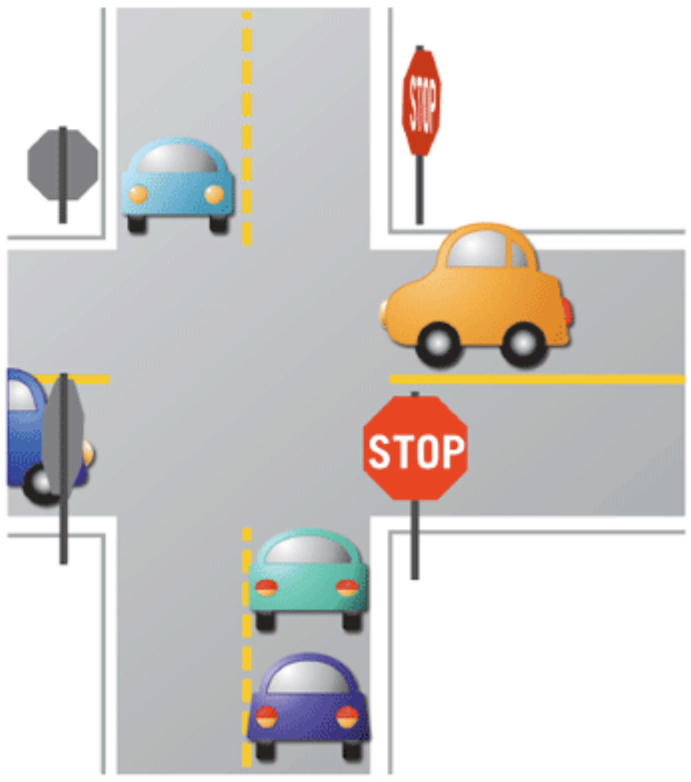


Figure 1

• **Best practice:** Race conditions can be prevented by surrounding critical sections of code that must be executed atomically with an appropriate preemption--limiting pair of behaviors. To prevent a race condition involving an ISR, at least one interrupt signal must be disabled for the duration of the other code's critical section. In the case of a race between RTOS tasks, the best practice is the creation of a mutex specific to that shared object, which each task must acquire before entering the critical section. Note that it is not a good idea to rely on the capabilities of a specific CPU to ensure atomicity, as that only prevents the race condition until a change of compiler or CPU.

Shared data and the random timing of preemption are culprits that cause the race condition. But the error might not always occur, making the tracking of race conditions from observed symptoms to root causes incredibly difficult. It is, therefore, important to be ever-vigilant about protecting all shared objects. Each shared object is an accident waiting to happen.

• **Best practice:** Name all potentially shared objects--including global variables, heap objects, or peripheral registers and pointers to the same--in a way that the risk is immediately obvious to every future reader of the code; the Netrino Embedded C Coding

Standard advocates the use of a "**g_**" prefix for this purpose. Locating all potentially shared objects would be the first step in a code audit for race conditions.

Bug 2: Non-reentrant function
Technically speaking, the problem of a non-reentrant function is a special case of the problem of a race condition. And, for related reasons, the run-time errors caused by a non-reentrant function generally don't occur in a reproducible way--making them just as hard to debug. Unfortunately, a non-reentrant function is also more difficult to spot in a code review than other types of race conditions.

Figure 2 shows a typical scenario. Here the software entities subject to preemption are also RTOS tasks. But rather than manipulating a shared object directly, they do so by way of function call indirection. For example, suppose that Task A calls a sockets-layer protocol function, which calls a TCP-layer protocol function, which calls an IP-layer protocol function, which calls an Ethernet driver. In order for the system to behave reliably, all of these functions must be reentrant.

Libraries and device drivers can harbor non-reentrant functions.

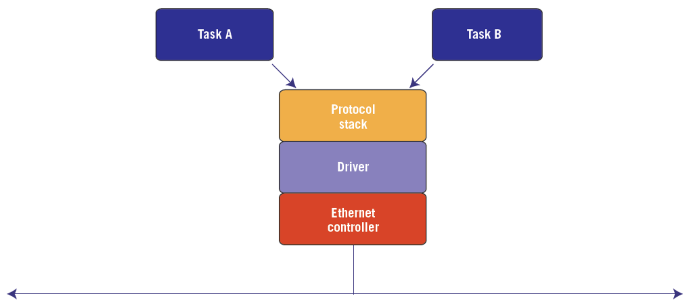


Figure 2

Click on image to enlarge.

