The Embedded Muse 134

Editor: Jack Ganssle (jack@ganssle.com) October 11, 2006

EDITOR: Jack Ganssle, jack@ganssle.com

CONTENTS:

Editor's Notes

Want to become your company's embedded guru? I'll present my "Better Firmware Faster" seminar December 1 in Santa Clara, California. Check out http://www.ganssle.com/classes.htm for details. The site is just a few miles from San Jose airport, which is inexpensively served by Southwest Airlines and others. Come for the seminar on Friday and explore San Francisco over the weekend!

TV chef Emeril is always "kicking things up a notch" by adding garlic and other zestiness to his recipes. How are your firm's engineering recipes? Want to kick up your development processes by more than a notch? I can present my Better Firmware Faster class at your facility. See http://www.ganssle.com/classes.htm .

Wil Blake sent a great link to a source of free books about technology. No kidding: free. See http://www.freetechbooks.com/ .

I've long been interested in how we approximate functions, and have had a paper about using them to form trig functions online for some time (http://www.ganssle.com/approx/approx.pdf). Now I've added info about approximating logs, roots and exponentiation. See http://www.ganssle.com/approx/approx-2.pdf .

Reading Code

I have a beef with many college computer science departments. If one wanted to be a great novelist the English Department would require that this person knew how to read, and would have insured he'd read many great novels before starting one of his own. The Music Department would assume an aspiring composer had listened to lots and lots of music, and read many scores. Yet CS Departments take a different tack: Here's how a for loop works. Your assignment: write some code. As a result most graduates are forced to invent their own styles and adopt some half-baked (if any) process.

Programming students should read great code. A lot of it. Professional developers should read great code. A lot of it. We can and must be constantly learning.

But where can one get truly great code to read? I've long admired Jean Labrosse's uC/OS-II, a real-time OS whose source is on a CD that comes with the book MicroC/OS-II: The Real Time Kernel. That code is stunning in its beauty. It's clearly very well-crafted. In fact, the latest version can be part of a product certified to DO-178B Level A, the highest safety-critical level for avionics. It's beautiful and it works; two attributes that often go hand-in-hand.

Now he has released a companion TCP/IP stack. The source is available for a 45 day trial (signup required) from http://www.micrium.com/products/tcp-ip/tcp-ip_download.html . Check out the source code. It, too, is stunning and even inspirational.

At http://www.ddj.com/blog/debugblog/archives/2006/09/five_questions_1.html Gerald Weinberg professes surprise that we continue to try and test quality into our systems, rather than design it in from the outset. Writing beautiful code is an important part of building quality firmware.

Debouncing

Mark Dobrosielski sent in an implementation of a debouncing algorithm taken from the ideas in http://www.ganssle.com/debouncing.pdf: At home, I'm working on a small PIC-based gadget (PIC 12F629) for one of the handicapped kids for whom my wife provides Physical Therapy. I liked the Alternative Software Debouncer on page 21.

Anyway, here's my implementation of that algorithm in PIC assembly. The first section does the debouncing. The next two sections check for (debounced) off-to-on and on-to-off transitions. The last section takes care of things between transitions.

```
SwitchDB:
  bcf  STATUS, C          ; clear the carry bit
  btfsc GPIO, SWITCH      ; if the switch bit = 1,
  bsf  STATUS, C          ; set the carry bit
  rlf  SW_DB_STATE, F     ; SW_DB_STATE = (SW_DB_STATE << 1) | switch
bit
  movlw #0xE0             ;
  iorwf SW_DB_STATE, F    ; SW_DB_STATE |= 0xE0

SW_On_Edge:
  movlw #0xF0             ; SW_DB_STATE == 0xF0? If so, it means
  subwf SW_DB_STATE, W    ; that SW has been pressed for 5 samples.
  btfss STATUS, Z         ;
  goto SW_Off_Edge        ; If not, check for an 'Off' edge
  bsf  SW, 1              ; If so, set the 'On Edge' bit in SW
  bsf  SW, 0              ; Also, set the 'Level' bit in SW
  return                  ;

SW_Off_Edge:
  movlw #0xEF             ; SW_DB_STATE == 0xEF? If so, it means
  subwf SW_DB_STATE, W    ; that SW has been released for 4 samples.
  btfss STATUS, Z         ;
  goto SW_No_Edge         ; If not, handle the 'No Edge' case
  bsf  SW, 2              ; If so, set the 'Off Edge' bit in SW
  bcf  SW, 0              ; Also, clear the 'Level' bit in SW
  return                  ;

SW_No_Edge:
  bcf  SW, 1              ; clear the "On Edge" and 'Off Edge' bits
  bcf  SW, 2              ;
  return                  ;
```

In my project, this routine is called every 10ms by the Timer ISR. The main program looks at SW for switch status. Bits 2:0 contain the useful info:

Bit 2: 'Off' Edge. 1 during sample where an on-to-off transition occurs, 0 otherwise.
Bit 1: 'On' Edge. 1 during sample where an off-to-on transition occurs, 0 otherwise.
Bit 0: 'Level'. 1 after an off-to-on transition, 0 after an on-to-off transition.

