The Embedded Muse 47

Editor: Jack Ganssle (jack@ganssle.com) March 22, 2000

Conspiracy Theory, Take 2

Last issue I wrote about RTOSes, and how (in my opinion!) they are a very powerful tool for building firmware. This prompted an astonishing number of very interesting responses, some of which I'd like to share with you.

But first – some readers got the impression I think an RTOS is the silver bullet required for all embedded apps. No way. This industry is completely fragmented, so it's impossible to generalize about anything, generally. An RTOS offers some serious benefits for a lot of applications, but is totally inappropriate for many. One reader wrote about a microprocessor-controlled soldering iron. I can't imagine how an RTOS would benefit something so simple!

Further, commercial RTOSes address (again, in my opinion) a wide range of RTOS needs. But there's little question that most share the same few scheduling and other mechanisms. Some systems require very different techniques, which may not be addressed by off-the-shelf products.

The responses were about 50% pro- and 50% anti- coding your own RTOS. In the pro-custom-RTOS camp, the most common reason was mistrust of vendors. Developers worry that buying code means buying bugs, bugs that they'll be left to sweat out at 2AM. A custom RTOS means you have the source, and you know how the beastie works, so bugs, if they appear, can be swatted effectively.

I hope the commercial RTOS people are reading and learning…

So, on with the letters. And thanks much to everyone who wrote in.

Michael Purcell wrote:
Your remarks in the latest Muse regarding buying an RTOS are 100% on the mark. The "conspiracy" comment is an example of a violation of something I call "Evans' Law":

> Just because you *can* do something
> does not mean you *should*!

(Named after a fellow software engineer who used the axiom to challenge us when we were about to do something stupid.)

Brian Trial commented:
Well good god yes! Labrosse's OS supports preemptive multitasking for 63 tasks, floating point math, in a C++ environment. How many embedded applications need this much horsepower? I'd never even think of tackling those requirements without a commercial RTOS.

How many embedded applications are written in assembler or plain Jane C, on an 8 bit microcontroller, with fixed point math and no requirement for complex multitasking? I'll bet it's more than 2%, and any competent software engineer could certainly construct their own OS in this environment far cheaper than any commercial RTOS, and tailored to their specific needs.

We recently completed a project using the Motorola HC12. Our group found 3 bugs in the C compiler we used, which ate up a lot of our time tracking down. I shudder to think if we had to debug someone's buggy RTOS as well! The OS we wrote is simple and streamlined for our needs, and didn't take near a $60K effort or 4Klocs of C you mention.

Now, our next project will probably require a more complex 32 bit micro with more complex tasks to perform, so naturally we're considering an RTOS for that. But I'd say closer to 2% than 70% of the embedded world needs a 32 bit processor.

Sid King wrote:
We have gone through the agony of starting with Jean Labrosse’s port of UCOS (A great little OS I should add) for the XA from Philips and encountered many challenges just getting that working correctly. (The original UCOS Hi-tech 'C' port worked fine, using a new compiler was the challenge.) We probably invested over $50K in getting UCOS working with the Tasking Compiler. I might also add that we spent part of that $50K making the UCOS work with the CrossView debugger from Tasking, we had to do some rewriting of the monitor code to make it function with the OS. Our next project will be using the UCOS port for the XA that we did, as well as buying ThreadX for use on the Coldfire processor in that system as well. We will be using the GreenHills tool set for that. It will be interesting to see if ThreadX is much easier to work with since no porting should be required.

Another issue that should be addressed in selecting the RTOS, is making certain that the compiler/debugger and RTOS vendor work closely to make the port as seamless as possible. We are banking on that with the GreenHills/ThreadX development combination.

Brian Lim thought:
On your subject of why projects use custom operating systems, I offer my thoughts on factors that lead to this outcome.

1. Many people get jobs writing software because they enjoy writing software. The more they write, the more enjoyment.

2. While marketing people are responsible for achieving maximal profit from minimal expenditure, they often don't see to it that engineering has those same aims.

3. Normally you need to get people in an organization to sign a purchase requisition for things that you want to spend money on. You would have to justify the cost. You probably don't need signed purchase requisitions for bloated feature sets. A custom OS is part of the same problem.

4. There's risk involved in selecting and buying an OS. There's risk involved even if the OS is free of charge. Some engineers may be excessively risk averse in this area.

5. As with everything, it's mostly management's fault

Mark Bereit wrote:
This isn't an argument, necessarily, but my personal explanation for why I'm in the code-your-own-RTOS camp. (I'll bet I'm not the only one of these you get!)

First, a story. When wearing my programmer hat I frequently put structures in a linked list.

Aha, says the School of Code Reuse. A simple thing which should be coded once, and used endlessly without fussing the details. After all, each time you recode the routines you introduce the chance for a mistake, different naming conventions and style conventions which make no difference but have to be maintained, etc. etc. All very sound reasoning.

So how should I reuse a linked list? In C++ I could make a class which is just a linked list implementation, and then derive things from it. Oh, wait, that won't work for an object derived from something existing (like a Microsoft MFC class) because now it's multiple inheritance and that often breaks things (assuming you have multiple inheritance in your language in the first place) and complicates pointers and adds overhead. I can use a template... well, that confuses things similarly. And neither derivation works in C. I can make a linked list structure which is the first member of the larger structure, but this similarly breaks complex derivations and requires unsafe recasts. I can make a bunch of macros to do define the extra members and to do the adds, removes, etc.... although I lose all type safety with macros and when there's a bug the macro obscures what you did wrong. I can inline function some of it and macro some of it... but I still need to remember to do all the pieces...