But all of the functions of the Ethernet driver manipulate the same global object in the form of the registers of the Ethernet controller chip. If preemption is permitted during these register manipulations, Task B may preempt Task A after Packet A has been queued but before the transmit is begun. Then Task B calls the sockets-layer function, which calls the TCP-layer function, which calls the IP-layer function, which calls the Ethernet driver, which queues and transmits Packet B. When control of the CPU returns to Task A, it requests a transmission. Depending on the design of the Ethernet controller chip, this may either retransmit Packet B or generate an error. Packet A is lost and does not go out onto the network.

In order for the functions of this Ethernet driver to be callable from multiple RTOS tasks near-simultaneously, they must be made reentrant. If they each use only stack variables, there is nothing to do. Thus the most common style for C functions is inherently reentrant. But drivers and some other functions will be non-reentrant unless carefully designed.

The key to making functions reentrant is to suspend preemption around all accesses of peripheral registers, global variables including static local variables, persistent heap objects, and shared memory areas. This can be done either by disabling one or more interrupts or by acquiring and releasing a mutex. The specifics of the problem dictate the best solution.

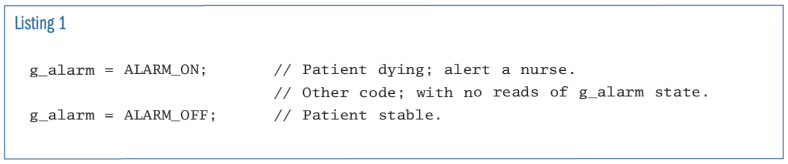
• **Best practice:** Create and hide a mutex within each library or driver module that is not intrinsically reentrant. Make acquisition of this mutex a precondition for the manipulation of any persistent data or shared registers used within the module as a whole. For example, the same mutex may be used to prevent race conditions involving both the Ethernet controller registers and a global or static local packet counter. All functions in the module that access this data, must follow the protocol to acquire the mutex before manipulating these objects.

Beware that non-reentrant functions may come into your code base as part of third-party middleware, legacy code, or device drivers. Disturbingly, non-reentrant functions may even be part of the standard C or C++ library provided with your compiler. If you are using the GNU compiler to build RTOS-based applications, take note that you should be using the reentrant "newlib" standard C library rather than the default.

Bug 3: Missing volatile keyword

Failure to tag certain types of variables with C's `volatile` keyword can cause a number of unexpected behaviors in a system that works properly only when the compiler's optimizer is set to a low level or disabled. The `volatile` qualifier is used during variable declarations, where its purpose is to prevent optimization of the reads and writes of that variable.

For example, if you write code like that in **Listing 1**, the optimizer may try to make your program both faster and smaller by eliminating the first line--to the detriment of the patient's health. If, however, `g_alarm` is declared as `volatile`, then this optimization will not be permitted.



Listing 1

```
g_alarm = ALARM_ON;        // Patient dying; alert a nurse.
                           // Other code; with no reads of g_alarm state.
g_alarm = ALARM_OFF;       // Patient stable.
```

Click on image to enlarge.

• **Best practice:** The `volatile` keyword should be used to declare every:

- Global variable accessed by an ISR and any other part of the code,
- Global variable accessed by two or more RTOS tasks (even when race conditions in those accesses have been prevented),
- Pointer to a memory-mapped peripheral register (or set or registers), and
- Delay loop counter.

Note that in addition to ensuring all reads and writes take place for a given variable, the use of `volatile` also constrains the compiler by adding additional "sequence points." Other volatile accesses above the read or write of a volatile variable must be executed prior to that access.

Bug 4: Stack overflow
Every programmer knows that a stack overflow is a Very Bad Thing. The effect of each stack overflow varies, though. The nature of the damage and the timing of the misbehavior depend entirely on which data or instructions are clobbered and how they are used. Importantly, the length of time between a stack overflow and its negative effects on the system depends on how long it is before the clobbered bits are used.

Unfortunately, stack overflow afflicts embedded systems far more often than it does desktop computers. This is for several reasons, including: (1) embedded systems usually have to get by on a smaller amount of RAM; (2) there is typically no virtual memory to fall back on (because there is no disk); (3) firmware designs based on RTOS tasks utilize multiple stacks (one per task), each of which must be sized sufficiently to ensure against unique worst-case stack depth; and (4) interrupt handlers may try to use those same stacks.

