The Embedded Muse 97

Editor: Jack Ganssle (jack@ganssle.com) May 4, 2004

EDITOR: Jack Ganssle, jack@ganssle.com

CONTENTS:

Editor's Notes

Want to learn to design better firmware faster? Join me for a one-day course in Chicago on May 17. This is the only non-vendor class that shows practical, hard-hitting ways to get your products out much faster with fewer bugs. See http://www.ganssle.com/classes.htm for more details. There's also cheap fly-in options listed on the web site for folks coming from out-of-town. Just a few seats are still available.

I often do this seminar on-site, for companies with a dozen or more embedded folks who'd like to learn more efficient ways to build firmware. See http://www.ganssle.com/onsite.htm.

I've created a video titled "Develop Firmware in Half the Time" that distills the basic ideas and processes needed to efficiently crank out great firmware. There's more information available at http://www.ganssle.com/video.htm.

Engineers send me their resumes, asking for editing assistance and advice on making a brag sheet that really sells. Sometimes they flood in here in almost biblical quantities. On one memorable day over 200 arrived! Alas, I simply haven't the time needed to even respond to such a volume of requests, let alone edit so much material. So I've written a guide to creating a great resume. Find it at: http://www.ganssle.com/sellyourself.pdf.

Along similar lines I've updated the How to Become an Embedded Geek report (http://www.ganssle.com/startinges.pdf). Till the recession started my biggest source of

email was from people who want to enter this field. This paper gives plenty of opinionated advice.

Robert Schwalb writes: “You mentioned several tools that generate McCabe metrics, but missed my favorite, CDoc, from Software Blacksmiths (http://www.swbs.com/). I use this package mainly to generate cross references and comment blocks, but it also includes a metrics package. Along with PC-Lint, it's one of my favorite tools.”

In response to my review of Glass’s book in the last Muse (Facts and Fallacies of Software Engineering), Howard Speegle sent along his top 10 list of engineering fallacies:

1. Two engineers = twice the output
2. The initial specification is complete and entirely accurate. Any errors or omissions found will have no schedule effect.
3. If the start date slips, there is no reason to change the completion date.
4. There is no schedule impact caused by features and requirements added after the schedule is agreed to.
5. There is no need for further work as soon as the main features appear to be operational.
6. Any manager can make a better decision during a meeting, than any engineer can after three months study.
7. Engineers don't understand business.
8. There is no schedule impact caused by management re-assigning n members of the team.
9. Experience doesn't matter. Anyone who can create a program that appears to work is just as good as anyone else. "My son is still in high school and he can program circles around you."
10. "If you really cared about this project and this company and your job, you would work harder."

Autobauding

I moved recently and in sorting out the various boxes of files came across an old algorithm that’s still quite useful.

Suppose your product connects to a terminal or other computer via an RS-232 link. Marketing wants to support all sorts of baud rates. It seems silly to make the user set a DIP switch or otherwise configure the system to the rate - isn’t there a way for your embedded code to figure out the exact rate of the incoming serial stream?

If you can make one rule, that the user must type a character after powering things up, then it’s easy to build autobauding software. If the remote device is a computer instead of

a terminal, and that computer is running your proprietary software, life is even easier as you can simply makes sure the first thing the software does is to send a character.

A single character is all you need to determine the baud rate of the serial data. The procedure is as follows:

1) Program the UART's baud rate generator to the highest rate you handle (say, 38,400 baud).

2) Watch the UART's Data Ready bit. When asserted, the character has arrived.

3) Now start a loop that counts how many characters arrive in the next quarter second or so. That's right... even though only one character gets transmitted, your UART will think more than one is received (for the case where the transmission speed is not matched to the receive speed).

Suppose the sender transmits a space (0x20, mostly zeroes) at, say, 9600 baud. We're listening at 38,400 so the receiver completely assembles a character when the space is only about ¼ transmitted. The UART will immediately trip into operation again... and again... and again... until the character is completely transmitted.

The UART's response is only a function of time. When it gets a start bit, it fully expects to be done with the character when the time for 8 data bits and one stop elapses. At 38,400 baud this is around 10*(1/38400) sec, or 260 microseconds (depending on your selections for parity and all). But, the 9600 baud space will transmit for about 10*(1/9600), or better than a millisecond.

That single space character looks like about 4 (scrambled) characters to the UART. At other baud rates, the number of characters counted will differ.

In my extensive experiments I've found that the number of characters detected is very stable.

