

Guidelines for Creating Robust Embedded Systems

Part 2 – Preventing Memory Leaks

By Bob Japenga

Scope

This is Part 2 of a white paper intended to provide a brief overview of lessons learned in creating robust systems over the past 35 years of embedded systems development (starting with the Intel 4040). This part will address Memory Leaks; what they are; how they are created; how to detect them and how to prevent them.

Memory Leaks

What is a Memory Leak?

A memory leak is defined as “A bug in a program that fails to free up memory that it no longer needs. As a result, the program grabs more and more memory until it finally crashes because there is no more memory left.”¹ See

http://en.wikipedia.org/wiki/Memory_leak for a good summary of memory leaks.

A Simple Example (in C - Note: There are many errors in this that will be discussed later)

```
char * foo(char *bar,unsigned int size)
{
    char *ptr = malloc(1000);
    *ptr = 0;
    strncat(ptr,bar,size);
    StripSpaces(ptr);
    *bar = 0;
    strncat(bar,ptr,strlen(ptr));
    return bar;
}
```

The memory allocated to ptr can never be usefully accessed or re-used and every time foo gets called a 1000 byte memory leak will occur.

What problems do Memory Leaks create?

One of the major problems with memory leaks is that they can go undetected and cause no harm even after days of testing or operation. Then one day, the system will fail to be able to allocate the necessary memory it needs to operate and the system will fail to operate as specified (hopefully if our other guidelines are followed, the system will restart).

For a system to be robust there must be zero memory leaks. Imagine that your embedded system runs 24/7 and there is function performed every hour. If that function allocates 1k bytes of data and does not free that memory, it will eat 8 meg after one year of operation. With some systems that have tons of memory, your embedded

application could have passed thousands of hours of testing and the memory leak go undetected.

How can we prevent Memory Leaks?

Some experts dodge the issue with statements like:

Programs must avoid dynamic data allocation, and avoid the use of (built-in) library functions that make use of dynamic memory allocation (e.g. *string.h*).²

Avoid using *string.h*? Avoid using dynamic data allocation? This advice works for military, aerospace and relatively simple embedded systems but is totally impractical for moderately complex industrial and commercial embedded systems. If we don't need these, we do follow this advice. However even relatively simple user interfaces now use GUI's that use massive amounts of dynamic data allocation.

There are three (simple to state – but difficult to implement) principles that we follow:

- a. Thoroughly educate all designers about how to design for zero memory leaks. The principle is simple but it must be hammered into the minds of every designer and programmer. In the famous words of Admiral Hyman Rickover “Good ideas are not adopted automatically. They must be driven into practice with courageous impatience.”
- b. Independently code review all designs – Memory leaks are often easier to detect during code review than during test.
- c. Utilize memory leak detection software during all phases of test. It is far easier to start using these tools early in development rather than trying to add them on when a leak is detected the hard way. With Visual Studio programming, you can use “*crtdbg.h*” and *CRTDBG_MAP_ALLOC*. For Linux (non-kernel programming) Eclipse provides some tools for detecting memory leaks. *memwatch* and *dmalloc* are mature and open source tools. (<http://www.icewalkers.com/Linux/Software/52790/memwatch.html> <http://dmalloc.com/>). *MemoryScape* from TotalView provides another tool.

Under Linux, using command line tools like “*top*” are inadequate but can be used to watch for large leaks. Get a set of good tools – learn how to use them – and then (Duh!) use them. Believe me – I know that in the rush of development, this can get missed.

I'm Programming Java – Do I need to worry about Memory Leaks?

From the IBM web site:

If the term *Java™ memory leak* seems like a misnomer to you, rest assured that you are not alone; after all, you were promised that Garbage Collection (GC) in Java will relieve you from the mundane duties of allocating, tracking, and freeing the heap memory. Indeed, the promise has been delivered, so it is reasonable -- and to quite an extent correct -- for you to conclude that there will be no memory leaks in your Java program. However, the catch is that GC can take care of only *typical* heap management problems.³

The above referenced IBM article has an excellent example of how you can create memory leaks with Java. If you are programming in Java it is well worth your time to fully understand how you can do this so that you can avoid it.

But we all are expert Java programmers and never create memory leaks this way with Java! But alas! We are still not out of the woods. We have found memory leaks in the Java implementations (from unmentioned suppliers). They can only be found with massive amounts of testing and measuring. Basically we noticed that by running the same sequence over and over we could exhaust 100 meg of RAM in 24 hours. And starting the Java thread with the command switch to limit dynamic memory usage did not stop the leak because the leak was in the JVM. When all was said in done, over 30 memory leaks were detected in the JVM.

The moral of the story is to test, re-test and test some more – all the while watching for memory leaks. Of course you won't find the leaks in your Java program but in the JVM.

In [Part 3](#) we will discuss Out of Bounds referencing.

¹ http://www.webopedia.com/TERM/M/memory_leak.html (they actually say “prevents it from freeing up...”)

² *Development Guidelines for Dependable Real-Time Embedded Systems* by Michael Short

³ http://www.ibm.com/developerworks/rational/library/05/0816_GuptaPalanki/