by Bob Japenga (USA)

# Concurrency in Embedded Systems (Part 2)

## Concurrency, Atomicity, and TOCTTOU

The first part of this article series introduced the topic of concurrency in embedded systems. This article discusses two of the most common problems in embedded system designs containing concurrency: using nonatomic operation in concurrent threads and time-of-check-to-time-of-use (TOCTTOU).

In the first part of this article series, ("Concurrency in Embedded Systems Part 1: An Introduction to Concurrency and Common Pitfalls", Circuit Cellar 263, 2012), we defined concurrency as a common feature found in embedded systems. We saw that concurrency takes place any time two or more activities can happen in the same time segment. We saw that these concurrent operations only cause problems when they interact with each other. We looked at some of the common pitfalls that happen with such systems and discussed, in thin slices, how priority inversion can happen and what you can do to prevent it.

As I sat down to write this article, I was thinking about the other common pitfalls I mentioned last time and asked myself: "Which one do I want to talk about this month?" I decided to discuss how nonatomic operations wreak havoc in our embedded systems and how to prevent the chaotic situation. I'll also discuss a somewhat related issue: time-of-check-to-time-of-use (TOCTTOU).

### DEFINITIONS

Let's start with some definitions. An atomic operation is any operation that cannot be interrupted by another operation. In a microprocessor, most single-assembly instructions cannot be interrupted until they complete their operation. If you design embedded systems, you should know which operations you use are atomic whenever you are operating on a shared resource. Using nonatomic operations on shared resources leads to the generation and use of noncoherent data in our systems.

### CASE STUDY #1: SHARED MEMORY

After a brief amount of preplanning for this article, I was called into a meeting to discuss a problem we were having during the alpha testing of a new product. The problem manifested itself in one of our embedded systems by reporting bad data to its host. The system was measuring energy usage in kilowatt hours and reporting this data to a remote host. Our interface contained a Maxim Integrated Products 71M6533 Teridian energy-metering IC that we talked to over a SPI bus. This is a powerful energy-monitoring chip that can be used to measure power in three phase electrical systems. This chip was programmed by another vendor to provide a memory interface we accessed over SPI. Both subsystems access this shared

### HELPFUL DEFINITIONS

**Atomic operation:** An operation that is noninterruptible by any other operation and never presents partial results to an outside observer

**Alpha testing:** Testing performed by an independent team on a system installed at a place other than the targeted customer's site

**Shared resource:** Any provider of information that is read or written to by separate threads

**KYZ pulse:** In a mechanical electrical meter, a pulse that changes state every half rotation of the meter's disk and represents a quanta of energy

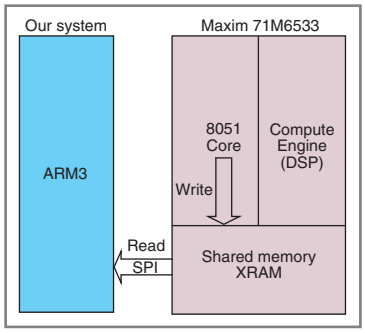


**Figure 1**—Our system used a SPI bus to talk to a Maxim Integrated Products 71M6533 Teridian energy-metering IC. Both systems access this shared memory interface at a 1-Hz rate.

memory interface at a 1-Hz rate (see Figure 1). After some analysis, we concluded that the problem could be caused by a classic nonatomic concurrent operation. Let me explain by oversimplifying the details while maintaining the essence of the problem.

We read cumulative energy usage created by the Maxim energy chip with a 4-byte read of a shared memory region over SPI. An 8051 core inside the energy chip writes to this 4-byte region. Since both our 4-byte read and the 8051's 4-byte write were interruptible (nonatomic), at times we could get incoherent data. An example is shown in Table 1.

The end result is that, in a sequence of reads, we would obtain energy readings of:

255
512
256

The error compounds if the collision occurs on the rollover of the second, third, or fourth byte. Of course those occur far less frequently than my example. This doesn't happen often, but with more than 2.5 million s in a month and more than 100,000 m in the field, it only has to happen once in 250 billion reads to happen once a month.

