Sensible software testing

By Sean Beatty
Engineering Manager
Software Engineering Professionals

Many demands are placed on a software engineer's time. High quality, robust feature count, and low cost are often considered critical goals even though they generally compete against each other. In many development environments, time to market is critical, and the specification is in flux throughout the development process. Too often, testing gets whatever time is left between the point at which the code is finished and the date it must be shipped.

While it may be impossible to fix all the problems with the environment in which software engineers must work, it is helpful to quantify the challenges they face. This article aims to help the software engineer develop a practical approach to testing firmware. By understanding the goals of a given testing strategy and the associated costs, better decisions can be made as to what to test, and when.

Program

To meet the goal of identifying a practical software testing strategy for a given project, I propose analysing the problem along three lines:

- Identify the types of bugs common in embedded systems software
- Discuss the various methods used to find bugs
- Apply the "best" methods as part of a sound software development process

This approach sounds simple, but like many plans, the devil is in the details. The engineer needs to know the frequency of a certain type of bug's occurrence and the relative effect it has on the function of the system. Most of this article is devoted to elaborating on the various types of bugs so these issues can be better understood. It's only after developing a good scope of the problem that an engineer can develop an appropriate solution.

Once the problem is understood, various ways of uncovering software bugs are considered. This includes discussing the effectiveness of given methods in finding the different types of bugs. When all these topics are properly understood, the engineer is well on the way to implementing a sensible embedded systems software test plan.

Before proceeding further, I must clarify some of the terms used frequently in the following discussion:

- The words bug, failure, and error are used interchangeably. They all indicate some problem with the software
- The terms subroutine, function, and method are used synonymously to indicate some code that can be called
- Bug frequency is categorised in four groups: rare, less common, more common, and common
- Bug severity is indicated by one of four labels: non-functional, low, high, and critical

Program

When devising a plan to remove bugs from software, it helps to know what you're trying to find. Software can fail in many ways, and mistakes are introduced into the code from many different sources. Some bugs have greater repercussions than others, and almost all of them have consequences determined by the type of application and the domain in which it operates.

What follows is a catalogue of errors found in embedded systems software. The list is long, yet important. Without understanding how software can err, it's difficult to find the potential errors. The relative frequency and severity is listed for each type of bug. Definitions for some of the terms used in the discussion appear in the sidebar.

Non-implementation error sources

Errors can be introduced into the code from an erroneous (or ambiguous) specification or an inadequate design. They can also result from hardware that doesn't operate correctly, or operates differently than specified or otherwise understood. *Frequency:* All too common. *Severity:* Ranges from non-functional to critical.

Implementation error sources

Bugs introduced into the software during the "coding" or implementation phase are quite common. These types of errors will receive emphasis in this article. There are many types of implementation bugs of varying severity. It helps to group them into general classifications based on some common elements. Arguably, some bugs could appear in more than one classification.

Algorithm/logic/processing bugs

Off by "1"

It's common to be "off by one" in a calculation. For example, a loop needs to execute 10 times, and a construction such as for (x = 0; x < = 10; x++) is used. This will execute 11 times, not 10 times. Another example: for (x = array_min; x < array_max; x++). If the intention is to set x to array_max on the last pass through the loop, the software is in error. *Frequency:* Common.
Severity: Varies, but is typically high, since the program doesn't operate as intended. However, in other instances this type of error may never be detected. For example: filtering a variable. If an average of the last 10 samples is intended, and instead nine samples or 11 samples are averaged, it's possible that a difference in the program's function will not be detected.

Parameter passing

Incorrect arguments or parameters may be passed to a subroutine. Examples include passing a financial number in dollars when a yen amount was expected or returning a temperature in degrees Celsius when the calling program is assuming Fahrenheit. Frequency: Common only when many complicated function invocations are used. Severity: Varies.

Return codes

Improper handling of return codes is another potential error source. Assuming the called function executes correctly and not checking for unexpected return codes can cause problems. Avoid using the same return code for different conditions. A programmer could misinterpret a return code, especially when using a routine developed by someone else. *Frequency:* Common when using unfamiliar libraries or complicated functions with many return codes. *Severity:* Varies.

Math overflow/underflow

Unless you're using a floating-point library, arithmetic overflow or underflow must almost always be checked. Fixed point and integer types can only hold numbers of a certain range. The result of an operation should be checked to ensure an overflow/underflow did not occur, before using the result in any meaningful way. Failure to check for overflow/underflow can result in data-sensitive problems that can be difficult to track down. If an overflow condition is detected, it must be handled in some appropriate way (often by limiting the data to the largest number that can be represented in the data type). These checks are unnecessary only when the input data is well known and it's impossible for the operation to ever overflow or underflow. *Frequency:* Common where arithmetic operations are performed using integer or fixed-point math. *Severity:* Generally high.

