

# Guidelines for Creating Robust Embedded Systems

## Part 3 – Out of Bounds Memory References

By Bob Japenga

### Scope

This is Part 3 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 out of bounds memory references; what they are; how they are created; how to detect them; how to prevent them; and how to handle them should they occur.

### What are “out of bounds memory references”

Simply put, an “out of bound memory reference” is any read or write that is outside the range of a defined structure. It can include (but is not limited to) writing 21 characters into a 20 character string buffer; referencing index 21 in a 20 element array; or referencing an uninitialized or discarded pointer.

### Catching out of bounds memory references and recovering

This section is written for other programmers. None of us ever create software that references out of bounds memory locations. Not!

First we need to address two separate conditions: For systems with memory management and for those without.

### Systems without Memory Management

Most of the guidelines here can be applied to systems without memory management since this is the more difficult of the two systems. We will approach this from two perspectives: preventative and prescriptive.

#### Preventative

Of course the easy answer for this is not to write code that references out of bounds memory. Easy to say – hard to do. But here are some guidelines that we have followed:

1. Fully understand all of the functions that can create out of bounds conditions. For C these would include but are not limited to:
  - a. strcpy
  - b. strncpy
  - c. strcat
  - d. strncat
  - e. memcpy
  - f. memmove
  - g. all types of printf (sprintf, printf, fprintf, etc)
  - h. fgets
  - i. gets (avoid this one completely because there is no limit)

2. We recommend that you either create your own safe versions of these or use some libraries that have safe versions of these. We first saw this about 20 years ago when a company we were consulting for had these functions called `kstrncpy` and `kstrncpy` and so on. We finally asked someone and they said – “Oh those are Kevin’s `strncpy`’s. They are safe versions of the C functions.” We learned something that day.
3. As part of code review – search for all uses of these functions and analyze that the buffers are always adequate to the task.
4. Fully understand (remember Hyman Rickover’s injunction) how array referencing can create out of bounds conditions.
5. When programming in C, explicitly define the array size in both the declaration and in an extern
6. Include bounds checking option for arrays and pointers if your compiler allows it. For GNU gcc use `-fbounds-checking`

## Systems with Memory Management

Systems with memory management provide a lot of protection against out of range memory references. But sometimes I get lulled into passivity with the protection. The memory manager finds so many of my problems I think that it is finding all of them. Although the memory management scheme will prevent out of address space references, for the most part, they do not protect against out of bounds references within your address space. With this in mind, we provide the following guidelines for designing these systems;

1. Utilize the memory manager to the fullest extent possible. Let it do most of the hard work. If efficiency allows, create as many address spaces as possible within your architecture. For example, in Linux, create separate executables rather than separate threads if you have the memory. This will also force a discipline upon your design by minimizing common variables across threads.
2. Don’t allow threads to just terminate with a memory protection violation. Trap the error, restart the thread or the app as necessary and log the failure. This way, even if you have a problem – you can recover from it. I cannot tell you how many times we have not done this and it has come back to bite us.
3. Follow the guidelines for non-memory management systems

In [Part 4](#) we will discuss proper ways of referencing I/O.