

The Ganssle Group

Perfecting the Art of Building Embedded Systems

Learning to program has no more to do with
designing interactive software than learning to touch
type has to do with writing poetry.
- Ted Nelson


Seminars Newsletter Videos Tool & Book Reviews Special Reports Articles Random Rants
Computer Humor Contact/Search

Follow @jack_ganssle

Reentrancy

Most real time systems require a certain amount of reentrant code, yet too many programmers have no idea what this entails.

Published in Embedded Systems Programming, February 1993

 For novel ideas about building embedded systems (both hardware and firmware), join the 25,000+ engineers who subscribe to [The Embedded Muse](#), a free biweekly newsletter. The Muse has no hype, no vendor PR. It takes just a few seconds (just enter your email, which is shared with absolutely no one) to [subscribe](#).

By Jack Ganssle

For many reasons, debugging an interrupt-based system is much harder than one built of simple looping code. One of the most insidious sorts of bugs, that is tough to find and sometimes tougher to understand, is a reentrancy problem.

Reentrant functions, AKA "pure code", are often falsely thought to be any code that does not modify itself. Too many programmers feel if they simply avoid self-modifying code, then their routines are guaranteed to be reentrant, and thus interrupt-safe. Nothing could be further from the truth.

A function is reentrant if, while it is being executed, it can be re-invoked by itself, or by any other routine, by interrupting the present execution for a while.

Reentrancy was originally invented for mainframes, in the days when memory was a valuable commodity. System operators noticed that a dozen or hundreds of identical copies of a few big programs would be in the computer's memory array at any time. At the University of Maryland, my old hacking grounds, the monster Univac 1108 had one of the early reentrant FORTRAN compilers. It burned up a (for those days) breathtaking 32kw of system memory, but being reentrant, it required only 32k even if 50 users were running it. Each user executed the same code, from the same set of addresses.

A routine must satisfy the following conditions to be reentrant:

1. It never modifies itself. That is, the instructions of the program are never changed. Period. Under any circumstances. Far too many embedded systems still violate this cardinal rule.
2. Any variables changed by the routine must be allocated to a particular "instance" of the function's invocation. Thus, if reentrant function FOO is called by three different functions, then FOO's data must be stored in three different areas of RAM.

Item (2) deserves a bit more discussion. One of the better trends in this industry is the use of professional software engineers in firmware development projects. In the old days the hardware designer, who had perhaps little formal software engineering education other than that gained in the school of hard knocks, wrote the code as an afterthought to building the hardware. Real software types understand and use more sophisticated program structures, like recursion, leading to cleaner code (usually), but with new sorts of perils.

A recursive function calls itself. The classic example is computing $n!$ (n factorial), which is most elegantly done with a few lines of recursive code. Any recursive function must be reentrant, because each instance of its execution must have its own set of local variables to avoid corrupting

Do you need to **eliminate bugs** in your firmware? **Shorten schedules?** My one-day **Better Firmware Faster** seminar will teach your team how to operate at a world-class level, producing code with far fewer bugs in less time. It's fast-paced, fun, and covers the unique issues faced by embedded developers. [Here's](#) information about how this class, taught at your facility, will **measurably** improve your team's effectiveness.

Win a differential preamp for your scope! Enter [here](#).



Advertise with us! Reach 130K+ embedded developers per month. More info [here](#).

any other instance.

For example, consider the following very simple recursive function, taken from an example in the Borland C++ Programmer's Guide

```
double power(double x, int exp)
{
  if (exp<=0) return(1);
  return(x*power(x, exp-1));
}
```

This function will behave correctly in its obvious recursive environment as well as in an interrupting real-time system, where "power" may be called from both a main-line routine and from an interrupt service routine. The function is indeed pure - all the variables used are created for each instance of its execution.

Suppose we modify the function as follows:

```
double power(double x)
{
  if (exp<=0) return(1);
  --exp;
  return(x*power(x));
}
```

where exp is now defined as a public variable, accessible to many other functions. The function still works correctly. However, if this function was called by, say, main(), and then an interrupt which calls the same function came along while it is executing, it will return an incorrect result. The variable exp is fixed in memory; it is not unique to each call, and is therefore shared between callers, a disaster in an interrupting environment.

So, pure code must never modify itself, and must never, ever share data with any other instance of itself, whether invoked by recursion, an interrupt service routine, or some other process.

In the example of the previously mentioned FORTRAN compiler, while the compiler itself was loaded but once into system memory, each user allocated a chunk of his or her memory area to the compiler for its own internal use. Every variable or data array the compiler used was referenced via a base register that set the beginning of the compiler's data space for that user.

C is elegantly oriented towards reentrancy, as automatic local variables are generated as stack frame entries each time a function is invoked. Still, as we've seen, it is possible to write problematic code in C. In assembly, of course, chaos rules.

