by Bob Japenga (USA)

Concurrency in Embedded Systems (Part 3)

Avoiding Concurrency Problems

The first two parts of this article series introduced concurrency in embedded systems and discussed two common concurrency design problems. The third part of this series examines generic ways to avoid some of these problems.

The introduction to this article series discussed some common pitfalls when using concurrency. The last article examined some case studies of specific pitfalls. This month, I want to step back and look a concurrency from a higher level, including some generic ways to help design robust systems with concurrency.

CONCURRENCY WITHOUT PREEMPTION

Concurrency is often used to remove the complexity from a design with many unique and independent functions. For example, consider a simple device with a control function, a data storage function, a networking function and a user interface (UI) with a keyboard and a touch screen. Conceptually, it may be easier to break these four functions into six separate concurrent threads. The UI would have three threads: one for the keyboard, one for the touch screen, and one to manage the user inputs to create the display. Additional "threads" may exist in the interrupt routines. Once broken up in these concurrent threads, the designer can think through each function separately and seamlessly intersperse necessary waits into the function. Breaking these up into separate concurrent threads provides other significant advantages. There is less coupling between functions, it is easier to delegate the tasks to separate people, and it is easier to debug.

If the threads are designed so that no thread runs for more than 10 to 20 ms and there are no high-speed and/or hard real-time requirements, preemption (one thread interrupting another) may be unnecessary, except in the interrupt service routines (which are inherently preemptive). Ideally, you may be able to eliminate custom interrupt routines, which can simplify everything. This can also simplify resource sharing by eliminating the need to coordinate reading and writing to these shared resources. It prevents the possibility of dead lock and priority inversion. It eliminates the need for semaphores, mutexs, and other locking mechanisms, which can cause more problems than they solve. There is no need to worry about file sharing. Without preemption, you don't need to worry about atomic operations or which library functions are thread safe.

Identifying what libraries are thread safe is no trivial task. One real-time operating system (RTOS) manufacturer that touts its reliability has almost 20 pages of documentation on which functions are completely thread safe and which are partially thread safe. It includes special cases of partially thread-safe functions, and functions that are thread safe except for one shared variable. The POSIX standard lists about 100 standard functions that are not thread safe.

CONCURRENCY WITH LIMITED PREEMPTION

Late in a major new product's development cycle, we began noticing some errors that were intermittent and difficult to duplicate. After significant pain and effort, we found a particular thread-safe library function was not thread safe. Our solution proved quite simple. We set all threads' priori-

ty the same except for one that absolutely had to be preemptive. We ensured this thread did not use our "not so safe" thread-safe library. The fix was quick and painless.

The other obvious way to use limited preemption is to only include preemption in your interrupt routines. Then you only need to worry about the concurrency pitfalls in routines with less resource sharing.

CONCURRENCY ON ACID

An old database paradigm called ACID—which stands for atomicity, consistency, isolation, and durability—is applicable to designing embedded systems with concurrency. Real-time multiuser databases have a lot of the same problems that we have when designing embedded systems. Imagine if you and your spouse went to different ATMs to remove funds from your checking account. You would have concurrency and you would have problems if the concurrency was not properly handled at the design stage. Ensuring that all your embedded design aspects have this paradigm's properties will help ensure safe and reliable operation.

ACID was developed in the late 1970s as a means of evaluating the reliability of distributed databases. The next sections will example each of ACID's elements.

ATOMICITY

The last article provided a detailed discussion on atomicity. Let's step back and look at it from 5,000'. A good way to describe atomicity from a higher level is to say: An atomic operation is one that can assure us that in all cases (including power failures, system resets, and all errors), the operation will complete as desired or have no effect on the system.

In the last article, we did not address the issue of an operation being atomic across power outages or system crashes. A power outage or a system crash is a much overlooked form of concurrency. Of course, it is only a problem if you are operating on a nonvolatile resource. I am always amazed how often we don't take this into account in our designs. Imagine the simple case of writing an important parameter in its primary location and its back up. You carefully lock out other tasks from touching the parameter (e.g., change the primary, change the back up, and then unlock the access). You think you have created an atomic operation and prevented normal concurrency from affecting your design. But have you covered the case where the system powers down or crashes between writes? You now have inconsistent data on your system. When you power up, which one is correct, the primary or the back up? It is not enough to make the operation atomic under normal operation. You must make it atomic across system crashes or power outages.

One solution is to provide some sort of nonvolatile "dirty" indicator to each element. The steps are to set the dirty flag for the primary element, write the primary, clear the dirty flag for the primary element, write the dirty flag for the back-up element and write the back-up element, and clear the dirty flag for the backup. At startup, if the dirty flag is set for one, you know you cannot trust the data so you take the other.

An approach I have found useful in my designs is to ask myself every time I modify something that is nonvolatile: What happens to the system if this operation does not fully complete? Have I provided a mechanism to handle an incomplete operation? Finally, since I don't really trust my own ability to do this perfectly, all of our systems must be tested under tens of thousands of power cycles (e.g., simulating crashes or asynchronous power outages) in an attempt to verify we have done this correctly.

