By Dr. Adam Kolawa,
CEO,
ParaSoft Corporation.

Techniques for Preventing and Detecting Errors in Embedded Systems

This article explains how error detection, and debugging techniques can dramatically improve embedded software quality and reduce the time and money spent on debugging. The explained practices can easily be applied to many different types of embedded software projects. This means that you can leverage your investment and expertise in these techniques as you move to new projects and different target technologies. The discussed techniques will also help you ensure that your code can easily be maintained, modified, and ported to new types of devices. In short, they will not only help you improve your current embedded applications and development process, but also help you ensure that as new embedded devices become available, you have the expertise needed to develop high quality applications for these technologies-- on time and on budget.

INTRODUCTION

It's no secret among embedded developers that embedded software is difficult to debug. Granted, debugging in general is no picnic, but embedded software poses special challenges for several reasons. For one thing, it is difficult to retrieve data from embedded software. The debugging process relies on output and feedback from an application, but embedded software lacks the print screen that developers of other kinds of software can rely upon.

So embedded developers must bravely seek ways to work around this unfortunate situation. One potential solution is to connect special equipment to the board, which is the hardware for which the embedded software is being written. This special hardware will let the developer see what is happening with the software. For example, there are memory monitors available that will let developers write to a chunk of memory, keep a record of it in the memory, and then come back after the system crashes and try to recover the full record of what happened. Embedded developers can also test with simulators, which are environments that make the program behave the same way it will behave on the board.

Simulators offer several advantages. They typically provide a debugger and allow the developer to print statements, but the inescapable problem is that simulators are merely simulators. A software application may work in the simulator and then die in the real environment. Simulators, then, provide only a partial solution. The fact remains that bugs can slip through the simulator only to surface in the hardware.

That's where the real trouble begins: as Figure 1 illustrates, bugs that slip through the testing phase and surface in the usage phase are significantly more costly to fix. If you find an error in a non-embedded software product, you can fix it and release an upgrade; the cost of these updates and releases is typically relatively small. However, if you find an error in an embedded system, you will need to recall and update the devices in order to fix it. The costs of such a recall can be staggering and can break companies.

I believe that the times and costs of finding and fixing defects are roughly doubled for embedded systems, due to the unique challenges outlined above. In light of this incredible cost, any method that can prevent these errors in the first place is of great value. Fortunately for embedded developers, some recent advancements in software development technology can be applied to preventing errors. The two techniques that are most highly endorsed are coding standards enforcement and unit testing.

Coding standards enforcement and unit testing currently enjoy more respect than usage. Nearly everyone

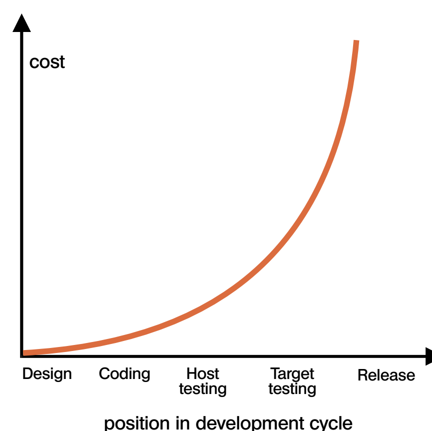

Figure 1. Cost to fix an error in embedded systems software.

Copyright 2001 by Dedicated Systems Magazine - 2001 Q1 (http://www.dedicated-systems.com)

in software development thinks these techniques are great ideas in theory, but few development teams use them. There are two main reasons for this two-faced attitude. First of all, many developers believe it is tedious to enforce coding standards and perform unit testing. I challenge this assertion. Considering the time and cost these techniques can save in the long run, developers should consider enduring a little bit of tedium now to avoid a great deal of tedium--and the possible cancellation of projects--in the long run.

Developers of real-time systems face additional challenges because they must also worry about problems related to time-dependencies. This article will conclude by explaining the difficulties inherent in debugging real-time systems, then by offering some debugging techniques that are geared towards solving the challenges of debugging real-time systems, but which can also be applied to all embedded system software.