## CASE STUDY #2: INCREMENTING A COUNTER

Many years ago, we ported some code from a customer's energy-monitoring system to a new platform they were developing. The system consisted of a single loop with one timer interrupt routine. Thus there were two concurrent threads. The most critical piece of information was the total energy read from each of four separate channels. In the timer thread, four KYZ pulses indicative of energy were polled and incremented in a 32-bit word each time a low to high transition was made. The software was written in C. The processor was an 8051 derivative.

The timer thread would have code like what is shown in Listing 1.

The background thread would send the `KYZ_counter` to the host, display them on the screen or log them to a log file.

The problem was that occasionally the `KYZ_counter` data would be wrong when sent to the host on the display or in the log. How could this happen? This happens because incrementing the `KYZ_counter_1` (as well as counters 2–4) was not an atomic operation even though it was a single C instruction. Underneath this one C instruction would be a half a dozen 8051 instructions—any of which could be interrupted. As we saw previously, the problem will occur anytime one of the 8-bit portions of the 32-bit number rolls over from 0xFF to 0x00 after it has interrupted a nonatomic read of the data. If the same code was ported to a 32-bit ARM9 processor, which has atomic 32-bit operations, there would be no problem.

## CASE STUDY #3: READING A 16-BIT COUNTER ON AN 8-BIT PROCESSOR

I first ran into this problem while working with a 16-bit counter in a single-threaded program running on an 8-bit processor. As we saw in Part 1 of this article series, sometimes the concurrency occurs in hardware as we will see in this case.

The design had a 16-bit counter used to count bottles going down a conveyor. The counter was read by an 8-bit processor in two operations. The counter updated in nanoseconds and the processor read the counter in two 8-bit reads about 1 µs apart. Hopefully you see the scenario coming. The counter is sitting at 0x1FF when the processor reads the low-order byte (0xFF). By the time the processor has read the high-order byte, the counter has incremented to 0x0200. Thus the processor sees 0x02 and sets the counter to 0x2FF rather than either of the two correct results: 0x200 or 0x1FF. You just processed 255 extra bottles.

## WHAT ARE SOME SOLUTIONS?

There are four basic solutions to avoid these kinds of pitfalls: Don't share memory unless absolutely necessary. Remember, concurrency problems only manifest when we share resources.

| 8051/Maxim Operation | Our SPI Read | Byte | Memory Location contents |
|---|---|---|---|
| 8051 writes byte 1 of 0x000000FF | | 0x00 | 0x00000000FE |
| 8051 writes byte 2 of 0x000000FF | | 0x00 | 0x00000000FE |
| 8051 writes byte 3 of 0x000000FF | | 0x00 | 0x00000000FE |
| 8051 writes byte 4 of 0x000000FF | | 0xFF | 0x00000000FF |
| 1 s later, the 8051 writes byte 1 of 0x00000100 | | 0x00 | 0x00000000FF |
| 8051 writes byte 2 of 0x00000100 | | 0x00 | 0x00000000FF |
| 8051 writes byte 3 of 0x00000100 | | 0x01 | 0x00000001FF |
| | Our system reads byte 1 | 0x00 | 0x00000001FF |
| | Our system reads byte 2 | 0x00 | 0x00000001FF |
| | Our system reads byte 3 | 0x01 | 0x00000001FF |
| | Our system reads byte 4 | 0xFF | 0x00000001FF |
| 8051 writes byte 4 of 0x00000100 | | 0x00 | 0x0000000100 |
| | 1 s later, our system reads byte 1 | 0x00 | 0x0000000100 |
| | Our system reads byte 2 | 0x00 | 0x0000000100 |
| | Our system reads byte 3 | 0x01 | 0x0000000100 |
| | Our system reads byte 4 | 0x00 | 0x0000000100 |

**Table 1**—Our 4-byte read and the 8051's 4-byte read. At times, both reads were nonatomic, resulting in incoherent data.

