Reentrancy

by Jack G. Ganssle

Virtually every embedded system uses interrupts; many support multitasking or multithreaded operations. These sorts of applications can expect the program's control flow to change contexts at just about any time. When that interrupt comes, the current operation gets put on hold and another function or task starts running. What happens if functions and tasks share variables? Disaster surely looms if one routine corrupts another's data.

By carefully controlling how data is shared, we create *reentrant* functions, those that allow multiple concurrent invocations that do not interfere with each other. The word *pure* is sometimes used interchangeably with reentrant.

Like so many embedded concepts, reentrancy came from the mainframe era, in the days when memory was a valuable commodity. In those days compilers and other programs were often written to be reentrant, so a single copy of the tool lived in memory, yet was shared by perhaps a hundred users. Each person had his or her own data area, yet everyone running the compiler quite literally executed the identical code. As the operating system changed contexts from user to user it swapped data areas so one person's work didn't effect any other. Share the code, but not the data.

In the embedded world a routine must satisfy the following conditions to be reentrant:

1. It uses all shared variables in an atomic way, unless each is allocated to a specific instance of the function.
2. It does not call non-reentrant functions.
3. It does not use the hardware in a non-atomic way.

Quite a mouthful! Let's look at each of these in more detail.

Atomic variables

Both the first and last rules use the word *atomic*, which comes from the Greek word meaning indivisible. In the computer world, atomic means an operation that cannot be interrupted. Consider the assembly language instruction:

```
mov     ax,bx
```

Since nothing short of a reset can stop or interrupt this instruction, it's atomic. It will start and complete without any interference from other tasks or interrupts

The first part of Rule 1 requires the atomic use of shared variables. Suppose two functions each share the global variable `foobar`. Function A contains:

```
temp = foobar;
temp += 1;
foobar = temp;
```

This code is not reentrant, because `foobar` is used non-atomically. That is, it takes three statements to change its value, not one. The `foobar` handling is not indivisible; an interrupt can come between these statements and switch context to the other function, which then may also try and change `foobar`. Clearly there's a conflict; `foobar` will wind up with an incorrect value, the autopilot will crash, and hundreds of screaming people will wonder "why didn't they teach those developers about reentrancy?"

Suppose, instead, Function A looks like:

```
foobar += 1;
```

Now the operation is atomic, right? An interrupt cannot suspend processing with `foobar` in a partially changed state, so the routine is reentrant.

Except... do you really know what your C compiler generates? On an x86 processor that statement might compile to:

```
mov     ax,[foobar]
inc     ax
mov     [foobar],ax
```

which is clearly not atomic, and so not reentrant. The atomic version is:

```
inc     [foobar]
```

The moral is to be wary of the compiler. Assume it generates atomic code and you may find "60 Minutes" knocking at your door.

The second part of the first reentrancy rule reads "...unless each is allocated to a specific instance of the function." This is an exception to the atomic rule that skirts the issue of shared variables.

An "instance" is a path through the code. There's no reason a single function can't be called from many other places. In a multitasking environment, it's quite possible that several copies of the function may indeed be executing concurrently. (Suppose the routine is a driver that retrieves data from a queue; many different parts of the code may want queued data more or less simultaneously). Each execution path is an "instance" of the code. Consider:

```
int foo;
void some_function(void) {
        foo++;
}
```

`foo` is a global variable whose scope exists outside that of the function. Even if no other routine uses `foo`, `some_function` can trash the variable if more than one instance of it runs at any time.

C and C++ can save us from this peril. Use automatic variables. That is, declare `foo` inside of the function. Then, each instance of the routine will use a new version of `foo` created from the stack, as follows:

```
void some_function(void) {
        int foo;
        foo++;
}
```

Another option is to dynamically assign memory (using malloc), again so each incarnation uses a unique data area. The fundamental reentrancy problem is thus avoided, as it's impossible for multiple instances to modify a common version of the variable.

Two more rules

The other rules are very simple. Rule 2 tells us a calling function inherits the reentrancy problems of the callee. That makes sense. If other code inside the function trashes shared variables, the system is going to crash. Using a compiled language, though, there's an insidious problem. Are you sure—really sure—that all of the runtime library functions are reentrant? Obviously, string operations and a lot of other complicated things make library calls to do the real work. An awful lot of compilers also generate runtime calls to do, for instance, long math, or even integer multiplications and divisions.

If a function must be reentrant, talk to the compiler vendor to ensure that the entire runtime package is pure. If you buy software packages (like a protocol stack) that may be called from several places, take similar precautions to ensure the purchased routines are also reentrant.

Rule 3 is a uniquely embedded caveat. Hardware looks a lot like a variable; if it takes more than a single I/O operation to handle a device, reentrancy problems can develop.

Consider Zilog's SCC serial controller. Accessing any of the device's internal registers requires two steps: first write the register's address to a port, then read or write the register from the same port, the same I/O address. If an interrupt fires between setting the port and accessing the register, another function might take over and access the device. When control returns to the first function, the register address you set will be incorrect.

Keeping code reentrant

What are our best options for eliminating non-reentrant code? The first rule of thumb is to avoid shared variables. Globals are the source of endless debugging woes and failed code. Use automatic variables or dynamically allocated memory.

Yet globals are also the fastest way to pass data around. It's not always possible to entirely eliminate them from real time systems. So, when using a shared resource (variable or hardware) we must take a different sort of action.

The most common approach is to disable interrupts during non-reentrant code. With interrupts off, the system suddenly becomes a single-process environment. There will be no context switches. Disable interrupts, do the non-reentrant work, and then turn interrupts back on.

Shutting interrupts down does increase system latency, reducing its ability to respond to external events in a timely manner. A kinder, gentler approach is to use a mutex (also known as binary semaphore) to indicate when a resource is busy. Mutexes are simple on-off state indicators whose processing is inherently atomic. These are often used as "in-use" flags to have tasks idle when a shared resource is not available.

Nearly every commercial real-time operating system includes mutexes. If this is your way of achieving reentrant code, by all means use an RTOS.

Recursion

No discussion of reentrancy is complete without mentioning *recursion*, if only because there's so much confusion between the two.

A function is recursive if it calls itself. That's a classic way to remove iteration from many sorts of algorithms. Given enough stack space, this is a perfectly valid—though tough to debug—way to write code. Since a recursive function calls itself, clearly it must be reentrant to avoid trashing its variables. So all recursive functions must be reentrant, but not all reentrant functions are recursive. **esp**

Jack G. Ganssle is a lecturer and consultant on embedded development issues, and a regular contributor to Embedded Systems Programming. *Contact him at jack@ganssle.com.*

Resources

Greenberg, Kenneth F. "Sharing Your Code," *Embedded Systems Programming*, August 1990, p 40.

Barr, Michael. "Choosing a Compiler: The Little Things," *Embedded Systems Programming*, May 1999, p. 71.

Simon, David. *An Embedded Software Primer*. Reading, MA: Addison-Wesley, 1999.

✂ CUT HERE ✂