Suppose I pick one of these half-complete solutions. Now, what about the head pointer? Is it a global, or a member of something else? How does that impact the implementation? Did my approach allow the maintenance of this list to be a member of a class, or is it required to be a global so I have to rewrite to make it fit? Who does cleanup? The owner of the pointer, but what makes THAT automatic? Should the owner of the head pointer do the cleanup? Should I have required a dummy first node, like I learned in school? What if the nodes are large and resource intensive? How well does this solution fit ME?

And so I have learned over the years to just Code The Stupid Thing. Every time. By now I rarely get it wrong the first try.

So we come back to the RTOS. There are plenty of choices. I see no reason to hate any of the options available, and if one were handed to me as a design requirement I would have no problem with starting there. But again, if I had to choose one, I would be seriously gridlocked in evaluating what is out there and what works well with my tool chain and my expectations and my project requirements and so on. And whatever was chosen, I would then proceed to use where possible and circumvent where insufficient... meaning that again, the code would only be partial reuse and partial replacement.

On some of my projects the dispatch mechanism is a while(1) loop which does the highest priority thing, then the next highest, then the next highest, etc., in each case repeating the loop without touching lower priority tasks if there is more to do on that priority. No reentrancy, no ugliness, and this can be interrupt-based or completely polled depending upon hardware. So I invariably divide all RTOS needs into two categories: those where what I just described is adequate, and those where it is not. For those where it is adequate I will code just that: I don't need threads, synchronization, or anything! What RTOS did I just use? Unless "none" is an option on the survey--pretty rare--my answer must be "custom made." Nothing else was even a consideration... because NO RTOS WAS NECESSARY.

Now in the other half, it comes down to, how much RTOS do I need? I find that a preemptive task switcher with critical sections is pretty easy to write for any platform, so that need won't push me to buy. If I need complex synchronization objects, that may call for outside help. If I need a TCP/IP stack, that definitely calls for outside help. (My current project uses USNET from US Software, with which we are pretty happy... but we didn't buy their RTOS, we bolted USNET quite nicely onto our own task switcher.) If I need thread-safe resource allocation, especially memory allocation, then if that isn't part of my C Runtime then I need outside help.

But when the need for outside help isn't strong, then I find that I am weighing the energy of coding and debugging my own code (for which I am being paid to accomplish) against the energy of researching and learning and debugging someone else's code (for which I additionally have to request an expenditure beyond existing compensation). So I won't buy an RTOS until the energy expended tradeoff makes sense.

Although I am not opposed to doing so, I have never yet bought an RTOS. I have bought a TCP/IP stack, and expect to buy an SNMP implementation in the future, because there is too much code that is new to me: I would be writing and debugging for too long to justify that tradeoff.

How much of that 70% of custom-made OSes represent the trivial dispatch loop approach I couldn't say. It represents almost 50% of projects I've worked on, though! I suspect it is only the RTOSes that offer a lot more than simple task switching who have any appreciable following, and then only for the projects where the needs are a lot greater than that.

Bottom line: if I can build and debug it in less than a week I am unlikely to buy it. Because that's how long I expect to spend learning, adapting and debugging code I buy from someone else.

Thought for the Week

From Paul Walker

"Source Code to Windows 2000"

```
/*  Source Code to Windows 2000   */

 #include "win31.h"
 #include "win95.h"
 #include "win98.h"
 #include "workst~1.h"
 #include "evenmore.h"
 #include "oldstuff.h"
 #include "billrulz.h"
 #include "monopoly.h"
 #define INSTALL = HARD

 char make_prog_look_bi  g[1600000];
 void main()
 {
```

```
while(!CRASHED)
{
    display_copyright_message();
    display_bill_rules_message();
    do_nothing_loop();
    if (first_time  _installation)
    {
       make_50_megabyte_swapfile();
       do_nothing_loop();
       totally_screw_up_HPFS_file_system();
       search_and_destroy_the_rest_of_OS/2();
        make_futile_attempt_to_damage_Linux();
       disable_Netscape();
       disable_RealPlayer();
       disable_Lotus_Products();
       hang_system();
    }

    wri te_something(anything);
    display_copyright_message();
    do_nothing_loop();
    do_some_stuff();

    if (still_not_crashed)
    {
        display_copyright_message();
        do_nothing_loop();
        basically_run_windows_3.1();
        do_nothing_loop();
        do_nothing_loop();
    }
}

if (detect_cache())
     disable_cache();

if (fast_cpu())
{
     set_wait_states(lots);
     set_mouse(speed, very_slow);
     set_mouse(action, jumpy);
     set_mouse(reaction, sometimes);
}
```

```
/* printf("Welcome to Windows 3.1");    */
/* printf("Welcome to Windows 3.11");   */
/* printf("Welcome to Windows 95");     */
/* printf("Welcome to Windows NT 3.0"); */
 /* printf("Welcome to Windows 98");     */
/* printf("Welcome to Windows NT 4.0"); */
printf("Welcome to Windows 2000");

if (system_ok())
    crash(to_dos_prompt)
else
     system_memory = open("a:  \swp0001.swp");

while(something)
{
    sleep(5);
    get_user_input();
    sleep(5);
    act_on_user_input();
    sleep(5);
}
create_general_protection_fault();
```

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.