An Alternate Approach to Avionic Software KISS

Gerry Tyra, Lockheed Martin Aeronautics

Abstract. Driven by customer perceptions of cost, there is a recurring drive in the avionics community to provide overarching software frameworks. It is believed that such a framework will simplify the software development process and promote reuse, hence reducing costs. While such goals may be laudable, this paper presents the argument that one-size-fits-all solutions will not fare well in most real world developments. A minimalist application environment is presented as an alternative. Like the tradeoffs between complex instruction set computers (CISC) and reduced instruction set computers (RISC), software has the option of complex frameworks vs. Keep It Simple...

Introduction

Avionic software development costs too much. Or, at least, it is *perceived* to cost too much. In an ideal world, a statement of work (SOW) is drafted, it is competitively bid, and the winning bidder, who is deemed to be technically competent, completes the work within the schedule and budget.

Unfortunately, we don't live in an ideal world. Talk to any group that has been around long enough to have earned their gray hairs, and you hear the same horror stories. The programs that over run have some combination of the usual suspects: The SOW was vague or had holes in it. The contracting authority didn't have funding to match the scope of the SOW, but didn't reduce the scope. The estimators were too optimistic. The proposal team underbid to win. Various aspects were unrecognized, hence were not factored into the bid at all. The contracting office slipped in additional requirements that were never negotiated. System engineering regurgitated the high level requirements without a high level design, leaving the independent product teams to go their own ways. The design phase was shorted to start producing code. Design decisions were made that range from "It seemed like a good idea at the time", to plain boneheaded. Schedule compression caused coding time to push into what should have been integration time. The integration facilities are unstable, out of configuration management and overbooked. All this before you get to flight test, which has its own issues.

Such is human nature and, so, schedules slip and budgets are overrun.

But, how are programs to overcome the tendency to blow budgets?

The bureaucratic solution is to add more process, structure and oversight. Uniformity will reduce cost and encourage reuse, while constraining "disruptive" activities. So was Ada, and other more recent standards efforts (e.g. FACE, UCI...), born.

Along the way, there has also been the effort to use Commercial-Off-The-Shelf (COTS) solutions. Some of these are still unproven academic exercises. Others are proven commercial products, but designed for specific environments, such as server farms.

When your main tool is a hammer, everything starts to look like a nail. The observant reader will notice that these solutions do little to address the root cause problems.

The alternative that is presented here is to provide a minimalist set of interfaces and a maximum degree of flexibility in implementation. Thereby simplifying design and integration, and providing a code base less subject to error. While this approach does not overcome the root causes, it does try to constrain them.

Requirements and Fallacies:

When your old and wise Computer Science professor tried to teach you "best practices," he was looking at the entire "ecology" of computer applications. But, avionics software is in its own niche. It is not that the general rules don't apply, rather, there is a difference in priorities that drives the software engineer to a different optimum solution. Here are a few most frequently abused realities of real-time.

SWaP

Size, weight and power; on an aircraft, these three are king. Ignore them and your program will fail. While you could build a flying server farm on a C-5 or an AN-124, a blade server attached to a hand launched quadcopter is not going to fly. Every kilogram of computer that you put on board is one less kilogram of fuel or payload available. Every additional watt has to be generated, using fuel, and then the resulting heat has to be dissipated. The bigger the system, the harder it is to fit inside the airframe, which usually implies that it will be harder to get at to maintain.

And, as a reminder to the ground station developer, if your facility is to be deployed to a remote site, the same rules apply. What does it take to transport your system? How many aircraft sorties will it take to supply fuel to the generators powering your system? Or feed the technicians operating it?

Tactical Bandwidth Is More Valuable Than SWaP

In any theater of operation, secure tactical bandwidth is scarce. That one satellite that you want to bounce off of may have a higher value than your entire program. Don't assume that all of the bandwidth is yours; this isn't a local 10 Gb/sec Ethernet connection. The bandwidth that you are allocated may be tiny. Be prepared to live with what you get.

Learning Curves

The more complex the tool set is, the longer it takes to become proficient using it. For a program with a lot of people unfamiliar with the tools, either you have to delay while they learn, or proceed and risk bad design choices that can haunt a program for years.

Services Cost

There is a school of thought that advocates an independent process/service for each defined task. The services are then strung out like a string of pearls. Sometimes, this is required. Sometimes it is useful. But, the process to process communications increases system latency, sometimes to the point of operational system failure.

What is Open?

One recurring theme is to use self-defining interfaces, (e.g., CORBA, XML, UCI, FACE, et.al.) and using virtual machines to mask underlying hardware. The argument is that this will simplify updates and allow swapping components. The reality is that there is nothing "Open" about the avionics system of an aircraft. It is designed, tested and certified as a whole. There are too many examples of what happens when something is updated without testing, "because it was only a minor change." Why pay the overhead for an open system, when it has to be integrated, tested and certified in a closed system environment?

And when the time comes to go off-board, reread the comment on bandwidth above.