ENFORCING CODING STANDARDS

The best way to improve the quality of embedded software-- or any software-- is to prevent as many errors as possible from entering the code.

The first step in preventing errors is realizing that bugs can indeed be prevented. One of the greatest hurdles in keeping bugs under control is the widely-held belief that bugs are inevitable. This is completely false. Errors don't just appear; every error enters code because a developer introduced a defect into the code. Humans are not flawless, so even the best developer will occasionally introduce defects when given the opportunity to do so. The key to preventing errors is thus reducing the opportunity for making errors. One of the best ways to do this is to implement and enforce coding standards that will prevent errors from occurring in the first place.

Coding standards are language-specific "rules" that, if followed, will significantly reduce the opportunity for developers to introduce errors into an application. Coding standards should be enforced as soon as the code is written-- before it is transferred to the target platform-- and they should be implemented in all languages. Because most embedded software developers are working in C, I will focus on C coding standards, but coding standards are also available for other languages, including C++ and Java.

There are generally two types of coding standards:

- **Industry-wide coding standards:** Rules that are accepted as "best practices" by experts in the given language. (For example, "Avoid breaks in for loops").
- **Custom coding standards:** Rules that are specific to a certain development team, project, or even a certain developer. There are three types of custom coding standards that can benefit embedded software developers: company coding standards, personal coding standards, and target-specific coding standards.

Company coding standards are rules that are specific to your company or development team. For example, a rule that enforces a naming convention unique to your company would be a company coding standard.

Personal coding standards are rules that help you prevent your most common errors. Every time you make an error, you should determine why it occurred, then design a personal coding standard that prevents it from reoccurring. For example, if you found that you repeatedly use assignment in if statement condition when you should use equality (e.g., you write if (a=b) when you should write if (a==b)), you could create and enforce the following coding standard: "Avoid assignment in if statement condition."

Target-specific coding standards are rules that flag constructs which will cause problems on a specific target platform. For example, target-specific coding standards could enforce target-specific restrictions on memory usage or variable length.

The best way to explain what coding standards are and how they work is to show you some examples. First let's look at the following C code:

```
char *substring (char string[80], int start_pos, int
length)

{
   .
   .
   .
}
```

This code declares the size of a single dimensional array in an argument declaration. This is dangerous because the C language will pass an array argument as a pointer to the first element in the array, and different invocations of the function may pass array arguments with different sizes. If you or another developer sees this construct, you may use a fixed size buffer of 80 assuming that this is going to be passed to you; this may lead to memory corruption. If the developer of this code had followed the coding standard "Do not declare the size of a single dimensional array in an argument declaration" (taken from one of the leading telecommunication companies' set of C coding standards), the code would read as follows, and these problems would have been avoided.

```
char *substring (char string[], int start_pos, int
length)

{
   .
   .
   .
}
```

Coding standards can also prevent problems that may not surface until code is ported. For example, the following code may work on some platforms, but problems may arise when it is ported to others:

Copyright 2001 by Dedicated Systems Magazine - 2001 Q1 (http://www.dedicated-systems.com)

```
#include

void test(char c) {
   if( 'a' <= c && c <= 'z') { //Violation
   }
   if( islower(c) ) { //OK
   }
   while( 'A' <= c && c <= 'Z') { //Violation
   }
   while( isupper(c) ) { //OK
   }
}
```

The portability problem is caused by character tests that do not use the ctype.h facilities (isalnum, isalpha, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit, tolower, toupper). The ctype.h facilities for character tests and upper-lower conversions are portable across different character code sets, are usually very efficient, and promote international flexibility.

Figure 2. Error prevention techniques prevent costly recalls.

The best way to enforce these and other coding standards is to enforce them automatically, with a technology that enforces a set of meaningful industry-wide coding standards and that offers a way to easily create and enforce custom coding standards. Some things to consider when selecting such a technology include:

- Does it work with your code and/or compiler?
- Does it contain a set of meaningful coding standards?
- Does it let you easily create and enforce custom coding standards (including target-specific coding standards)?
- Is it easy to customize reports to your team and project priorities?
- How well would it integrate into your existing development process?