Further complicating this issue, no amount of testing can ensure that a particular stack is sufficiently large. You can test your system under all sorts of loading conditions but you can only test it for so long. A stack overflow that only occurs "once in a blue moon" may not be witnessed by tests that run for only "half a blue moon." Demonstrating that a stack overflow will never occur can, under algorithmic limitations (such as no recursion), be done with a top-down analysis of the control flow of the code. But a top-down analysis will need to be redone every time the code is changed.

• **Best practice:** On startup, paint an unlikely memory pattern throughout the stack(s). (I like to use hex 23 3D 3D 23, which looks like a fence '`#==#`' in an ASCII memory dump.) At runtime, have a supervisor task periodically check that none of the paint above some pre-established high water mark has been changed. If something is found to be amiss with a stack, log the specific error (such as which stack and how high the flood) in nonvolatile memory and do something safe for users of the product (for example, a controlled shut down or reset) before a true overflow can occur. This is a nice additional safety feature to add to the watchdog task.

Bug 5: Heap fragmentation
Dynamic memory allocation is not widely used by embedded software developers--and for good reasons. One of those is the problem of fragmentation of the heap.

All data structures created via C's `malloc()` standard library routine or C++'s `new` keyword live on the heap. The heap is a specific area in RAM of a predetermined maximum size. Initially, each allocation from the heap reduces the amount of remaining "free" space by the same number of bytes. For example, the heap in a particular system might span 10 KB starting from address 0x20200000. An allocation of a pair of 4-KB data structures would leave 2 KB of free space.

The storage for data structures that are no longer needed can be returned to the heap by a call to **`free()`** or use of the **`delete`** keyword. In theory this makes that storage space available for reuse during subsequent allocations. But the order of allocations and deletions is generally at least pseudo-random--leading the heap to become a mess of smaller fragments.

To see how fragmentation can be a problem, consider what would happen if the first of the above 4 KB data structures is free. Now the heap consists of one 4-KB free chunk and another 2-KB free chunk; they are not adjacent and cannot be combined. So our heap is already fragmented. Despite 6 KB of total free space, allocations of more than 4 KB will fail.

Fragmentation is similar to entropy: both increase over time. In a long running system (in other words, most every embedded system ever created), fragmentation may eventually cause some allocation requests to fail. And what then? How should your firmware handle the case of a failed heap allocation request?

• **Best practice:** Avoiding all use of the heap is a sure way of preventing this bug. But if dynamic memory allocation is either necessary or convenient in your system, there is an alternative way of structuring the heap that will prevent fragmentation. The key observation is that the problem is caused by variable sized requests. If all of the requests were of the same size, then any free block is as good as any other-even if it happens not to be adjacent to any of the other free blocks. **Figure 3** shows how the use of multiple "heaps"-each for allocation requests of a specific size-can be implemented as a "memory pool" data structure.

Fixed-sized memory pools don't suffer fragmentation.

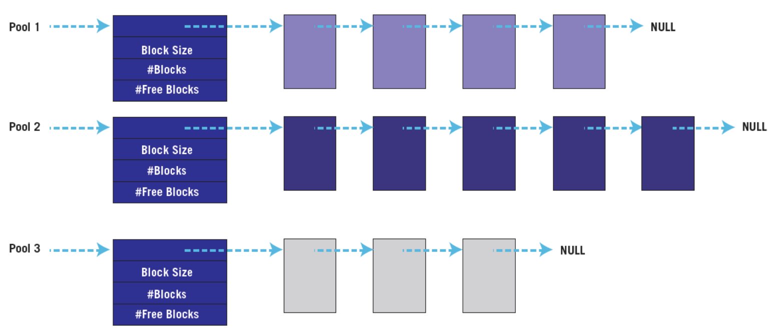


Figure 3

Click on image to enlarge.

Many real-time operating systems feature a fixed-size memory pool API. If you have access to one of those, use it instead of **`malloc()`** and **`free()`**. Or write your own fixed-sized memory pool API. You'll just need three functions: one to create a new pool (of size *M* chunks by *N* bytes); another to allocate one chunk (from a specified pool); and a third to replace **`free()`**.

Code review is still the best practice

You can save yourself a lot of debugging headaches by ensuring that none of these bugs is present in your system in the first place. The best way to do that is to have someone inside or outside your company perform a thorough code review. Coding standard rules that mandate the use of the best practices I've described here should also be helpful. If you suspect you have one of these nasty bugs in existing code, performing a code review may prove faster than trying to trace backward from an observed glitch to the root cause.