Standards and Standardized

Using any standard entails having to carry the baggage that goes with it. Ethernet works. It works well in many applications, and in others, not so well. But it is an ongoing, evolving standard, moving from 10Mb/sec to 100Gb/sec in three decades. On the other hand, Mil-Std-1553 has been basically stagnant over the same period.

Similarly, you can use an Android phone to call a friend with an iPhone, though you can't share apps. The underlying technology is improving constantly, even if your older device can't make use of the newer options and speeds. Then there is Link 16, again a stagnant technology.

Consider the difference between the commercial standards and the military standards. The commercial standards grow because the commercial players are investing in the technology in order to attract customers. The military standards are decreed and there is only one customer. The developers may suggest innovations, but that one customer is in control.

Some standards you want to use, some you are required to implement. One size never fits all. Just because you want to use an interface, doesn't mean that it is available, or can be made available within the scope of your program. Always do a cost/benefit analysis in the context of your application.

Moore's Law Will Not Save You

So, your software is running on the ragged edge of what your hardware will support? The newest hardware will certainly fix it, right?

No, it won't.

Software bloat will eat any hardware improvement[2]. And that assumes that you will get a tech refresh. The reality is that many systems are never updated; only replaced when they reach the end of their service life. Even systems with planned tech refreshes are subject to having those updates delayed by years. Live within your means, as they are. Don't count on a refresh that may never come to save your program.

Abstraction Does Not Help

The argument has been made, repeatedly, that if we work at a higher level of abstraction, we will see tremendous improvements in developer productivity. In some areas this may be true, but for most of avionics, it is not the case.

What isn't covered by a good library function has to be coded by hand. And drawing a lot of UML pictures to describe a function is no less labor intensive than just writing the function (though you may have a more understandable design when you are done). And that simple picture you drew in UML can generate some truly hideous code that you will have to integrate and maintain.

Writing "a = b + c" is more efficient that trying to do it in machine code or assembly language. But burying that same equation in multiple generations of derived C++ classes does not make it easier.

Reuse Is a Myth

Reuse would save time and money, if it worked. Most of the time, the old code was written for a different platform, with a different interface and different requirements. Or, worse, it was slapped together on an IR&D project or an expedient program. In which case, the code is held together with baling wire and chewing gum. You can use such code as a design starting point, but do you really want to live with the maintenance headaches of the code itself? There are two cases for reuse: established libraries, and when you are bringing an existing subsystem into your program without modification.

MLS Is a Trap

Security is important. Losing secrets is bad. Keeping the secrets is expensive. So, Multi-Level Security (MSL) is frequently proposed as a solution. In the author's opinion, it fails for two reasons.

The first is perception of the environment. In a vast networked server environment, there are well defined islands of highly classified data in a sea of less classified/unclassified data. In an airframe, pretty much everything talks to everything else and all the data is needed. Consider all of the interactions that take place among the navigation, sensor, pilot interface and weapons system in order to have a weapon released. The islands and sea have become a swamp, the ground and water are both very muddy. Maintaining functionality with separation is difficult, if not impossible.

The second issue is time. It can take years to get an MLS system certified for operational release. And every update has to be re-certified. Some interfaces, such as Electronic Warfare (EW) can be changing from one mission to the next. If the update is critical to safety of flight, the entire fleet could be grounded until the certification is completed.

It's Not What You Say, Its How You Say It

C and C++ have issues. But, Java has a couple extra problems. One problem is the foot print, referring back to SWaP. The other problem is security, how stable is that virtual machine really? How much control do you have over it for maintenance? Who coded what, where and when?

Java has its uses and is very good in some applications. But, does it belong on your aircraft?

Do. Or do not. There is no try.

When bad things happen, we don't want them to become worse. But, how often are we successful at error recovery versus just paying lip service to it?

As an example, the C++ try/catch pair imposes a cost on computation, but how often does a catch actually do something useful? A print out to the console, when there is no console, doesn't help. Such a catch does nothing to resolve the root cause. If your memory allocator has run out of memory to allocate, the catch can report this. But, there is little it can do to alter your current lack of memory. You have a fundamental problem that should have been found in integration and test. Your application is about to crash. The best that you can hope for is a log file that will be useful back in the lab. But that is for another day.

Another common requirement is to check for a valid pointer being passed into a function. If you spend the CPU cycles verifying that a pointer is valid, what have you gained? An invalid pointer will crash the application. A trapped pointer will result in anomalous behavior (usually a premature return) in the function. Why did you call that function in the first place and what were you expecting it to do? What does not doing the expected imply to the system as a whole?

Now, extend this to error trapping in general. If the trapping does something that keeps the vehicle in the air and on mission, do it. If it only masks the root error and delays an inevitable system reset, why are you doing it?[1]

Power Point Slides Do Not a Design Make

At the Preliminary Design Review (PDR), the design team explains the direction that they are going in. There should be some initial analysis, though not compete, presented to demonstrate that the proposed approach can be made to work.