PERFORMING UNIT TESTING

Often, developers hear about unit testing and think that the term refers to module testing. In other words, developers think that they are performing unit testing when they take a module, or a sub- program that is part of a larger application, and test it. Module testing is important and should certainly be performed, but it is not the technique that I want to concentrate on here. When I use the term "unit testing," I am talking about an even lower-level testing--testing the smallest possible unit of an application while it still sits on the host system; in terms of C, unit testing involves testing a function as soon as it is compiled.

Unit testing can dramatically improve software quality and development process efficiency. When you test at the object level, you are much closer to the methods, and have a much greater chance of designing inputs that actually reach errors and of achieving 100% coverage (see Figure 3). Second, if you test code this soon after it is written, you will not have to wade through layer after layer of errors in order to find and fix a simple error: just find the single, simple error, and that problem is solved. This easier error reduction leads to reduced development time, effort, and cost because less time and resources are consumed finding and fixing errors.

Unit testing can be divided into at least two distinct processes. The first process is black-box testing, the process of discovering functionality problems. When performed at the unit level, black- box testing checks a function's functionality by determining whether or not the function's public interface performs according to

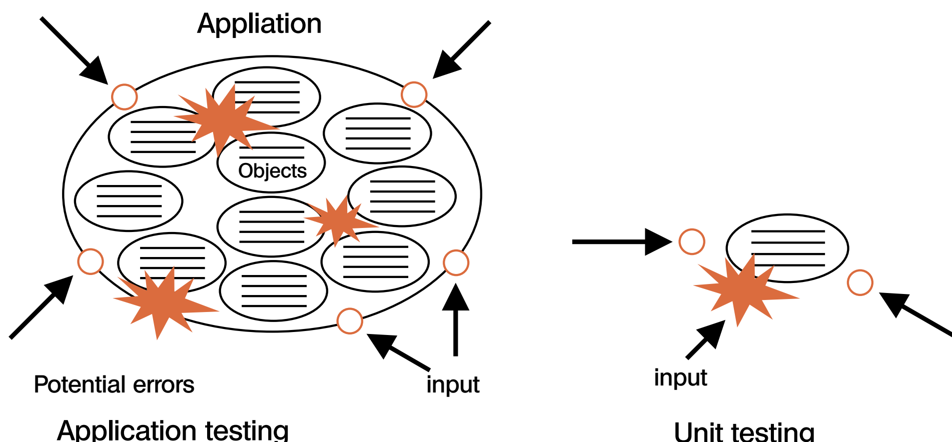

Figure 3. Ease of reaching errors with unit testing.

specification; this type of testing is performed without knowledge of implementation details. Performing black-box testing at the unit level lets you ensure that each function behaves according to specification--before one minor functionality problem spurs a myriad of difficult to find and fix problems.

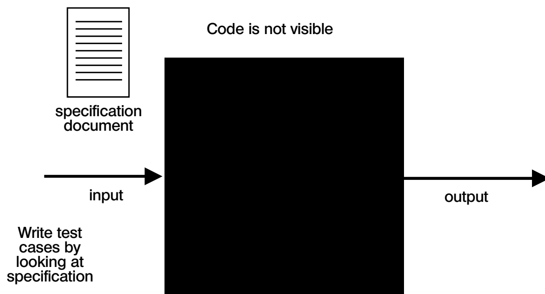

Figure 4. Black box testing.

The second process is white-box testing, the process of discovering construction problems. When performed at the unit level, white-box testing validates that unexpected inputs to a function will not cause the program to crash; this type of testing must be performed by someone with full knowledge of the function's implementation details. By performing white-box testing, you can prevent crash-causing errors and ensure that your functions are robust (i.e., your functions perform well even when faced with unexpected inputs).

