The Embedded Muse 85

Editor: Jack Ganssle (jack@ganssle.com) Nov 6 1003

EDITOR: Jack Ganssle, jack@ganssle.com

CONTENTS:

Editor's Notes

A couple of times a month I give my day-long Better Firmware Faster seminar to companies, on-site at their facility. It's always fun to discuss issues important to a particular group. Last week one attendee asked "what's the most important tool a developer needs?". Well, that's sort of like asking a mechanic the same question; he'd be helpless without a box chock full of tools. Is a wrench more important than a screwdriver? Most of the class tossed out ideas like editors, debuggers, lint, and the like.

Those are all great answers. Maybe a better one, though, is a version control system. We shouldn't even start to think about programming until a decent, reliable VCS is installed, with a corresponding backup strategy. The VCS is the nexus around which all of our development work occurs; we check in and check out files as needed. It makes collaboration possible and seamless.

This is a nice – and short - page with links to various VCS products:
http://www.codeorganizer.com/version_control/tools.htm

For a more complete set of links to configuration management tools see:
http://kmh.yeungnam-c.ac.kr/comScience/se/scm-links.html

This issue of The Embedded Muse is going out on the eve of the California recall election. You might be interested in an article I wrote about electronic election machines, and the associated poll, which asks developers to speculate on who will win this election. Check it out at http://www.embedded.com. As of this writing developers seem to feel Arnold has the edge.

Check out Bob Paddocks' safety critical software site – there are a lot of great links there: http://www.softwaresafety.net/.

Software WILL Fail

In the last issue (TEM 84) I wrote about insuring our code fails safely, or that we can recover from failure. We want to build bug-free systems (how to do so is a different topic), but bugs will creep in, especially as our products grow in size. What can we do to handle those bugs in a customer friendly manner? And, secondarily, how can we trap the problems early, leaving debugging breadcrumbs behind?

Lots and lots of people wrote in with ideas. Thanks much to everyone. If I transcribed each email here the Muse would be far too long. So here's a summary of your ideas. I'm sorry I couldn't credit everyone. But the embedded community – and I – thank you for your thoughts.

In an effort to provide a more or less comprehensive source of ideas, a few are listed that also appeared in Muse 84.

Use a safe language! C and C++ are messes that allow us far too much freedom to create flawed products. Ada eliminated most of these problems… but few projects use Ada anymore. Cyclone, an x86 compiler hosted under Linux (http://www.research.att.com/projects/cyclone/) is a variant of C that promotes inherently safe programs. As far as I know, as yet it's not appropriate for most embedded systems due to lack of ports. But what a great idea! These alternative languages and dialects promote safety in a variety of ways, not the least of which is built-in runtime exception checking.

Consider a layer of "middleware" to capture dumb stuff the program does. One of the intriguing aspects of Java is the virtual machine, which does runtime error checking (among other things). Ed Sutter's free MicroMonitor is an execution environment that includes a flash file system, networking and more. It also captures crashes and has a scripting language that let's you program appropriate actions in case of a failure (http://www-out.bell-labs.com/project/micromonitor/). Also see his excellent book (Embedded Systems Firmware Demystified).

Use a dynamite Watchdog Timer to reset crashed programs, and to signal to the user, or the developer, that a problem was found. Eric Krieg and others warn against kicking the dog in an ISR. He, Bob Landman, and one or two others gave stories about spending too much debug time finding code that crashed simply because it does not kick the WDT often enough. I’d argue that’s an indictment of the system design rather than a problem with watchdogs, but it’s something to look out for. Perrie Matthews has a neat trick that lets him safely service a WDT inside an ISR: “I always use a two-level scheme for servicing a WDT. Since I don't like to sprinkle watchdog accesses all over my code, and I don't like having to make sure that I count cycles in my code so that I don't inadvertently add too many lines between WDT services, I have an interrupt service routine that services the WDT at a high enough rate, but only if a flag has been set by the foreground code. If the flag is not set, the ISR will continue to service the WDT for only about 10 seconds (or whatever time the designer chooses), at which time it will quit servicing the WDT and allow it to reset the CPU. The ISR clears the flag whenever it detects the flag is set. So all I need to do is set the flag periodically in my foreground code, usually at only one place in the main executive loop, or in other long loops such as polling for serial input, etc. Then if the foreground code gets stupid, it will stop setting the flag, and the background will eventually give up and let the WDT do its thing to bring the system back on track.” Other people contributed other ideas for watchdogs I can’t really put in here. Though I’ve written about WDTs in the past (see http://embedded.com/2003/0301/, http://embedded.com/2003/0302/, and http://embedded.com/2003/0303/), more is coming, with sample code, shortly. Stay tuned!