**Listing 1—**Time thread code

```
if (KYZ_1 != KYZ_1_last_value)
{
            KYZ_counter_1++;
            KYZ_1_last_value = KYZ_1;
}
```

Make all operations on shared resources atomic (noninterruptible) by using some kind of locking mechanism. For example, in Case Study #2, you would want to lock out any reads while the data was being read or written (perhaps by disabling interrupts). Or in our simple single-loop system, you could create a snapshot of the data at the top of the loop with interrupts disabled and then freely use this coherent snapshot of the data throughout the loop. In Case Studies #1 and #3, that was impossible because the 8051 code was written by a vendor.

Build an indication of the coherency of the data into the structure. Using the hardware model, we could provide a busy bit on the data structures indicating that the data structure is being changed. Before changing the data, set the busy bit. After writing the data, clear the busy bit. It is also good practice to define the maximum amount of time the busy bit can remain set. Again, we didn't control the 8051 in Case Study #1. As noted in the next section, we will see how this too must be carefully used. In Case Study #2, we could create a data structure with a header, the KYZ counter and a footer sequence number.

Use multiple reads. If the writing of the data is slower than the rate you are reading it, you can perform multiple reads looking for coherent data. This is the least deterministic of the solutions. It would work well in Case Study #3.

## TOCTTOU

The final concurrency issue we will look at this month sounds more like a South American bird. TOCTTOU (pronounced "tock too") vulnerability occurs when data being used changes after you read it. Perhaps the simplest commonplace example of how this can happen in everyday use is in an airline reservation. The user looks at what flights are available at 7:00:00 (time-of-check) but then attempts to book the once-available flight at 7:01:00 (time-of-use) and is rejected because the flight has been fully booked by someone else.

**Listing 2—**Results of the reading device

```
flag1 = ReadTheFlag();
if (flag1 & 0x01 == 0)
{
            ReadData();
            if (flag1 == ReadTheFlag())
            {
                        // Use the data
            }
            else
            {
                        // Wait and try later
            }
}
else
{
            // Wait and try again later
}
```

```
The data is 0x000001FF
SPI reads busy bit and it is not set (Time-of-check)
8051 sets the busy bit
SPI reads byte 1 as 0xFF (Time-of-Use)
8051 writes byte 1 as 0x00
8051 writes byte 2 as 0x20
SPI reads byte 2 as 0x20
8051 writes byte 3 as 0x00
8051 writes byte 4 as 0x00
SPI reads byte 3 as 0x00
8051 clears the busy bit
SPI reads byte 4 0x00
SPI reads busy bit and it is not set
```

**Figure 2—**A potential condition that could occur when the solution that involves building an indication of the concurrency of the data is applied to Case Study #1.

Let's take an in depth look at the third solution above (busy bit) and show where this breaks down because of TOCTTOU issues. If we apply this solution to Case Study #1, the condition that could occur is shown in Figure 2.

In this case, even with the busy bit, because of TOCTTOU, the SPI gets incoherent data (0x2FF instead of either 0x1FF or 0x200). The SPI reads the busy bit (time-of-check) and takes action on the busy bit (time-of-use) and the SPI has some bytes from the first sample and some from the second.

## WHAT ARE SOME SOLUTIONS TO TOCTTOU?

A simple solution to this would be to use a more complicated busy indicator which is really an 8-bit counter where the low-order bit indicates busy.

The writing device would do the following:

```
flag |= 0x01;
WriteData();
flag++;
```

Note: OR a `1` into the flag instead of using `flag++` to guarantee synchronization. The reading device results are shown in Listing 2. Now we would be protected from incoherent data regardless of the overlaps of reading and writing the shared resource.

## CONCLUSION

Concurrency issues can create nightmarish problems for embedded systems designers. The designer of the software in the energy chip did not carefully think through the implications of having three devices all reading and writing to a common shared memory when the interface was created. We were using the interface and didn't know that it was a shared-memory interface. The problems of nonatomic operations and TOCTTOU issues on shared resources can create problems that are like hurricanes in Hartford, CT—they hardly ever happen, but when they do...

*Bob Japenga has been designing embedded systems since 1973. In 1988, along with his best friend, he started Micro-Tools, 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.*

## RESOURCE

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

## SOURCE

**71M6533 Teridian energy metering IC**
Maxim Integrated Products | www.maxim-ic.com