In an upcoming column, I'll introduce you to a few other common causes of nasty embedded software bugs.

Michael Barr *is the author of three books and over 50 articles about embedded systems design, as well as a former editor in chief of this magazine. Michael is also a popular speaker at the Embedded Systems Conference and the founder of embedded systems consultancy Netrino. You may reach him at mbarr@netrino.com or read more by him at www.embeddedgurus.net/barr-code.*

Five more top causes of nasty embedded software bugs

Michael Barr

11/2/2010 10:32 PM EDT

What do memory leaks, deadlocks, and priority inversions have in common? They're all Hall of Famers in the pantheon of nasty firmware bugs. **What do memory leaks, deadlocks, and priority inversions have in common? They're all Hall of Famers in the pantheon of nasty firmware bugs.**

Finding and killing latent bugs in embedded software is a difficult business. Heroic efforts and expensive tools are often required to trace backward from an observed crash, hang, or other unplanned run-time behavior to the root cause. In the worst scenario, the root cause damages the code or data in a way that the system still appears to work fine or mostly fine—at least for a while.

In an earlier column ("Five top causes of nasty embedded software bugs," April 2010, p.10, online at www.embedded.com/columns/barrcode/224200699), I covered what I consider to be the top five causes of nasty embedded software bugs. This installment completes the top 10 by presenting five more nasty firmware bugs as well as tips to find, fix, and prevent them.

Bug 6: Memory leak
Eventually, systems that leak even small amounts of memory will run out of free space and subsequently fail in nasty ways. Often legitimate memory areas get overwritten and the failure isn't registered until much later. This happens when, for example, a NULL pointer is returned by a failed call to `malloc()` and the caller blindly proceeds to overwrite the interrupt vector table or some other valuable code or data starting from physical address 0x00000000.

Memory leaks are mostly a problem in systems that use dynamic memory allocation.[1] And memory leaks are memory leaks whether we're talking about an embedded system or a PC program. However, the long-running nature of embedded systems combined with the deadly or spectacular failures that some safety-critical systems may have make this one bug you definitely don't want in your firmware.

Memory leaks are a problem of ownership management. Objects allocated from the heap always have a creator, such as a task that calls `malloc()` and passes the resulting pointer on to another task via message queue or inserts the new buffer into a meta heap object such as a linked list. But does each allocated object have a designated destroyer? Which other task is responsible and how does it know that every other task is finished with the buffer?

Best practice: There is a simple way to avoid memory leaks and that is to clearly define the ownership pattern or lifetime of each type of heap-allocated object. **Figure 1** shows one common ownership pattern involving buffers that are allocated by a producer task (P), sent through a message queue, and later destroyed by a consumer task (C). To the maximum extent possible this and other safe design patterns should be followed in real-time systems that use the heap.[2]

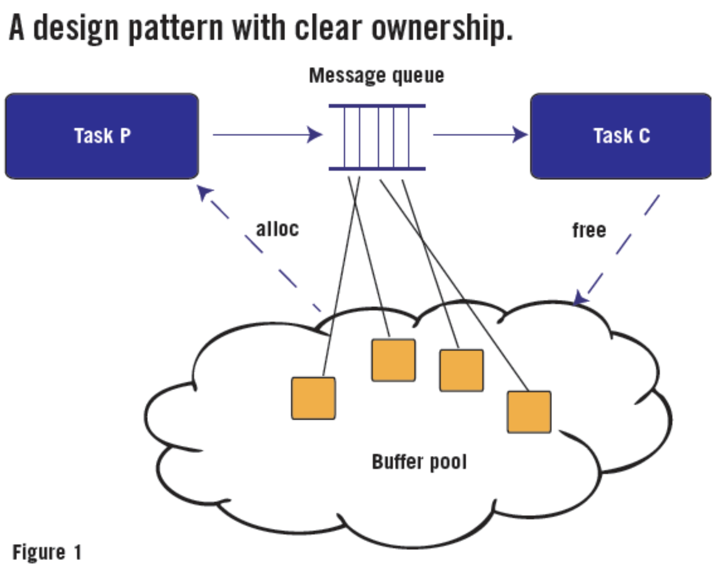


Figure 1

Click on image to enlarge.

