The Embedded Muse 126

Editor: Jack Ganssle (jack@ganssle.com) March 9th, 2006

EDITOR: Jack Ganssle, jack@ganssle.com

CONTENTS:

Editor’s Notes

Will wonders never cease? I thought the relay computer described in Muse 125 was pretty amazing. The Legos machine in Muse 124 blew my mind. Now David Strip sends this: “How about Danny Hillis's tinkertoy computer? (Actually, it's Hillis and several other folks, but his is the name that has come to be associated with the machine.) Here's a link to a Scientific American article that describes the system in detail - http://www.rci.rutgers.edu/~cfs/472_html/Intro/TinkertoyComputer/TinkerToy.html . Here's a link to a pretty good picture of the beast. http://www.cob.sfasu.edu/sbradley/tinkertoy.html .”

Ventoro has released their interesting 115 page “Offshore 2005 Research, Preliminary Findings and Conclusions.” A lot of engineers are worried about competition from inexpensive developers overseas, and this report has some worthwhile insights.

I track offshoring statistics pretty closely, and find no consistency between studies and figures. Some numbers suggest that the end for first-world engineers is nigh; others make one think we’re on the edge of a terrible shortage of home-grown talent. One this is clear: the world is changing. Engineering – and engineers – will have to change as well. My take on the subject is here: http://embedded.com/showArticle.jhtml?articleID=180206418 . Please note, though, that the first two paragraphs are tongue-in-cheek.

Getting Better Firmware

In 1971 Intel invented the microprocessor… but few knew what to use it for. Even Intel predicted a tiny market for the device. But they persisted and the following year came out with the 8008, the first 8 bitter. Companies quickly realized how adding a bit of intelligence to their products reduced costs and increased functionality. The embedded systems industry was born.

I was in college, working as an electronics tech at a small outfit named Neotec, but was the only employee at the company who knew anything about programming. (We had been using a consultant, who remains in occasional contact.) They tapped me to be their first in-house firmware developer. Though it was easy to master 8008 assembly language, my software engineering skills were null. This was typical of the industry. All early embedded people had been trained as EEs, not computer scientists. 30 years of lessons learned in the software business were ignored by this fledgling industry.

Programs were small and management clueless about software projects. But the 8008 had a max address space of 16k so hacking was reasonably effective.

Projects grew in pace with expanding address spaces offered by the 8080, 8088, 68k and more. Feature sets exploded. Complexity skyrocketed. Glass claims that for every 25% increase in features complexity grows by 100%, a number that's hard to dispute.

At Neotec we reacted in the usual manner. Hired more people. Worked longer hours. Made promises we never fulfilled. And, of course, shipped plenty of bug fixes to irate customers.

I eventually left and started a consultancy with a partner. What a ride we had! What fun it was to put our gear in, say, a steel mill with a hundred foot long piece of 2000 degree steel four inches thick moving at a breakneck pace. (That used a PDP-11 and a number of Z80s). Or systems that lived 10,000 feet deep in the ocean, running on a few AAs for a year (RCA 1802). Then there was the White House security system (over 100 8085s), which we designed and installed during the Reagan years.

After too many late nights we started to sense that there were better approaches than our usual heroics. But what was wrong with us? Why were we working so hard?

Then I started an in-circuit emulator company. Now providing tech support to embedded developers all over the world I discovered something interesting. ALL of us had the same problems. Little groups of engineers all used heroics, and nearly all delivered late. Ironically, we who invented the communications age didn't talk to each other. So we all faced the same sorts of challenges, thinking that we were doing something wrong and that other teams had the secrets to success.

I studied software engineering. The field, all but ignored by firmware folks, had a great body of accumulated wisdom. Frustrated with the state of our practices, in 1988 I started writing for various electronics/embedded publications about technical issues and better ways to develop code.

Over the years I've had the honor of serving on a number of boards, including one technical college, and was even tapped to teach engineering. In those cases it was shocking to see that too many schools push the same old dysfunctional software methods: here's how a "for" loop works. Now write some code.

Ten years ago I sold the emulator business and tried retirement. After one day it seemed a rather bleak and meaningless state, so looked for something interesting to do, something which could be of benefit to others. Turns out I'm clueless about most traditional philanthropic activities, so decided to try and help out in the one field I know and love. Today I continue to write, and continue to try to educate, berate, and encourage embedded developers and their managers to improve their techniques.