At the Critical Design Review (CDR), there should be enough solid data presented, along with supporting data, that a new team should be able to come in and take over the effort.

But how many CDRs are buried in pretty Power Point presentations that have no useful data? The design group does it because they had a milestone to meet, but they weren't actually ready for it. Program Management wants to get past the milestone, so it lets the problem slide. The Contracting Office also wants to show progress so it:

1. Overlooks the lack of supporting data.
2. Lacks the internal technical expertise to recognize the problem.
3. Doesn't notice because the presentation is such a snow job.
4. The Contracting Office just rotated in new personnel, they don't have a clue yet.

Keep It Simple S... Problem Space

In the beginning, there was hardware.

If your sizing and timing estimates, with I/O requirements, point to a microcontroller, rejoice. Get a simple development system and use C, perhaps with a little assembly and the standard packages that come with the chip. Don't build a world class super-computer when a single chip will suffice.

However, if you are doing something more complex, such as the Mission Systems suite for an aircraft, you will need a more sophisticated design.

At the simplest level a computer takes input, manipulates it, and provides outputs. If the process uses discrete and/or serial I/O, there is real work to be done. Bit twiddling for I/O is labor intensive and exacting. Similarly, the implementation of algorithms has to be done, state transitions properly defined and implemented, the epitome of "No Silver Bullet[3]." These things take time and effort. The details cannot be abstracted away.

But the environment that they exist in can be simplified.

Except under exceptional requirements, it is recommended to buy, not build. A program should also consider paying to get access to the source code for the OS and BSPs. Even well designed, well supported software has been known to demonstrate obscure bugs, usually late at night and at a remote site. The young engineers might think they can build it better and faster, but the wise manager has probably seen this all before. The OS, boards and BSPs represent man-years of engineering effort and acquired experience. Few programs can afford or justify this type of expenditure to build, maintain and support these items from scratch. After all, the objective is to make the plane fly sooner, not later.

Starting with the obvious; go back to the requirements and start partitioning the problem into functional units. But, maintain logically functional blocks. Don't subdivide just for the sake of subdividing. Then identify a logical set of processes to execute those functions and map those processes to hardware, keeping reasonable performance margins.

Remember that the farther apart two processes are, the greater the bandwidth cost to have them communicate.

Keeping It Simple

The first step towards simplicity is to provide a small, clean, stable framework capable of handling the mundane activities of the processes. The emphasis is on small, constrained and maintainable. There should be little or no middleware between an application and the OS, simply requiring a POSIX compliant OS solves many problems. A few select libraries can be used to abstract tedious activities (e.g., abstract the basic Ethernet or IEEE 1394 interfaces). This will allow the man-hours to be spent on application design, implementation and test, not fighting the middleware and OS.

All applications should be derived from the smallest number of base classes or templates. These base classes operate in a consistent manner and interface with the middleware interfaces consistently. And, consistency is a primary virtue for maintenance.

While the author has his own opinions on how to build a robust mission system, the details exceed the scope of this paper. Interested parties are invited to check out the methodology and sample code at:
http://www.planet-tyra.com/Software/index.html

Conclusion: How Simple is Keeping It Simple?

Start with a good solid design. But recognize that as work progresses, the design will shift and morph. Expect this and allow for it. If your original design was not overburdened with extraneous structures, the evolution should be mostly painless. With too complex a structure, any change is like scratching cut crystal, it is likely to crack and shatter.

This will not save a program from bad requirements or underbidding. But it will save some redesign, refactoring and late nights in the integration lab.

It is not the intent of this paper to dictate a particular approach to implementing software for avionics systems. Rather, it only hopes to show that there are alternative approaches to structuring the required software.

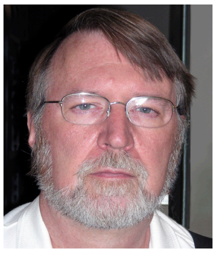
Disclaimer:

This paper presents the opinions of the author and does not represent the means or practices of Lockheed Martin.

REFERENCES

1. Don't confuse test code traps with what is needed in released flight software.
2. Blog by Brian Maccaba, "Why Software Doesn't Follow Moore's Law" http://www.forbes.com/sites/ciocentral/2014/05/19/why-software-doesnt-follow-moores-law/
3. "The Mythical Man Month: Essays on Software Engineering. Anniversary Edition" Frederick P. Brooks Jr. 1995

ABOUT THE AUTHOR



Gerard Tyra is a Sr. Staff Embedded Software Engineer with Lockheed Martin Aeronautics, Advanced Development Programs. He joined LM Aero in 2003. He trained as an Aeronautical Engineer (BS Engineering Science, Purdue University), and he served in the Navy's Civil Engineering Corps. After leaving the Navy, he migrated into real time embedded software while working for Martin Marietta. His work has covered many aspects of sensors and mission systems for submarines, armored vehicles, aircraft, missiles and spacecraft.