4) To complete the algorithm, empirically determine the counts expected for a space at each baud rate, and write code that compares the count received to an anticipated value for each baud rate.

It's a good idea to compare to a range of values to add margin to your design. If the UART sees 4 characters at 9600 and 8 at 4800, use the in-between value - 6 - as the selection criterion.

Musings on Debugging

Since most of us write error-free code, debugging is hardly an issue. We carefully plan our design, include range checks on arguments, have associates read the code before trying it out, and do everything possible to insure that it runs correctly immediately.

Not.

The debugger industry thrives because programmers make mistakes - LOTs of mistakes. Errors are expected and tolerated.

It would be awful to be a civil engineer. Can you imagine telling your boss that the bridge you just erected has a few bugs, "but don't worry - we'll release version 1.1 in a few months!"

Write your code assuming it will be riddled with bugs. Do all of the good things we know we should do.... yet that are too often ignored.

Crazy, complex, convoluted code is simply unacceptable. It's always either impossible to debug, or too costly to debug. By assuming it will be full of bugs, then you clearly should either document it to death or avoid the complex code altogether. Remember Brian Kernighan's insight: "Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."

One old adage is "program for clarity first, optimize second". Nah. Never remove clarity. You might be a genius; assume your successor will be an idiot.

If, when reading your code, you ever see something that makes you scratch your head and wonder how it does something: instantly rewrite or recomment it! The agile community strongly pushes "refactoring", which means rewriting code to improve its structure without changing its functionality. It's a great practice.

Constrain time-sensitive code to small routines. Spawn tasks off that can process non-critical activities in the background. Clearly identify the timing limitations expected in each interrupt service routine.

Use decent tools. Your time is expensive. Yet, according to a poll I did (http://embedded.com/pollArchive/?surveyno=12900001) 86% of us won't, or aren't allowed, to spend more than $1k on tools meant to enhance the code's quality.

Never, ever, not in a thousand years, write an I/O instruction in the body of the code. Yes, by their nature embedded systems interact constantly with the outside world. Restrict I/O to subroutines ("get_uart_data"). When performance is an issue, and you can't stand the overhead of the CALLs, RETURNs, and inherent argument stacking, then put the I/O in a C macro. Someday the I/O WILL change. Maybe tomorrow when the bloody vendor discontinues the peripheral.

If you are using decent tools, remove them and burn a ROM once in a while. It's a quick way to insure there's not a lurking uninitialized variable, or weird hardware problem, that will keep the system from running standalone.

Use a reasonable debugging strategy. Don't start changing things for no reason. Never proceed without truly understanding the problem. This is the curse of all projects: a very smart person gets just a glimpse at a problem and immediately jumps to a conclusion before even finding out what all of the symptoms are.

Never make assumptions. It's easy to toss out one symptom because "it's clearly related to the other problem." Are you sure? Can you prove it? Symptoms are all you have to go on when debugging; be reluctant to discard them because they don't fit your model of possible causes.

Finally, use the scientific method:

```
For (i=0; i< # findable bugs; i++)
  {
    1) Observe the behavior to find the apparent bug; observe
       collateral behavior to gain as much information as possible
       about the bug;
    2) Eliminate the obvious (power problems, etc.)
    3) Generate a hypothesis;
    4) Generate an experiment to test the hypothesis;
    5) Fix the bug;
};
```

Do check out Steve Litt's troubleshooters.com, which has some valuable advice. His book, Troubleshooting Techniques of the Successful Technologist (available on his website) is a valuable tool for people looking for more scientific debugging strategies.

Jobs!

Let me know if you're hiring firmware or embedded designers. I'll continue to run notices for embedded developers as long as the job situation stays in the dumper.

Michael Pont writes: "We are currently advertising for a "Lecturer in Embedded Systems" to join the Embedded Systems Laboratory at the University of Leicester (UK). Further details are on the university WWW site: http://www.le.ac.uk/personnel/jobs/a&r.html".

Spirent Communications wants to hire an engineering manager for embedded systems development, who will be is responsible for the ongoing enhancement and continued maintenance of the company's Linux-based embedded platform. This position requires working closely with development managers, program managers and team members to help direct the product design and to create effective development, test, release and support strategies. This person will supervise 6 to 8 people, in Calabasas, California. Contact Carol Cocchiarella, (818) 676-2300.

Joke for the Week

```
Geek wrote: Microsoft Project allows us to set targets -- they seem to
be centered on our backs.
```

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.