by Bob Japenga (USA)

Concurrency in Embedded Systems (Part 1)

Embedded Systems and Some of Their Pitfalls

This is the first in a multi-part article series dealing with concurrency in embedded systems. This article defines concurrency in embedded systems, discusses some pitfalls, and examines one of them in detail.

I work for a small company that does embedded systems and software design. When we started in the late 1980s, we had no reputation outside our previous industries. In order to get work, it was important for us to establish our credibility and to demonstrate our expertise. Here's how we did that during those early years.

After we got through the first level of vetting with a new company (we weren't criminals and at least we talked a good story), our clients would ask us for a proposal to either modify their existing code or to create new code based on the functionality of their old code. After the exchange of a nondisclosure agreement, they would send us their source code to enable us to quote the job. During that process, I would immediately look at their serial drivers because I know this is an area that is usually prone to errors. Most serial drivers are interrupt-driven and thus require knowledge of how to design concurrent threads in embedded systems. In our own experience and in the experience of others, we had seen that there were many possible pitfalls. If we could *[Editor word missing – help!]* our clients solve an existing problem quickly and for no charge, I thought this would both establish our credibility and demonstrate our expertise. So I would briefly look in their code for some of the common flaws we had both created and seen throughout years of dealing with interrupt-driven serial drivers. In almost 50% of the cases, we were able to find a real problem in their code—and it was always caused by concurrency issues. Sometimes they were serious and sometimes they were relatively harmless.

Getting concurrency right is not easy. Over the next several articles, I will address the issue of concurrency as it relates to embedded systems. In this article, I define concurrency, list some of the common pitfalls, and look at one of them in particular. I will address other common pitfalls in upcoming articles.

DEFINING CONCURRENCY

Concurrency takes place any time two or more activities can happen in the same time segment. For example, I can concurrently wash the dishes, watch TV, and listen for a text message on my phone. A phone can concurrently listen for incoming calls while displaying a TV program.

According to Wikipedia, in computer science, concurrency is a property of systems in which several computations are executing simultaneously, and potentially interacting with each

HELPFUL DEFINITIONS

Thread: Any functionality that can be active during the same time segment. It can be in hardware or software.
Blocked: When a software thread relinquishes control of the processor to the operating system.
Preempt: When one thread interrupts another.

other.

Of course, if they don't interact, there are no pitfalls, but they are still concurrent. However, they almost always interact in some way and that is what creates the problems.

COMMON CONCURRENCY PITFALLS

Any embedded system that has concurrency built into its design can experience one or more of the following problems: race conditions, where the order of execution affects the outcome of a given result; corrupting of shared resources, where shared resources are used by two or more concurrent threads; logical complexity creating less than airtight algorithms—the more things that can affect each other, the more difficult it is to carefully think your design through and make it bullet proof; deadlock, where functions stop working for no apparent reason; time-of-check-to-time-of-use (TOCTTOU)—all embedded software that monitors the real world has to deal with this kind of concurrency—I read the data from an input and make a decision, but by the time I use the data, the data has changed; and priority inversion.

I will look at these conditions over the next several articles. In this article, I address priority inversion: what it is and how to avoid it.

WHAT IS PRIORITY INVERSION?

Given the number of problems concurrency has given me over the years, I wondered how concurrency issues ranked in the history of major software failures. As I looked over various lists of "the worst software bugs" I found that only a few of them directly involved these common pitfalls. The Mars Pathfinder bug was one of them. Initially flawless in its fulfillment of a very complicated mission, early on, the Pathfinder began experiencing periodic resets and the subsequent loss of data. The cause was a classic case of priority inversion.

To illustrate concurrency and how priority inversion can happen in an embedded system, I'll create a very simple embedded system with one input, one output, a very fast logging device, and a graphical user interface (GUI). You can create such a system with or without concurrency. You could, for example create software with a single loop as shown in Table 1.

Our Example Design Without Concurrency in the Design
Check to see if the user has hit a key
Output the new display as a result of the key press (takes about 100 ms)
Check to see if something changed in the external world
Process the changes from the input from external world and set the output accordingly
Log the user input and the output state to our logging device (takes about 50 ms)
Rinse and repeat

Table 1—Steps that single-loop software may take in a design without concurrency built into it

We have seen literally scores of systems designed like this. Simple. They work. No concurrency is necessary in the software. But wait a minute. There is concurrency in this system. The user and the external device can change things at the same time. That means the system has concurrency. But does the software need concurrent threads to handle this? Well, that depends. If the input changes more quickly than our time to execute our simple loop, we have a problem. If it takes 100 ms to write to our graphics display and our external input device can do something every 20 ms, we would miss critical data while going through the loop.

So how do we solve that? We could start sprinkling I/O reads into the middle of the graphics library, but that is extremely messy. We could create two concurrent threads in our system: one to process the user interface (UI) and one to process the external device. In an embedded system this could be accomplished in three ways: with a multitasking operating system (OS), with interrupts, or with hardware. In a multitasking OS you would put a blocking delay at the end of the high-speed thread. In addition, you would enable the external thread to preempt the UI loop. This means the external device thread does not have to wait until the UI thread is complete before it can start. This could be done by giving the external device thread a higher priority than the user input thread. For example, if you are polling the external input every 10 ms, you would be "interrupting" the UI loop some place in the code every 10 ms. With interrupts on the external device, you would basically be doing the same thing without the need for a thread delay. You would be "event-driven" rather than periodically polling. But you would still be interrupting the UI thread every time the external device changed. Finally, you could solve this by using or creating hardware that could obtain the external data while the UI software is busy and go back to your single-thread solution. Table 2 provides a look at solving it using the multithreaded OS. Notice that the UI thread never stops but polls continuously (sometimes this is called the idle thread). The external device thread blocks (gives other threads the opportunity to run) and thus effectively polls the external input every 10 ms (plus our processing time).