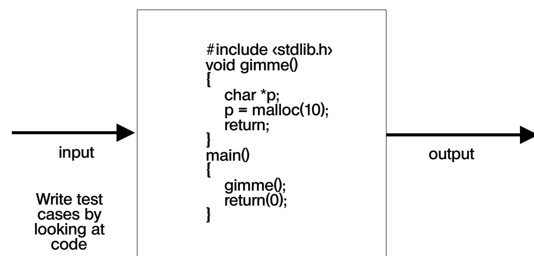

Figure 5. White box testing.

You can use these two processes as the basis for a third process: regression testing. If you save your black-box and white-box test cases, you can re-run them to perform unit-level regression testing and monitor your existing code's integrity as you modify it. The concept of performing regression testing at this level is novel. When you perform unit-level regression testing, you can immediately determine if your modifications introduced problems. This way, you can fix problems immediately after you introduce them, and you do not have to wade through layers of code in order to fix each error that you introduced.

The problem with unit testing is that it is difficult, tedious, and time-consuming when performed without automatic unit testing technologies. A brief look at what is involved in unit testing reveals why it is difficult-- if not impossible-- to integrate it into today's development cycles if you do not automate it.

The first step in performing unit testing on embedded software is constructing an environment that allows you to execute and test the function on the host system. This requires two main actions:

- Designing scaffolding that will run the function.
- Designing stubs that return values for any external resources that are referenced by the function under test, but that are not available or accessible.

The next step is designing and building appropriate test cases. In order to thoroughly test the function's construction and functionality, you should design two types of test cases: black-box and white-box.

Black-box test cases should be based on the specification document. Specifically, at least one test case should be created for each entry in the specification document; preferably, these test cases should test the various boundary conditions for each entry. You should not just verify that a simple input produces the expected outcome; rather, you should consider what range of input/outcome relationships you need to verify in order to prove that the specified functionality is implemented correctly, then write test cases that will fully verify that functionality. You may want to test for omissions as well as faults of commission.

White-box test cases should uncover defects by fully exercising the function with a wide variety of inputs. These test cases should try to do two things:

- Aim for 100% coverage of the function: As I explained earlier, this degree of coverage is possible at the unit level because it is so much easier to design inputs that reach all parts of the function when you test a function apart from an application. 100% coverage may not be possible in all situations, but it is the goal that you should strive for.
- Make the function crash.

However, it is incredibly difficult to create such test cases on your own, without a technology to create them for you. To create effective white-box test cases, you must examine the function's internal structure, then write test cases that will cover the function as fully as possible and uncover inputs that will cause the function to crash. Achieving the scope of coverage required for effective white-box testing mandates that a significant number of paths are executed. For example, in a typical 10,000 line program, there are approximately 100 million possible paths; manually generating input that would exercise all of those paths is infeasible.

After these test cases are created, you should execute the entire test suite and analyze the results to determine where errors, crashes, and weaknesses occur. You should have a way to run all of these test cases and easily determine which test cases had problems. You should also gauge coverage to determine how thoroughly the function was tested and to determine what additional test cases are necessary.

Whenever a function is modified, you should perform regression testing to ensure that no new errors were introduced and/or that previous errors were corrected. Integrating unit-level regression testing into your development infrastructure will keep many errors at bay; because errors will be detected immediately after they

Copyright 2001 by Dedicated Systems Magazine - 2001 Q1 (http://www.dedicated-systems.com)

are introduced, they will not be allowed to enter the application and spawn more errors.

There are two ways you can perform regression testing. The first way is to have a developer or tester examine every test case and determine which test cases are affected by the modified code. This approach uses human effort and time to save the computer work. A more efficient approach is to have a computer automatically run all test cases every time the code is modified. When you take this approach, you can optimize your time because you do not need to examine the entire test suite to determine which test cases need to be run and which do not.

If you can automate the unit testing process, it will not only improve quality, but also save you significantly more time and resources than it consumes. If you are developing in C, you can use an automatic unit testing technology to automate your unit testing. The more processes the technology can automate, the more helpful it will be.