Page 2
Bug 7: Deadlock
A deadlock is a circular dependency between two or more tasks. For example, if Task 1 has already acquired A and is blocked waiting for B while Task 2 has previously acquired B and is blocked waiting for A, neither task will awake. Circular dependencies can occur at several levels in the architecture of a multithreaded system (for example, each task is waiting for an event only the other will send) but here I am concerned with the common problem of resource deadlocks involving mutexes.

Best practice: Two simple programming practices are each able to entirely prevent resource deadlocks in embedded systems. The first technique, which I recommend over the other, is to never attempt or require the simultaneous acquisition of two or more mutexes. Holding one mutex while blocking for another mutex turns out to be a necessary condition for deadlock. Holding one mutex is never, by itself, a cause of deadlock.[3]

In my view, the practice of acquiring only one mutex at a time is also consistent with an excellent architectural practice of always pushing the acquisition and release of mutexes into the leaf nodes of your code. The leaf nodes are the device drivers and reentrant libraries. This keeps the mutex acquisition and release code out of the task-level algorithmics and helps to minimize the amount of code inside critical sections.[4]

The second technique is to assign an ordering to all of the mutexes in the system (for example, alphabetical order by mutex handle variable name) and to always acquire multiple mutexes in that same order. This technique will definitely remove all resource deadlocks but comes with an execution-time price. I recommend removing deadlocks this way only when you're dealing with large bodies of legacy code that can't be easily refactored to eliminate the multiple-mutex dependency.

Bug 8: Priority inversion

A wide range of nasty things can go wrong when two or more tasks coordinate their work through, or otherwise share, a singleton resource such as a global data area, heap object, or peripheral's register set. In the first part of this column, I described two of the most common problems in task-sharing scenarios: race conditions and non-reentrant functions. But resource sharing combined with the priority-based preemption found in commercial real-time operating systems can also cause priority inversion, which is equally difficult to reproduce and debug.

The problem of priority inversion stems from the use of an operating system with fixed relative task priorities. In such a system, the programmer must assign each task it's priority. The scheduler inside the RTOS provides a guarantee that the highest-priority task that's ready to run gets the CPU—at all times. To meet this goal, the scheduler may preempt a lower-priority task in mid-execution. But when tasks share resources, events outside the scheduler's control can sometimes prevent the highest-priority ready task from running when it should. When this happens, a critical deadline could be missed, causing the system to fail.

At least three tasks are required for a priority inversion to actually occur: the pair of highest and lowest relative priority must share a resource, say by a mutex, and the third must have a priority between the other two. The scenario is always as shown in **Figure 2**. First, the low-priority task acquires the shared resource (time t_1). After the high priority task preempts low, it next tries but fails to acquire their shared resource (time t_2); control of the CPU returns back to low as high blocks. Finally, the medium priority task—which has no interest at all in the resource shared by low and high—preempts low (time t_3). At this point the priorities are inverted: medium is allowed to use the CPU for as long as it wants, while high waits for low. There could even be multiple medium priority tasks.

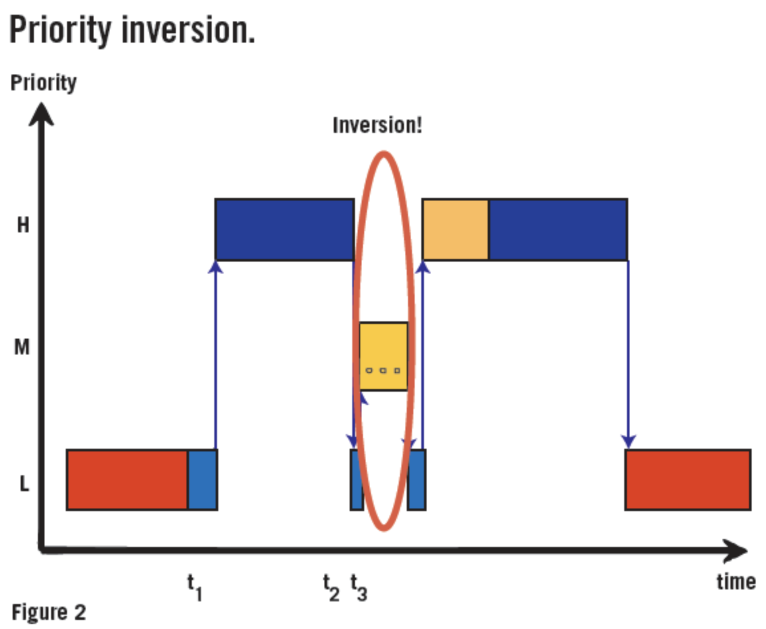


Figure 2

Click on image to enlarge.