Logic/math/processing error

Of course, it's easy to make mistakes when implementing the logic of a program. Incorrect decision logic (IF, THEN, ELSE, SWITCH, WHILE, GOTO , and so on) grows common in complicated functions and deeply nested decisions. For example: IF ((this AND that) OR (that AND other) AND NOT (this AND other) AND NOT (other OR NOT another)). Boolean operations and mathematical calculations can also be easily misunderstood in complicated algorithms.

Frequency: Common.
Severity: Generally high.

Re-entrance problem

If a section of code can be interrupted before it completes its execution, and can be called again before the first execution has completed, the code must be designed to be re-entrant. This typically requires that all variables referenced by the re-entrant routine exist on a stack, not in static memory. In addition, any hardware resources used must be manipulated carefully. If not, data corruption or unexpected hardware operation can result when the interrupted (first) execution of the routine finally completes.

Frequency: Rare in embedded systems code, since most code is not re-entrant.
Severity: Generally critical.

Incorrect control flow

The intended sequence of operations can be corrupted by incorrectly designed for and while loops, if then else structures, switch case statements, goto jumps, and so on. This causes problems such as missing execution paths, unreachable code, incorrect control logic, erroneous terminal conditions, unintended default conditions, and so on.

Frequency: Common.
Severity: Varies from non-functional to critical.

Data bugs

Pointer error

Pointer errors border on the famous (infamous?). Every programmer who has used pointers with any frequency is familiar with the mysterious symptoms with which a bad pointer manifests itself. Pointer problems can be notoriously difficult to track down. Some coding guidelines I've seen even recommend avoiding all pointer usage, if possible, to avoid these nasty bugs. Pointer errors are often more common when certain types of structures are used in the code. Doubly linked lists make heavy use of pointers, so it's easy to point to the wrong node or link to a NULL pointer. When using look-up tables or lists, take care to properly increment any pointer used to step through the table or list. General pointer problems of de-referencing a NULL pointer or pointing to the wrong thing grow more common as the number of nested references increase, for example **array_of _ptrs_to_ptrs[*index_ptr] . A bad function pointer can cause the wrong subroutine to be called. Frequency: Common in languages that support pointers, such as C. Severity: Almost always high or critical.

Indexing problem

Where "C" programmers use pointers, assembly language programmers use index registers. Index registers (or similar types of registers in other architectures) provide the same type of indirection useful for table look-up, walking through lists, trees, and other data structures, and calling a routine determined at run-time. They also have the same potentials for error.

High-level language programs often make heavy use of arrays. Many times, strings are stored as arrays of characters. Individual elements within an array are identified with an array index. Accessing the wrong element within an array is another example of an indexing problem.

Frequency: Common.
Severity: Almost always high or critical.

Improper variable initialisation

Sometimes improper initialisation can be obvious, as when reading a variable that has never been written. Other times it's more obscure, such as reading a filtered value before the proper number of samples have been processed.

Frequency: Less common.
Severity: Often low, but it varies.

Variable scope error

To get the expected results, the correct data must be processed. The same name can be applied to different data items that exist at different scopes. For example, an automatic variable can coexist with a static variable of the same name in that file. Different objects instantiated from the same class refer to their members with the same name. When pointers are used to reference these objects, it becomes even easier to make a mistake.

Frequency: Less common.
Severity: Generally low to high.

Improper data usage

Initializing a variable properly is only the first step in using it correctly. It's generally a bad idea to use a variable for more than one purpose. It's too easy to modify it in one place for one reason and then alter it again in another place for a different reason-undoing the first change. This is generally only a problem in smaller systems that make heavy use of global data and are short on RAM. Another improper data usage involves modifying data but never storing it or testing it. This is unlikely to perform as intended. Storing a data value in the wrong units is also a serious problem, for example calculating a result in degrees Fahrenheit and storing it in a temperature variable that expects degrees Celsius.

Frequency: Common.
Severity: Varies.

Incorrect flag usage

This is a specific type of incorrect data usage, but it occurs commonly enough to merit its own category. Flags are typically used to communicate between various parts of a program, are generally global in scope, and are almost always static. When an RTOS is used, this communication function may be handled with a semaphore. Every flag should be set, cleared, and tested at some point in the program. Missing one of these three generally indicates an error. A flag may inadvertently be used for more than one purpose, or used to indicate more than one condition. This is also typically an error.