I can simplify the code a little bit if I don't need to look at the switch level or if I don't need to detect both edges. In my current project, I need to detect both edges but not the level, so I deleted the lines that set and clear SW bit 0. SwitchDB can be implemented in 23 instructions to look at everything (shown), 21 instructions to look at the edges only, and 14 instructions to look at only one of the edges (and only 12 instructions if you have the app clear the edge bit in SW). Looks like the guy who can do this in 11 lines of 8051 assembly still has me beat, though!

Preserving Design Decisions

Last issue I asked: What do you think? How can we pass the critical error messages created by our exception handlers into useful customer artifacts?

Dan Norstedt replied: Now, that is a simple one: Embed them in the source code, and extract them mechanically from it. The thing you have to do is to create or set up the tool first. Works like a charm. A major system I architected basically uses the format "err_return(xxx_ERR+nn); /*= <Error message text> */" (multiple lines are allowed, as are some other variations), err_return() and xxx_ERR are defined in a common header file and nn the error code within the system. There are also some rules on how to handle system calls that return unexpected error codes and other inheritance problems, but still the tool has only taken a day or two in coding and support over a 15 year period... And the support department can look up and pinpoint errors like no other system I'm aware of.

Sterling Eanes commented: Our coding rules require that we put all error messages through a single HandleError routine that takes care of reporting the error. Each HandleError call is passed a unique code error code #defined in a #include file, for example:

```
    #define     Err_BadInputFilename     101
    #define     Err_TooManyIterations    102
```

The include file that defines the error codes thus becomes the focal point for the documentation of the meaning of the error. Our naming conventions also make the defined name itself carry at least part of the meaning of the error. And since each code must be defined in this one file, this becomes the place for comments amplifying the meaning of the error. This #include file is an integral part of the source code, so it can't get lost (assuming reasonable SW CM).

From Paul E. Bennett: We will all fall foul of this problem at some time or other. There are ways, I believe, to minimise such problems occurring.

It has been my practice, as I mostly write in Forth and Assembler, to begin by describing the required function of each word before I write one scrap of code. I hardly ever start at the very top for software design. Instead I pull out the major functionality and try and see what I need to implement some aspect of that first. I can usually come up with a suitable name for the required word and a descriptive glossary text with stack behaveour notation. With this level of comment, perhaps down a couple of levels of detail, I can go to review with just the comments to see if the idea fits the system structure. It doesn't take long to explore similarities to words that exist in the library (good for the re-use aspects) and where more effort may be required to solve particular thorny problems.

The maxim of writing the code’s comments first has held me in quite good stead. It yields less surprises and with suitable documentation capture tools, to copy the marked commentaries into another file, the technical authors get a great deal handed to them on a plate.

Of course, for the most part, I am working from well defined system requirements documents where desired error reporting is already well established. The type of commentary that gets included in the glossary text in the software sources may even be tabulated text copies of the requirements error code tables; which makes cross checking easier in reviews. The code functional review ensures that the code carries out the stated activity in the glossary text (which is used a a mini-requirements statement).

Ed VanderPloeg (http://www.agile.bc.ca) had this to say: I've worked with a wonderful solution for the capture of error messages and their meaning: We kept all message documentation in the source code, and made a script to extract the documentation and convert it to webpages on the company intranet. In the mid-90's I worked for a company that manufactured large, complicated pieces of high-tech machinery powered by custom embedded control electronics and firmware. The user interface was a PC equipped with high-level control software. Status and error messages were passed along from the firmware as both a numeric error code and a brief sentence or two of text. There must have been hundreds of these messages coming from various parts of the firmware, with little to no documentation as to their meaning. The firmware engineers made a few periodic attempts to write it all down in a document or two, but keeping this up to date as the firmware changed was impossible. Even hiring professional full-time technical writers didn't help.

Our customers and our service engineers demanded something better, and we firmware folks were getting tired of explaining the messages over the phone to other people, so we designed our own solution. We recognized that separate documents were doomed to failure, whether written by the engineers or tech writers. We also recognized that tech writers like to live in their docs, and firmware engineers prefer to live in their code. Finally, we had to admit that nobody had the discipline to constantly communicate the meanings of these messages, and their changes, to the interested parties. Therefore, the best place to document these messages was in the code and an automated system of extracting the messages was required.