Embedded Reentrancy

Any real time embedded system must address reentrancy issues, as all but the simplest systems will most likely require reentrant code. Before seeing where reentrancy is critical, perhaps we should look at typical areas where code is "impure".

Some processors have limited I/O addressing capabilities. For example, the 8085 can only send data to specific, hard coded addresses. There is no mechanism to generate an indirect I/O port (like "output to the port contained in register C"). The traditional work around was to generate output and return instructions in RAM, dynamically patch in the desired port, and then call the RAM-based code. This violates every rule of reentrancy. A better approach is to generate a table of output instructions in ROM space, and indirectly call the proper one. This does, however, consume vast amounts of memory.

The 1802 had no stack... at all. There were no call or return instructions (no kidding!). Reentrancy was all but impossible.

Even modern real time operating systems sometimes have short machine-specific segments of code that are not reentrant. Since there is probably no other more dangerous spot for impure code, all the vendors do a good job of protecting the impure sections by disabling interrupts for a brief time.

Reentrancy is crucial in any section of code that may be invoked by another process. In a real time OS, each task is independent and subject to reentrancy concerns. Any subroutine shared

between tasks can be a real source of concern, since the RTOS can context switch on a timer tick during the execution of this critical routine, and then schedule another task that invokes the same function.

There is a much more insidious problem. Suppose your main line routine and the ISRs are all coded in C. The compiler will certainly invoke runtime functions to support floating point math, I/O, string manipulations, etc. If the runtime package is only partially reentrant, than your ISRs may very well corrupt the execution of the main line code. This problem is common, but is virtually impossible to troubleshoot since symptoms result only occasionally and erratically. Can you imagine the difficulty of isolating a bug which manifests itself only occasionally, and with totally different characteristics each time?

One moral is to be sure your compiler has a pure runtime package.

Even assembly programmers suffer from this malaise, as any common low-level routine, shared between an ISR and other code, *must* be pure. If you purchase a floating point library, communications library, etc., be sure the vendor guarantees reentrancy in all modes.

Since so many embedded systems execute from ROM, some level of reentrancy is assured. No matter how hard one tries, it is impossible to write self modifying code, at least in the ROM space!

Unfortunately, this breeds a certain complacency. My company sells emulators. Our number one source of customer support calls comes from programmers whose code works from ROM, but crashes when executing from the emulator's internal memory. The problem is always the same - the user's code inadvertently writes over itself. In ROM, the problem goes unnoticed. Surely, though, this indicates a more serious problem. That write should probably have occurred to some data area in RAM, which is not getting updated. Or, if the write is indeed spurious, then perhaps some other random setting of the index registers will cause the write to trash the stack or some other critical data structure.

Given a halfway decent CPU it's not too hard to write reentrant code from scratch (decent stack resources are the biggest requirement), but I've found that it is almost impossible to make a big impure program reentrant. Generally these are monsters that relay on heaps of global variables.

If reentrancy is needed in only small areas it is sometimes possible to protect the worst code by disabling interrupts around it. Then, no ISR can synchronously muck with the critical resources. This may not work well in a multitasking system.

Unfortunately, disabling interrupts brings its own problems. Interrupt latency increases, and in some cases the code could miss interrupts altogether.

Many years ago I had to convert an assembly language floating point package to pure code. I tried to shoehorn fixes in, but in the end found that only a total rewrite cured the problem.

Finding Purity

How do you know if your code is truly reentrant? No matter how carefully you design a new project, it is far too easy to inadvertently slip in the occasional impure data reference. In an upgrade to an old project the problem is more severe, as you may not be totally comfortable with the code's structure or the original programmer's intents.

I doubt if there is a complete test exist for reentrancy. Some partial tests are easy and well worth the effort.

It's pretty easy to find code that accidentally (or otherwise) writes over itself. Any decent emulator lets you catch writes to program space. When using a ROM monitor, simulator, or the like, regularly run checksums on the code. If the checksum ever changes (for a particular version of the code), then something is wrong.

I do global searches for STATIC declarations in C. STATICS can indicate a potential reentrancy problem.

In assembly language routines, any direct reference to a RAM address is cause to examine reentrancy issues. There is nothing implicitly wrong with a MOV [COUNT],AX; most of the time this sort of effective address generation will be fine even in a reentrant routine. Still, it might be a

flag that the code is impure. A sure-fire solution is to store local variables in the current stack frame.

How can you tell if a purchased runtime package is reentrant without reading all the code? One crude approach is to link the program without the package (stubbing out the called routines), and again with it. Compare the linker's reported use of data space. A really clean reentrant package will require no additional RAM beyond the stack and/or heap.

The Ganssle Group - info@ganssle.com - copyright TGG, all rights reserved. Contact info [here](#). Interested in advertising with us? More information [here](#).