Our Example Design with Two Threads	
User Input Thread	**External Device Thread**
Check for user input	Check to see if something changed in the external world (input changed state or character received)
Process the user input	Process the changes from the input from external world and set the output accordingly
Update the display	Log the external data to our logging device when free
Log the user data to our logging device when free	Block for 10 ms
Rinse and repeat	Rinse and repeat

Table 2—Steps that a user input thread and an external device thread may take in a system design with two threads

Our Example Design with Three Threads		
User Thread (Low Priority)	**Network Thread (Medium Priority)**	**External Device Thread (High Priority)**
Check for user input	Check to see if there is data to be sent	Check to see if something changed in the external world (input changed state or character received)
Process the user input	Send the data over the Internet to the host	Process the changes from the input from external world and set the output accordingly
Update the display	Wait for the response for up to 100 ms	Log the external data to our logging device when free
Log the user data to our logging device when free	Block for 5 s	Block for 10 ms
Rinse and repeat	Rinse and repeat	Rinse and repeat

Table 3—Steps that a user input thread, a network thread, and an external device thread may take in a system design with three threads

We now have a complexity that both threads share the same logging device and cannot use it at the same time. So they would need to wait for the other to complete before use. But that is no problem since it is a fast device and thus neither thread will be unable to perform if it has to wait even the maximum amount of time (50 ms).

We deliver our masterpiece to marketing and they are happy with one exception. They want to make this an Internet appliance. They want to send data periodically to a web server and to receive responses back from the web server. We will need to check to see if we have data every 5 s. Now we have to add a third concurrent thread. This will be a medium-priority thread that sends the data to the Internet. It needs to respond more quickly than the UI but more slowly than the external data interface (see Table 3).

We have now set up the classic case for priority inversion to occur and we will occasionally miss critical data (albeit rarely). Here is how it happens. The low-priority UI thread is writing to the log device when the high-priority device wants to write to it. So the high-priority thread blocks waiting for the device to become free. During that tiny window, the medium-priority thread preemptively interrupts the low-priority thread and runs for 100 ms thus blocking the high-priority thread from recording a critical transition of its input. Because it happens so seldom, this would probably never be found during testing. Only after we deliver the first 10,000 units to the field would this be reported.

PREVENTING PRIORITY INVERSION

There are two primary ways of preventing this from happening. The first involves locking out preemption during critical sections of code. This can be done by locking out interrupts or through some OS-locking mechanism. In our example, we would place these locks in the low-priority thread around the use of the shared resource: the logger. When the low-priority thread logged our data, it would prevent anyone from preempting the low-priority thread until complete. This would prevent a higher-priority thread from preempting a lower-priority thread when it is dealing with a shared resource.

The second and less complex (for the designer) method is to let the OS prevent this from happening by using either priority inheritance or the priority ceiling techniques with shared resources. This would mean that any time a shared resource is used by any thread, that thread would take either the highest priority of the threads using the shared resource (priority inheritance) or the priority assigned to the resource (priority ceiling). There are problems with these features affecting latency, but they are beyond the scope of this article.

Any POSIX-compatible OS (including QNX, VxWorks, and Linux) has priority inheritance built in and it is well worth any real-time system designer's time to become very familiar with how this is used in their OS.

MANAGING CONCURRENCY REQUIREMENTS

Most systems we design have concurrency requirements in them. It is our job as system designers to identify them and then create designs that robustly handle this concurrency—at times adding concurrency into our software. We need to be aware of the dangers of both implementing concurrency and the dangers that the solutions can cause as well. Next time I will look at some more pitfalls of concurrency in embedded systems.

Bob Japenga has been designing embedded systems since 1973. In 1988, along with his best friend, he started MicroTools, which specializes in creating a variety of real-time embedded systems. With a combined embedded systems experience base of more than 200 years, they love to tackle impossible problems together. Bob has been awarded 11 patents in many areas of embedded systems and motion control. You can reach him at rjapenga@microtoolsinc.com.

RESOURCES

Computerworld, Inc., "Epic Failures: 11 Infamous Software Bugs," 2010, www.computerworld.com/s/article/9183580/Epic_failures_11_infamous_software_bugs?taxonomyId=18&pageNumber=1.

N. Dershowitz, The Blavatnik School of Compute Science, Tel Aviv University, "Software Horror Stories," www.cs.tau.ac.il/~nachumd/verify/horror.html.

S. Garfinkel, *Wired* Magazine, "History's Worst Software Bugs," 2005, www.wired.com/software/coolapps/news/2005/11/69355?currentPage=all.

M. Jones, Microsoft Corp., Microsoft Research, "What Really Happened on Mars?" 1997, http://research.microsoft.com/en-us/um/people/mbj/mars_pathfinder/mars_pathfinder.html.