Our solution was to move any message the user might see into separate files (which was good practice anyway). All such messages would be in these files, and only such messages. Each message entry was prefixed by a simple custom-designed comment block, with one-sided tags to identify the various fields. Each message comment block explained when the error occurs, what to do about it, which firmware versions were applicable, etc. We needed field identifiers that could be read by a script, yet we didn't want the added complexity and potential distractions of tags such as html. A Perl script ran nightly to extract the messages and their meanings, and converted it all into easy-to-navigate html webpages. The tech writers extracted their required information from these pages, and the service engineers could log into the intranet from anywhere on the planet to help them figure things out. The nightly script also reported on which errors codes were new, which had changed, etc. Finally, any errors or missing information was flagged by the nightly script and the results were emailed to the firmware team members.

This system was very effective because the code couldn't even be built without all errors messages getting at least minimal documentation. It also let the engineers update the explanations as part of their daily routine without requiring them to take their mind off the code. Finally, the number of phone calls to the engineers was greatly reduced and everyone on the receiving end got their answers much sooner.

Keys to the success of this system included: - admitting our human weaknesses towards maintaining decoupled documentation - clear and consistent commenting standards for all outgoing messages - a field identifier system that was as simple as possible so as to not distract the engineers from the content of the documentation - getting a computer to do the mundane work - a smart script that not only created useful output but flagged any errors or omissions in order to drive immediate correction.

Tools

Dan Norstedt also had a tool suggestion: You may also want to check out metapad from http://www.liquidninja.com/metapad which is only less than half the size of Notepad2 and more true to the Notepad superset idea. Doesn't have syntax highlighting, and if you think a Notepad replacement shouldn't have that (as I do, I use a real editor for source code browsing/editing), metapad is probably more to your liking. I've used metapad as a notepad.exe replacement (replaced the notepad.exe executable) for more than 5 years on all my systems with absolutely zero problems.

From Tim McDonough: You often mention favorite editors. My personal pick is the Crimson Editor available at <http://www.crimsoneditor.com/>. It has column mode, syntax highlighting, spell checking, project management, etc. Definitely worth a try for all the editor junkies out there.

Simon Large contributed this: If you want a short and sweet (and free) replacement for the Windows standard Notepad, try Notepad2 from http://www.flos-freeware.ch/notepad2.html. At 250k bytes it is bigger than Notepad, but still small enough to load quickly. Unlike Notepad it handles Unix line-endings (and Unicode/UTF-8), has syntax highlighting for quite a variety of formats, and generally behaves much better than the MS default tool.

From Philip O. Martel: My current favorite is Notepad++. http://sourceforge.net/projects/notepad-plus/ they sound pretty similar. NP++ also allows having multiple files open.

Another tip for those who haven't found it yet is to add Notepad (or your favourite text editor) to the Send To list. Just create a shortcut to the editor in Documents and Settings\{USER}\SendTo. That way you can view any document type as text directly from a right click.

You can also type "sendto" into the run box in the start menu to open up that directory.

Jobs!

Let me know if you're hiring firmware or embedded designers. No recruiters please, and I reserve the right to edit ads to fit the format and intents of this newsletter.

QUALCOMM in San Diego, California is seeking 20+ SW Engineers for our Corporate R&D division (embedded, FW, DSP, Wireless Protocols), ranging from college new grads to highly experienced engineers.

Here is one of our openings:

Embedded Software Engineer Division - Corp R&D Software Requisition # E1702230

Role - This person will be a member of a team of engineers designing and building modem software for a cellular base station modem. This will include the development of both DSP and host processor software.

Skills/Experience - Demonstrated ability to develop embedded software is a must. - Experience with C is required.

Additional Skills - Recent experience implementing wireless modem software is preferred. - Recent experience implementing CDMA (1X, EVDO) or WCDMA modem software is a plus. - Experience with C++ and DSP assembly language is preferred.

Education Requirements - Master's or Bachelor's and five years of experience is required.

Please apply by emailing your resume to daveh@qualcomm.com and/or by applying online at https://jobs.qualcomm.com

Plantronics is looking for a firmware engineer in its Santa Cruz, CA headquarters. Send letter of introduction and resume to fred.sarkissian@plantronics.com.

Responsibilities: Implementation of current DECT/UPCS based products. Development of next-gen DECT/UPCS products.

Experience required:
- 5 years industry experience, at least 2 years wireless communications (DECT/cellular) systems
- Experience in high volume production under tight cost-control and aggressive schedules
- BS degree in Computer Engineering or equivalent
- Strong C skills, embedded real-time development, written and communications skills

Experience desired:
- Technical experience with wireless telephony products
- Experience with headsets (corded or cordless)
- Experience with mobile, small, battery-powered systems (mobile-phones, pagers, cordless phones)
- Voice on DECT or 802.11 standards
- Familiar with test equipment

Joke for the Week

.
From a recent issue of EE Times:

What's the difference between a large pizza and an engineer?

A large pizza can feed a family of 4.

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*". ". BUT - please use YOUR email address in place of "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.