Frequency: Common where hard-coded constants are used to represent the bit-position of a flag within a flag-word, instead of using symbolic constants. Less common when using bit-fields as part of a structure.
Severity: Varies.

Incorrect address

Most of the time, a bad address is the result of an incorrect pointer. Nevertheless, it is possible to hard-code a bad address into the code. This generally happens only when the memory sub-system or some peripheral changes.

Frequency: Rare.
Severity: Generally high to critical.

Data/range overflow/under-flow

These problems can be hard to detect. Inadvertent overflow of a data type can produce some very strange symptoms. Most of the time, the program executes as expected. Then occasionally it goes haywire, seemingly unexplainably. Errors of this type include passing a parameter that is out of bounds, and storing the result of a calculation in a data type not large enough to hold the data. Of course, the more strongly typed the language, the less of a problem this becomes.

Frequency: Comm

on in assembly language programs, and high-level language programs that target small (8-bit) processors. In the latter, the types may be smaller than expected. For example, is it safe to assume an integer is 16-bits wide? Severity: Low to critical. Sometimes the effects can go unnoticed.

Signed/unsigned data error

A mix of signed and unsigned data types can easily lead to

calculations that produce wrong results. Assembly languages have different branch instructions used after comparing signed and unsigned data. Using the wrong branch instruction is often a critical error. When mixing signed and unsigned types, care must be taken to understand the sign of the result and store it in the proper data type. Mixed sign arithmetic can easily overflow the data types used in the calculation if not handled properly.
Frequency: Common in assembly language programs, or when using fixed-point (integer) math. Not a problem where floating-point math is used exclusively.
Severity: Varies, generally high to critical.

Incorrect conversion/ type-casting/scaling
Converting a data value from one representation to another is a common operation, and often a source of bugs. Data sometimes needs to be converted from a high-resolution type used in calculations to a low-resolution type used in display and/or storage. Conversion between unsigned and signed types, and string and numeric types is common. When using fixed-point math, conversion between data types of different scales is frequent. Typecasts are useful to get data into whatever representation is needed, but they also circumvent compiler type-checking, increasing the risk of making a mistake.
Frequency: Common in programs that are more complicated.
Severity: Varies, low to critical.

Data synchronisation error
Many real-time embedded systems need to share data among separate threads of execution. For example, suppose an operation that uses a number of different data inputs is performed. This operation assumes these data are synchronous in order to perform its processing. If the data values are updated asyn-chronously, the processing may be using some "new" data items with some "old" data items, and compute a wrong result. This is especially true if a control flag is used to interpret the data in some way. Some embedded systems use a serial port to send a "system snap-shot" of the critical data items in response to an asynchronous request. If the data items in the snapshot are not updated synchronously, the snapshot may contain a mix of some current information and some old information.
Frequency: Less common.
Severity: Low to high.

Real-time bugs
Interrupt handling
It's critical to be able to handle all interrupts that the system will ever receive. Receiving an unexpected interrupt without being able to handle it is usually disastrous. For this reason, even interrupts that are not expected to occur should still be handled with an "unexpected interrupt" handler, just in case. Vectors for every interrupt that your processor could receive, either intentionally or inadvertently, must be present and must point to the correct handler.

An equally disastrous mistake is an incorrect return from an interrupt handler. Most processors have separate instructions to "return from subroutine" and "return from interrupt." If you use the wrong instruction to return from the interrupt, you will corrupt the stack (by not unstacking registers that were pushed onto the stack automatically when the interrupt was acknowledged). High-level languages often use a special keyword to indicate to the compiler that the "return from interrupt" instruction should be used with a particular function.
Frequency: Rare.
Severity: Critical.

Interrupt suppression
Most programs that use interrupts also suppress them around critical

sections of the code. Receiving an interrupt during a critical section may cause the program to miss some time-related specification, corrupt data, mishandle external hardware, and so on. Therefore, it's critical to ensure that all sections of code that need interrupt suppression have it. Often, this is system-specific knowledge that should be well documented.