Page 3

The risk with priority inversion is that it can prevent the high-priority task in the set from meeting a real-time deadline. The need to meet deadlines often goes hand-in-hand with the choice of a preemptive RTOS. Depending on the end product, this missed deadline outcome might even be deadly for its user!

One of the major challenges with priority inversion is that it's generally not a reproducible problem. First, the three steps need to happen—and in that order. And then the high priority task needs to actually miss a deadline. One or both of these may be rare or hard to reproduce events. Unfortunately, no amount of testing can assure they won't ever happen in the field.[5]

Best practice: The good news is that an easy three-step fix will eliminate all priority inversions from your system.

- Choose an RTOS that includes a priority-inversion work-around in its mutex API. These work-arounds come by various names, such as priority inheritance protocol and priority ceiling emulation. Ask your sales rep for details.
- Only use the mutex API (never the semaphore API, which lacks this work-around) to protect shared resources within real-time software.
- Take the additional execution time cost of the work-around into account when performing the analysis to prove that all deadlines will always be met. Note that the method for doing this varies by the specific work-around.

Note that it's safe to ignore the possibility of priority inversions if you don't have any tasks with consequences for missing deadlines.

Bug 9: Incorrect priority assignment
Get your priorities straight! Or suffer the consequence of missed deadlines. Of course, I'm talking here about the relative priorities of your real-time tasks and interrupt service routines. In my travels around the embedded design community, I've learned that most real-time systems are designed with ad hoc priorities.

Unfortunately, mis-prioritized systems often "appear" to work fine without discernibly missing critical deadlines in testing. The worst-case workload may have never yet happened in the field or there is sufficient CPU to accidentally succeed despite the lack of proper planning. This has lead to a generation of embedded software developers being unaware of the proper technique. There is simply too little feedback from non-reproducible deadline misses in the field to the original design team—unless a death and a lawsuit forces an investigation.

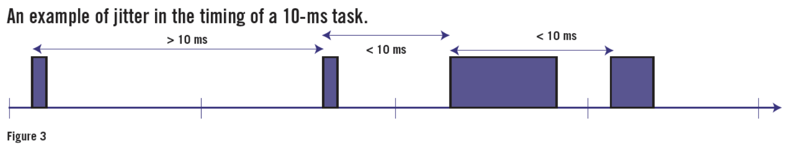
Best practice: There is a science to the process of assigning relative priorities. That science is associated with the "rate monotonic algorithm," which provides a formulaic way to assign task priorities based on facts. It is also associated with the "rate monotonic analysis," which helps you prove that your correctly-prioritized tasks and ISRs will find sufficient available CPU bandwidth between them during extreme busy workloads called "transient overload." It's too bad most engineers don't know how to use these tools.

There's insufficient space in this column for me to explain why and how RMA works. But I've written on these topics before and recommend you start with "Introduction to Rate-Monotonic Scheduling"[6] and then read my column "3 Things Every Programmer Should Know About RMA."[7]

Please know that if you don't use RMA to prioritize your tasks and ISRs (as a set), there's only one entity with any guarantees: the one highest-priority task or ISR can take the CPU for itself at any busy time—barring priority inversions!—and thus has up to 100% of the CPU bandwidth available to it. Also note that there is no rule of thumb about what percentage of the CPU bandwidth you may safely use between a set of two or more runnables unless you do follow the RMA scheme.

Page 4
Bug 10: Jitter
Some real-time systems demand not only that a set of deadlines be always met but also that additional timing constraints be observed in the process. Such as managing jitter.

An example of jitter is shown in **Figure 3**. Here a variable amount of work (blue boxes) must be completed before every 10 ms deadline. As illustrated in the figure, the deadlines are all met. However, there is considerable timing variation from one run of this job to the next. This jitter is unacceptable in some systems, which should either start or end their 10 ms runs more precisely.

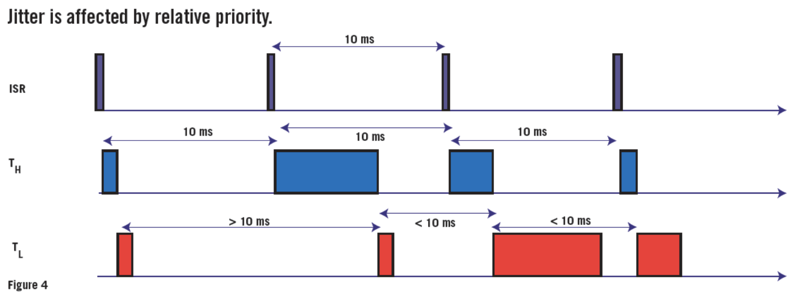


Click on image to enlarge.