CONSISTENCY

An embedded system that exhibits consistency cannot end up in an inconsistent state. We addressed consistency in our last article when we discussed writing multiple-byte data structures in an environment without atomicity. But consistency goes beyond temporary data. Many of the systems we design as embedded engineers must run 24/7 and endure hundreds of power fails in their lifetime (and unfortunately a few crashes). As we discussed above, power outages and crashes are a much overlooked kind of concurrency. Ensuring the system does not find itself in an inconsistent state is no small task. I have often been involved in designs where these inconsistent states get discovered in testing (and usually late in testing).

Here are some things that have helped me avoid inconsistent states: Identify all of the states I know could be considered inconsistent. Repeatedly ask yourself questions (e.g., Have I handled this case?) Put the checks in startup for inconsistent states. Even if I think there isn't a chance of it getting into that state, why not put the code in at startup to correct the case? Check for inconsistent states during normal operation. Concurrent operations could also create such a state. Do you have code in it that can handle these inconsistent states? The key to the definition is "end up in inconsistent states." Your system may get into this state temporarily—but make sure at startup and during operation you have provided for a mechanism to restore it to a consistent state.

ISOLATION

Sometimes I think the acronym creators needed an "I" for their acronym when they chose the word "isolation." I prefer to think of this principle as making your embedded design order of operation independent insofar as the operations are order independent. With concurrency, we can often create cases where operations are interleaved. And variable timing can create race conditions. Race conditions in software have killed people (e.g., the famous Therac-25 failures[1]).

How can we prevent race conditions and create order independent operations? In M. Barr's, "Firmware-Specific Bug #1: Race Condition," (Embedded Gurus, 2010) a respondent to the blog wrote, "Without sharing resources, race conditions won't happen." That is just not true. Race conditions can happen in single-threaded software systems that have hardware with a mind of its own. As I stated in the first article, concurrency can happen with the hardware being the concurrent thread. So, eliminating shared resources can reduce, but not eliminate, the probability of a race condition.

Here are some of my basic guidelines to prevent concurrency based order of operation based issues: Identification. Start by identifying operations that must be ordered. Look at your design and ask: Does this depend on timing from an external resource (HW or SW)? This is nontrivial. I didn't see most race

conditions I created until it was too late. Once identified, start asking yourself: What if this device responds much more slowly? What if this gets interrupted? What resource (HW or SW) under me can change during this operation? Can I design this in an order-independent manner?

Preallocation. If possible, allocate or reserve all the resources necessary for an operation before you start the operation.

Identify the ordering. Use timestamps or sequence numbers in operations where appropriate. My maxim is, "Use of sequence numbers or timestamps covers a multitude of evils." This is especially true in networking. With load-balancing servers and powerful multithreading in the devices you are talking to, you can be surprised at the order your data comes back to you.

Protect against preemption. Any code that must require specific ordering (again avoidance is the best medicine) that deals with external shared resources should be protected to guarantee this specific ordering.

Test, test, and test some more. Purposely shake up the testing order (i.e., stress test the identified sections). For example, if you can significantly add or reduce delays and response times of the hardware or software that is being shared you may uncover race conditions you have missed. I highly recommend testing on as many different hardware platforms as possible. I have another maxim: "For every order of magnitude increase in production, you discover another problem in your perfect system."

DURABILITY

Durability means that once an action is taken it is taken. Sounds like a Yogism. When using an operating system or a file system, remember there is concurrency you don't even know about. We had an issue with a JFFS2 system on Linux where we would write a variable to a flash file system file prior to rebooting. We did this to let the system know the state of the system from the last boot when it powered up. We wrote the variable, flushed the cache, synched the file system (fsync), and the variable was still not there occasionally when we came up. Once the action was taken, it was not really taken. We then discovered that we needed to fsync the directory file as well. This closed a few more holes. Finally, one more note in the API said: "If the underlying hard disk has write caching enabled, then the data may not really be on permanent storage when fsync() returns." Oh great! So much for durability. Our solution was to dismount the file system and remount it read only. Finally, the underlying file system cooperated. The moral of the story: Not all actions taken are taken!

KNOWLEDGE = POWER

Most systems have concurrency in them. You must understand it in order to plan for it. The more concurrency you build into the system, the more complexity you are bringing to the table. Rigorous and thoughtful designs are hard to achieve but well worth the effort.

Next time I will examine building concurrency into an embedded Linux system.

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.

REFERENCE

[1] Wikipedia, Therac-25, http://en.wikipedia.org/wiki/Therac-25.

RESOURCES

B. Japenga, "Concurrency in Embedded Systems (Part 1): An Introduction to Concurrency and Common Pitfalls," *Circuit Cellar* 263, 2012.

———, "Concurrency in Embedded Systems (Part 2): Atomicity and TOCTTOU," *Circuit Cellar* 265, 2012.

M. Barr, "Firmware-Specific Bug #1: Race Condition," Embedded Gurus, 2010.