Implement exception handlers… and test each one extensively. Something like 2/3 of all system crashes can be traced to unimplemented or poorly tested exception handlers.
Use the memory management unit, if your CPU has one, to isolate tasks from each other. A task crash will throw an exception; the code can recover from the problem or at least log debugging info.

Reset output ports frequently. This suggestion prompted a lot of email. Some folks complained their hardware will die horribly if a port assumes an incorrect value for even a microsecond. It’s hard to generalize about embedded systems as they come in so many flavors; use these ideas only if appropriate for your system. Other people wondered about the nature of ESD-generated port transients. To my knowledge there’s nothing published on this, but many readers noted that they, too, have seen similar issues. It seems that an ESD transient sometimes just resets the output latch in the processor. A read-back will most often show the latch in a bad state. Mat Bennion suggests flagging an error if the read-back shows the ports flipped.

Apply sanity checks on any input which might possibly be incorrect. 50 years of software engineering has taught us to bound inputs to functions… yet we don’t. In the embedded world, don’t forget to check hardware inputs, like those from an A/D, for values that are “impossible”. (Rule of thumb: When we say “impossible” we often really mean “unexpected”. Expect the unexpected.)

Jack Marshall suggests measuring execution time of tasks. In one project he found (and he's not alone here!) a task which should have taken 30 usec actually burned more like a msec. That indicates either a bug, or a lack of understanding about the environment. Either is a problem.

I've always advocated filling unused memory with single byte or one-word software interrupts. Crashes often cause the code to wander off; the interrupt, plus exception handler, can capture the crash and take action. Craig Yerby offers an alternative: fill unused memory with NOPs. At the end of memory put a jump to the diagnostic routine. Not a bad idea, especially for CPUs without a single-byte software interrupt.
Charles Bates is fascinated with built-in tests, and warns people to implement loop-back checking wherever possible. Doing a comm link? Why not have a software-controlled loop-back that lets the code test the hardware?

Set all unused interrupt vectors to point to an exception handler. All sorts of problems (hardware glitch, misprogrammed vector or peripheral, software crash) can create an interrupt one did not expect. This is an easy and cheap way to capture such an event.
Always think through possible overflows in arithmetic calculations, especially with integer calcs. Michael Greenspan notes that adding two large positive numbers can overflow, resulting in a negative. C, of course, is perfectly happy to return this meaningless computation. He writes assembly macros that return, for a 16 bit computation, 32767 when there's an overflow. The number is not a correct result, but it's more correct than a negative value.

Bob Paddock and numerous others want you to put constants on the left side of an equation when doing compares. Mix up = and ==, and the compiler will issue a warning.
Paul Walker stresses the importance of looking at the return values from functions, so the caller can tell if the function worked. Good point! For example, I *never* see malloc's return value tested to see if the allocation actually worked. We just assume everything will be OK.

Phil Koopman passed along a nice paper which gives a series of problems to look out for, along with a mnemonic to remember them. Check out http://citeseer.nj.nec.com/maxion98improving.html; it's quite thought-provoking.
Why not use assert()s? We don't, yet the assert() macro is a powerful way to detect all sorts of errors. Max van Rooij wants to take it a step further. Create an "ensure" macro like:

```
void Ensure(bool_t condition, jmp_buf jb)
{    if (condition == FALSE)
    {  MarkInternalError();
       longjmp(jb,-1);
    }
}
```

He prefers this over the ASSERT() since the ASSERT() can be switched off. He gives the jump_buf as a local var instead of a global one, because he can then specify more then one return point in the code.