Some things to look for when selecting a unit testing technology:

- Does it work with your code and/or compiler?
- Does it automatically create test harnesses?
- Does it automatically create test cases?
- Does it provide an easy way to enter user-defined test cases and stubs?
- Does it automate regression testing?
- Does it include or integrate with an automatic runtime error-detection technology?

UNINSTRUSIVE DEBUGGING TECHNIQUES

Because real-time operating systems perform certain tasks within predetermined time constraints, timing is a crucial variable that developers must compensate for when they install the software. They typically encounter lots of interrupts as they run, and it is crucial for the application to behave properly when the interrupts occur. Things get even more complicated when there are multiple interrupts, or when multi-threaded applications have different threads interacting with each other. These applications essentially have multi-path executions, which means that different pieces of code look like they execute at the same time, even if there is only one central processing unit (CPU). It is interesting to note that if these applications were running on multiple CPUs, the different threads would actually be executing on different CPUs.

If the errors that occur in real-time applications are directly related to the interactions between the interrupts and the program itself, the errors will be very much time-sensitive. In this case, it becomes crucial to record the order in which the errors occur because you must understand the cause and effect of each error. Herein lies the main problem with debugging real-time systems: there are quite a few hard-to-detect bugs that are only visible under certain timing conditions.

The problem with these timing-related errors is that they are not easily repeatable. It is difficult to recreate the precise timing conditions that caused the program to malfunction. The debugging mechanism that helps debug these applications has to be as unintrusive as possible. Any intrusiveness will alter the timing of the program and will cause the error not to occur. In practically any other type of development, causing errors not to occur is a good thing. In this case, it foils our efforts to debug the operating system.

Anyone who has studied physics can think back to their knowledge of the Heisenberg effect to help them understand the unique problems of debugging a real-time operating system. Werner Heisenberg, a German physicist, wrote the Uncertainty Principle, which states that it is impossible to simultaneously determine the location and velocity of a moving particle. Heisenberg believed that by measuring where a particle is, you alter its position, and when you alter its position, you will no longer be able to get the measurement you wanted. The act of measurement disturbs the effect you are supposed to be measuring. The Uncertainty Principle is an important tenet of Quantum Mechanics.

To put this dilemma simply, debugging requires you to gather information about the state of the system. But gathering information about the state of the system disturbs the timing of the system enough to make defects hard to reproduce reliably.

So essentially, the goal here is to come up with a method that allows you to detect real-time errors and see what is happening in the program without disturbing the timing. Perhaps your first instinct would be to use a debugger, but typical debuggers interrupt the execution of the real-time application and alter the timing. Simulators are also an incomplete solution because they cannot simulate the timing of the hardware. Nobody has built a simulator that can simulate real time; you can only measure timing when you put the software on the hardware itself.

Doing this requires a mechanism to record the state in a lightweight fashion. Writing to the memory is one possible mechanism; it is useful because writing to the memory is a very fast operation. One of the ways to use this mechanism is to establish a buffer somewhere in the memory and have a pointer in your code to that buffer. This pointer should write to the beginning of the buffer. In the code, add assignments with different values to write to that pointer. Every time you want to write to the memory, you write to the pointer, then you increment the pointer. Sometimes, it is good to use a circular buffer (where once the pointer writes to the end, you start again from the beginning). This lets you track the sequence that leads to a problem. You also need to figure out a way to preserve the buffer once the program crashes or exits; you need to be able to access it and do what I call "post-mortem debugging." How you achieve this depends on your hardware; often, you can do this by not resetting the hardware.

At this point, you need a mechanism to read the memory. You can use a debugger, or you can use other techniques to retrieve the information from the memory. For instance, you can write a simple program that sends this data to a printer or file. Whichever technique you use, you will most likely end up analyzing the

buffer manually. If you used a circular buffer, be sure you know exactly where the buffer started; the steps that began the sequence will be immediately below the pointer; the steps that occurred immediately before the crash will be immediately above the pointer.