A couple of companies asked me to come in and provide training about firmware development. Well, high school speech class was a trauma. I was terrified of talking in front of anyone, a fear that persisted into my 30s. But as a personal challenge I had decided to get over this irrational dread and took some speaking classes, and then practiced. When the Embedded Systems Conference was born they asked me to give a talk… and I found it an exhilarating experience. What fun to interact with all of these very smart people! It seemed natural to develop a seminar that would help folks get better code faster.

Over the last decade I've given the seminar to hundreds of companies and thousands of developers.

No one is naïve enough to think there's a silver bullet that will solve all our problems. This field is intrinsically tough. Even if we knew how to generate 10 million lines of perfect code in a day, customers will scream to shorten the schedule to an hour.

But… it *is* possible to create better code faster. It *is* possible to drastically reduce bug rates and shorten schedules. Those two do go hand in hand – Capers Jones showed that bugs are number one cause of missed schedules.

For example, there's an approach that drives the usual 5-10% pre-test bug rate down by three orders of magnitude. Though I mention it in the seminars I don't teach the technique as few of us, for a variety of reasons, will ever embrace it. That seems odd, perhaps, but my interest is showing methods that most of us can start using *immediately* to get dramatic improvements.

And that's the entire thrust of the seminar. What can we do *today*, in a stealth fashion, under the scope of management's radar, to get better code faster? What can we do without embracing some monster methodology that few will really use, that requires a revolution in the engineering department that probably won't happen? What can we individuals do now to improve our performance?

That's my passion. I'm frustrated with consumer products that don't work well, with safety-critical code that isn't, with a situation where every three-year-old knows if something electronical acts odd, cycle power. Can't we find ways to meet the schedule?

We can. I'll show you how.

If you want more details and the usual marketing hype, see http://www.ganssle.com/classes.htm. Most of the time I present the seminar on-site, when a company asks me to come in for a day. Occasionally I do a public seminar at a hotel. We have decided to do public seminars in Dallas and Denver this April. About half the participants fly in; the others tend to be local. Please join us; it's a jam-packed but always fun day. I guarantee you'll learn a lot, all of which is backed up with hard numbers you can take back to the boss and your colleagues. But seats fill quickly.

</shameless promotion>
<resume normal Embedded Muse>

On Reentrancy

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 is 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 the other'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".

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. Everyone executed the same code, from the same set of addresses. Each person had his or her own data area, yet everyone running the compiler quite literally executed 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.

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, 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; an interrupt will not 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 the code might look like:

```
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 beyond that of the function. Even if no other routine uses foo, some_function can trash the variable if more than one instance if 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 stamp on a common version of the variable.

The rest of the 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 the runtime package is reentrant? Obviously string operations and a lot of other complicated things use runtime 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 insure 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 insure 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.

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 no end of 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 entirely possible to 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.

Most times this sort of code looks like:

```
long i;
void do_something(void){
  disable_interrupts();
  i+=0x1234;
  enable_interrupts();
}
```

This solution looks great and is used everywhere… but does not work. If do_something() is a generic routine, perhaps called from many places, and is invoked with interrupts disabled, it returns after turning them back on. The machine's context is changed, probably in a very dangerous manner.

Don't use the old excuse "yeah, but I wrote the code and I'm careful. I'll call the routine only when I know that interrupts will be on." A future programmer probably does not know about this restriction, and may see do_something() as just the ticket needed to solve some other problem… perhaps when interrupts are off.

Better code looks like:

```
long i;
void do_something(void){
  push interrupt state;
  disable_interrupts();
  i+=0x1234;
  pop interrupt state;
}
```

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 semaphore to indicate when a resource is busy. Semaphores are simple on-off state indicators whose processing is inherently atomic, often used as "in-use" flags to have routines idle when a shared resource is not available.

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

Don't have an RTOS? Sometimes I see code that assigns an in-use flag to protect a shared resource, like this:

```
while (in_use); //wait till resource free
in_use=TRUE;       //set resource busy
Do non-reentrant stuff
in_use=FALSE;      //set resource available
```

If some other routine has access to the resource it sets in_use true, cause this routine to idle till in_use gets released. Seems elegant and simple… but it does not work. An interrupt that occurs after the while statement will preempt execution. This routine feels it now has exclusive access to the resource, yet hasn't had a chance to set in_use true. Some other routine can now get access to the resource.

Some processors have a test-and-set instruction, which acts like the in-use flag, but which is interrupt-safe. It'll always work. The instruction looks something like:

```
Tset  variable    ; if (variable==0){
                  ;     variable=1;
                  ;     returns TRUE;}
                  ;  else {returns FALSE;}
```

If you’re not so lucky to have a test-and-set, try the following:

```
loop: mov   al,0        ; 0 means “in use”
      xchg  al,variable
      cmp   al,0
      je    loop        ; loop if in use
```

If al=0, we swapped 0 with zero; nothing changed, but the code loops since someone else is using the resource. If al=1, we put a 0 into the “in use” variable, marking the resource as busy. We fall out of the loop, now having control of the resource. It’ll work every time.

Yet More on Tools

The tool discussion that started a couple of issues ago is still generating a lot of response. Thanks, all!

David Bley wrote: “I have a piece of software that I would like to recommend. It is not free, but I use it everyday. It is a free-form database (records are note cards and you can define fields or not) and its most powerful feature is the neural search. It will return every card that contains the list of words that you type in the search field. I put everything in it that I have trouble remembering and can always quickly access it. The program is called Info-Select and I have been using it for many years. Their url is: http://www.miclog.com/ .”

John Johnson sent: “I think this tip - very, very simple tip - is useful in tracking development and debugging efforts. In the MPLAB IDE one can add a text file to the project file list under "Other" files.

“The file is a mouse click away and can be used to collect or capture notes, ideas, and in tracking development and debugging efforts which is a tedious activity that often falls in the cracks. I would like to see this feature in all IDEs.”

Jobs!

Let me know if you’re hiring firmware or embedded designers. No recruiters please.

Silicon Engines is an established design engineering company specializing in embedded control applications for automotive electronics, industrial electronics, and data communications. We are currently expanding our product development activities.

Hardware engineer: Schematic capture, PCB layout, Protel experience a plus, Debugging prototypes.

Software engineer: Windows CE, C, Visual Basic, C++, C#, GUI experience.

Engineers will work closely with clients from major corporations in development and consulting. We are looking for energetic self-starters who enjoy a hard-working, informal atmosphere. We offer competitive salary, benefits, great experience and learning opportunities. Please mail, fax or e-mail your resume to Silicon Engines. Principals only, please.

Silicon Engines, Ltd.
3550 W. Salt Creek Lane
Suite 105
Arlington Heights, IL 60005
Fax: 847-637-1185
E-mail: hr@siliconengines.net

Joke for the Week

Kathy Hohstadt sent, as she put it, a joke from the female point of view:

INSTALLING HUSBAND Version 1.0

Dear Tech Support,

Last year I upgraded from Boyfriend 5.0 to Husband 1.0, and noticed a distinct slow-down in overall system performance - particularly in the Flower and Jewelry applications, which operated flawlessly under Boyfriend 5.0.

In addition, Husband 1.0 uninstalled many other valuable programs, such as Romance 9.5 and Personal Attention 6.5, and then installed undesirable programs such as NFL 5.0, MLB 3.0, NBA 4.0, NASCAR 4.2 and Golf Clubs 4.1. Conversation 8.0 no longer runs, and Housecleaning 2.6 simply crashes the system. I've tried running Nagging 5.3 to fix these problems, but to no avail.

What can I do?

Signed,
Desperate

Dear Desperate:

First keep in mind, Boyfriend 5.0 is an Entertainment Package, while Husband 1.0 is an Operating System.

Please enter the command: "http//www.I-Thought-You-Loved-Me.com" and try to download Tears 6.2, and don't forget to install the Guilt 3.0 update.

If that application works as designed, Husband 1.0 should then automatically run the applications Jewelry 2.0 and Flowers 3.5. But remember, overuse of the above application can cause Husband 1.0 to default to Grumpy Silence 2.5, Happy Hour 7.0, or Beer 6.1. Beer 6.1 is a very bad program that will download the Snoring Loudly Beta.

Whatever you do, DO NOT install Mother-in-law 1.0 (it runs a virus in the background, that will eventually seize control of all your system resources).

Also, do not attempt to reinstall the Boyfriend 5.0 program. This is an unsupported application and will crash Husband 1.0.

In summary, Husband 1.0 is a great program, but it does have limited memory, and cannot learn new applications quickly. You might consider buying additional software to improve memory and performance. We recommend Hot Food 3.0 and Lingerie 7.7.

About The Embedded Muse

The Embedded Muse is an occasional newsletter sent via email by Jack Ganssle. Send complaints, comments, and contributions to him at jack@ganssle.com.

To subscribe, send a message to majordomo@ganssle.com, with the words "subscribe embedded *your-email-address*" in the body. To unsubscribe, change the message to "unsubscribe embedded *your-email-address*".

The Embedded Muse is supported by The Ganssle Group, whose mission is to help embedded folks get better products to market faster. We offer seminars at your site offering hard-hitting ideas - and action - you can take now to ***improve firmware quality and decrease development time***. Contact us at info@ganssle.com for more information.