If the work to be performed involves sampling a physical input signal, such as reading an analog-to-digital converter, it will often be the case that a precise sampling period will lead to higher accuracy in derived values. For example, variations in the inter-sample time of an optical encoder's pulse count will lower the precision of the velocity of an attached rotation shaft.

Best practice: The most important single factor in the amount of jitter is the relative priority of the task or ISR that implements the recurrent behavior. The higher the priority the lower the jitter. The periodic reads of those encoder pulse counts should thus typically be in a timer tick ISR rather than in an RTOS task.

Figure 4 shows how the interval of three different 10 ms recurring samples might be impacted by their relative priorities. At the highest priority is a timer tick ISR, which executes precisely on the 10 ms interval. (Unless there are higher priority interrupts, of course.) Below that is a high-priority task (T_H), which may still be able to meet a recurring 10-ms start time precisely. At the bottom, though, is a low priority task (T_L) that has its timing greatly affected by what goes on at higher priority levels. As shown, the interval for the low priority task is 10 ms +/- approximately 5 ms.



Click on image to enlarge.

Page 5

Hire an exterminator
As with any bug that's difficult to reproduce, your focus should be on keeping all five of these nasty bugs out of your system before they get in. For the particular bugs in this installment, the single best way to do that is to have someone inside or outside your company perform a thorough independent high-level review of the firmware architecture, looking especially at task and ISR interactions and relative priorities. Of course, coding standards and coding reviews are also helpful in picking up on some of these issues—as they were especially for the top five.[8]

Michael Barr *is the author of three books and over 50 articles about embedded systems design, as well as a former editor in chief of this magazine. Michael is also a popular speaker at the Embedded Systems Conference and the founder of embedded systems consultancy Netrino. You may reach him at mbarr@netrino.com or read more by him at www.embeddedgurus.net/barr-code.*

Endnotes:

1. Unlike fragmentation (see http://embeddedgurus.com/barr-code/2010/03/firmware-specific-bug-5-heap-fragmentation/), memory leaks can happen even with fixed-size allocators.
2. In addition to avoiding memory leaks, the design pattern shown in Figure 1 can be used to ensure against "out-of-memory" errors, in which there are no buffers available in the buffer pool when the producer task attempts an allocation. The technique is to (1) create a dedicated buffer pool for that type of allocation, say a buffer pool of 17-byte buffers; (2) use queuing theory to appropriately size the message queue, which ensures against a full queue; and (3) size the buffer pool so there is initially one free buffer for each consumer, each producer, plus each slot in the message queue.
3. In theory, the task that wants the mutex could starve while a series of higher priority tasks take turns with the mutex. However, the rate monotonic analysis can be used to ensure this doesn't happen to tasks with deadlines that must be met.
4. An additional benefit of this architectural pattern is that it reduces the number of programmers on the team who must remember to use and correctly use each mutex. Other benefits are that each mutex handle can be hidden inside the leaf node that uses it and that doing this allows for easier switches between interrupt disables and mutex acquisition as appropriate to balance performance and task prioritization. One of the most famous priority inversions happened on Mars in 1997. Glitches were observed in Earth-based testing that could not be reproduced and were not attributed to priority inversion until after the problems on Mars forced investigation. For more details, read Glenn Reave's "What really happened on Mars?" account (http://catless.ncl.ac.uk/Risks/19.54.html#subj6).
5. Barr, Michael and Dave Stewart. "Introduction to Rate Monotonic Scheduling," Beginner's Corner, *Embedded Systems Programming*, February 2002. Available online at www.embedded.com/showArticle.jhtml?articleID=9900522.
6. Barr, Michael. "Three-Things-Every-Programmer-Should-Know-About-RMA," Barr Code, Embedded.com, available at www.eetimes.com/discussion/other/4206206/Three-Things-Every-Programmer-Should-Know-About-RMA.
7. Barr, Michael. "Five top causes of nasty embedded software bugs," *Embedded Systems Design*, April 2010, p.10, available online at www.embedded.com/columns/barrcode/224200699.