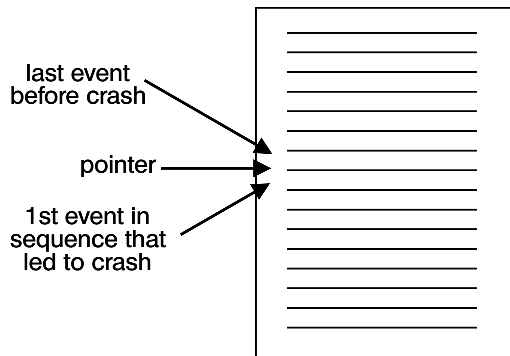

Figure 6. Sequence of events with a circular buffer.

Your goal now is to make sense out of the sequence of writes that occur in the buffer. This process is similar to trying to understand an airplane crash by looking at the sequence of instrument readings preserved in the airplane's black box. You will be observing the program's performance after the fact, which is obviously less intrusive than trying to watch during the execution itself.

Sometimes it is very difficult to reconstruct these events and you have only a vague idea of when the error happened. In fact, sometimes it can take months to understand the cause of the crash. In this case, you could use a logarithmic approach to debugging to determine what statement is responsible for the error. You can place markers (such as exit statements) around the code, write to the memory before the markers are reached, and see if the code actually crashes before the marker. If so, you know that the error occurs between the markers. This technique exposes time-related problems because you can use it to determine what segment of the code the race-conditions occurred in.

Another solution is to use firewalls as a debugging technology. A firewall is a point in the logical flow of a program where the validity of the assumptions made by the code that follows is asserted. Verifying that these assumptions are indeed correct is distinct from normal error checking. The firing of a firewall is an alarm for the developer and indicates that the internal state of the program is inconsistent. It could happen, for example, if a function that expects its integer argument to be strictly positive receives either zero or a negative number. At first, most firewalls appear trivial and superfluous. However, experience with large projects shows that as software systems evolve and age, the implicit assumptions they make on their environment become more and more likely to be violated. In many cases, even the original designer of a piece of code will have a hard time reconstructing what constitutes proper use of a piece of code.

Firewalls implemented inside embedded systems require a special connection to communicate messages outside of the hardware; how to establish such connections is beyond the scope of this paper.

CONCLUSION

The above error prevention, error detection, and debugging techniques can dramatically improve embedded software quality and reduce the time and money spent on debugging. The above practices can easily be applied to many different types of embedded software projects. This means that you can leverage your investment and expertise in these techniques as you move to new projects and different target technologies. These techniques will also help you ensure that your code can easily be maintained, modified, and ported to new types of devices. In short, these techniques will not only help you improve your current embedded applications and development process, but also help you ensure that as new embedded devices become available, you have the expertise needed to develop high quality applications for these technologies-- on time and on budget ■

Dr. Adam Kolawa is the CEO of ParaSoft Corporation, a leading provider of software productivity solutions. Dr. Kolawa holds a Ph.D. in theoretical physics from the California Institute of Technology. Dr. Kolawa has extensive experience in both programming and managing programmers. He has written several successful commercial software products, and has headed the development of a large number and variety of software projects, including software development tools, retail solutions, web development and management tools, data mining tools and more. Heading a company that sells products to programmers and managers has given Dr. Kolawa the unique opportunity to interact with software developers and managers at multiple levels. Dr. Kolawa established his ability to clearly communicate the expertise he has gained from his education and experience in the many papers on physics and computer science that appeared in such publications as the Caltech Concurrent Computing Project Memo and the Proceedings of the Hypercube Conference. Two of Dr. Kolawa's papers are included in Parallel Computing Works!. Ed. Geoffrey Fox. Dr. Kolawa has spoken at many conferences including UniForum, Linux Expo, STAR East '99, SuperComputing, SGI Expo, Xhibition, Quality Week, IEEE conferences and has also conducted national seminars on Strategies for Effective Software Development.

REFERENCES

- M. Aivazis and W. Hicken. "C++ Defensive Programming: Firewalls and Debugging Information." C++ Report (July-August 1999): 34-40.

Copyright 2001 by Dedicated Systems Magazine - 2001 Q1 (http://www.dedicated-systems.com)