It's important to check the stack size. The traditional method is to fill each stack with 0x55AA, run the program for a while, and then use a debugger to see how many of these are left. Max has an alternative: create a chunk of code you examine in each ISR for stack growth, something like: U16_t get_stack_bytes_left() { _asm mov ax, sp; }. (That's clearly x86 and perhaps compiler-specific, but you get the idea). This really appeals to me – it costs a few microseconds and some memory, but can be left in production versions to monitor for "unexpected events".

Chris Gates offered numerous suggestions. For instance, never use checksums; CRCs are better and almost as easy to code.

Chris also mentioned having the software confirm hardware actions. Example: after closing a relay, use an A/D to monitor something to be sure the contacts are really closed. Backing up data into nonvolatile memory? Use a CRC or duplicate memory area to guarantee the data is not corrupt.

Bandit wrote at length about managing data types. He said: By placing a type as the first field in the struct, you have self-describing data. This has a few implications:

1. If you send the struct as a message, you know how to crack the rest of the message – i.e., it is an explicit message type. I had a gig where we took the RPC tool that lets you create byte-order independent structs and modified it to automatically create these structs with the types auto-generated.

2. If you have the case:

```
typedef struct
{
    UINT16    type;
} S;

union
{
   S       s;
   FOO     foo;
   BAR     bar;
   BLETCH  bletch;
   UINT8   data[ MAX_DATA_SIZE ];
} u;
```

If you read in the message data (or pick it up from some source) all you need to is read in the first 2 bytes into u.data[] (byte-order is assumed known) or if there is a size associated with the read, then your code can look like:

```
switch( u.s.type )
{
case TYPE_FOO:
      ... crack u.foo  ...
      break;
case TYPE_BAR:
      ... crack u.bar  ...
      break;
case TYPE_BLETCH:
      .... crack u.bletch ...
      break;
}
```

3. If you want to have "persistent objects" where the object is your data, then it is easy to walk the data structs, with one function pair (write/read) per struct, and using some form of non-volatile memory (i.e., a text file) for persistence.

4. buffer overrun is always an issue. A simple way to deal with this is to create:

```
typedef struct
{
    UINT16   type;   // TYPE_STR
    int      len;    // number of bytes in the buffer
    int      used;   // needed if a data buff and not ASCII
string
    UINT8   *buf;    // pointer to a fixed buffer
} STR;
```

Create a buffer pool that *buf points to, and set the pointer when you "instantiate" the STR variable. You can even do bounds checking on *buf to make sure it points into the buffer pool.

You would need to create a replacement set of functions for sprintf(), memcpy, strlen(), strcpy(), etc. However, these are fairly simple to do.

Joke for the Week

The software engineering community has been placing a great deal of emphasis on metrics and their use in software development. Patti Beadles wrote the following metrics which are probably among the most valuable for a software project:

The Pizza Metric
Count the number of pizza boxes in the lab. Measures the amount of schedule under-estimation. If people are spending enough after-hours time working on the project that they need to have meals delivered to the office, then there has obviously been a mis-estimation somewhere.

The Aspirin Metric
Maintain a centrally located aspirin bottle for use by the team. At the beginning and end of each month, count the number of aspirin remaining aspirin in the bottle. Measures stress suffered by the team during the project. This most likely indicates poor project design in the early phases, which causes over-expenditure of effort later on. In the early phases, high aspirin-usage probably indicates that the product's goals or other parameters were poorly defined.

The Beer Metric
Invite the team to a beer bash each Friday. Record the total bar bill. Closely related to the Aspirin Metric, the Beer Metric measures the frustration level of the team. Among other things, this may indicate that the technical challenge is more difficult than anticipated.

The Creeping Feature Metric
Count the number of features added to the project after the design has been signed off, but that were not requested by any requirements definition. This measures schedule slack. If the team has time to add features that are not necessary, then there was too much time allocated to a schedule task.

The Status Report Metric
Count the total number of words dedicated to the project in each engineer's status report. This is a simple way to estimate the smoothness with which the project is running. If things are going well, an item will likely read, "I talked to Fred; the widgets are on schedule." If things are not going as well, it will say, "I finally got in touch with Fred after talking to his phone mail for nine days straight. It appears that the widgets will be delayed due to snow in the Ozarks, which will cause the whoozits schedule to be put on hold until widgets arrive. If the whoozits schedule slips by three weeks, then the entire project is in danger of missing the July deadline."

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.