One situation that doesn't require detailed system knowledge involves data corruption. Whenever data is written by both an interrupt service routine (ISR) and another place in the program, special care must be taken. Any read-modify-write on the data must be atomic, or interrupts must be suppressed around that section. When using a high-level language, be aware that writes to a multi-byte type may not be atomic. This may not be obvious by looking only at the source code. If an interrupt occurs between the read and modify cycles, the modification may be made inappropriately, since the data may have changed. If an interrupt occurs between the modify and write cycles, the new data updated by the ISR will be overwritten. Even if the data is not modified in this latter situation, the ISR-updated data could be overwritten. This type of read-don't-modify-write is sometimes done to refresh or test memory, or in certain peripheral interfaces. Too much of a good thing can be bad. If interrupts are suppressed too long, timing deadlines may not be met. Or a system clock may not keep time "correctly." Processors typically inhibit all interrupts of priority equal to or lower than the current interrupt priority. Therefore, when calculating the maximum interrupt suppression that your system could ever encounter, it's not enough to simply look for "interrupt disable" and "interrupt enable" instructions. You must also account for the time spent within any ISR. *Frequency:* Less common. *Severity:* Critical.

Task synchronisation
Tasks must be synchronised correctly. Some operations must wait for others to occur first or other tasks to complete. One task may acquire raw data; another may process this data as a set; still another may make control decisions on the processed data values. Proper synchronisation is sometimes implemented by relying on flags or semaphores to control task execution. Other tasks are synchronised by scheduling them to execute at regular intervals. If one task doesn't finish in time, a second task that depends on its completion may fail. Other task-related problems include race conditions and priority inversion problems. *Frequency:* Less common. *Severity:* Varies.

System bugs
Stack overflow/underflow
"Don't blow your stack!" Although this expression usually refers to something quite unrelated to embedded systems, it's quite applicable here. Pushing more data onto the stack than it is capable of holding is called overflow; pulling more data from the stack than was put on it is called underflow. Both result in using bad data, and can cause an unintended jump to an arbitrary address-very bad. The stack pointer can also be directly manipulated on many processors, and is sometimes so used to quickly generate temporary variable space on the stack.

Many high-level languages offer no direct way to manipulate the stack. However, deeply nested subroutines with many parameters can still cause an overflow. Therefore, it's important to ensure that the worst case stack depth generated by a program can never exceed the stack allocated. Multi-tasking systems complicate this analysis, since each task needs its own program stack. In addition, interrupts require stack space in order to save the value of the processor's registers. Moreover, the deepest stack needed by any interrupt must be added to each task's stack space (assuming any task can be preempted by any interrupt). It's easy to see how quickly the program's stack space can be con- sumed in these types of designs. Frequency: More common in assembly language programs and complicated designs. Severity: Critical.

Other stack errors
Other stack errors can corrupt data. For example: pushing the X-register then Y-register onto the stack to save their values, but pulling them off the stack in the wrong order. A stack imbalance occurs when not all the registers pushed onto the stack at the beginning of a routine are pulled off the stack before the routine returns (or vice-versa). This causes execution to jump to an arbitrary address. *Frequency:* Less common. Generally only occurs in assembly language routines. *Severity:* Stack imbalances are always critical, and generally produce immediate and dramatic failures. Data problems vary in severity.

Version control error
It doesn't matter how good your last bit of code was if it didn't get included in the build. Including the version of the file that still has the bug produces another bug report. Including a version that is now incompatible with the latest hardware may produce many bug reports! Version control grows in importance as the complexity of the software project (read: the number of people involved in the software) grows. *Frequency:* Common only in complicated systems with many files and many developers. This problem can become more difficult in distributed development environments. *Severity:* High to critical.

Resource sharing problem
Resource sharing is common in most embedded systems at some level. Wherever sharing occurs, strict rules for using the resource cooperatively must be defined and followed to avoid conflicts. Ignoring a mutual exclusion semaphore can corrupt data. Two different tasks that both use the same peripheral must cooperate. For example, an analogue multi- plexer may be used to direct one of a number of different inputs to a single A/D converter. If one task alters the mux setting to measure a given signal and another pre- empts it and sets the mux to pass a different signal, when control returns to the first task it will be measuring the wrong signal. *Frequency:* Less common. *Severity:* High to critical.

Resource mapping
Some microcontrollers allow the peripheral registers and memory to be mapped to many different locations. Some applications use different mappings for various purposes. Get this wrong, and it's likely the code won't even run. Less obvious is mapping the code or initialised data to a RAM area during development, where it's easy to modify. This is common when downloading the code into instrumentation of some sort. If the code isn't re- mapped before burning it into EPROM (or worse yet, releasing the ROM mask), your data or code becomes whatever happens to be in the RAM after power-up! *Frequency:* Rare. *Severity:* Critical.

Instrumentation problem
Sometimes a software bug is not actually a problem with the soft- ware at all. Instrumentation gen- erally alters the behaviour of the system, albeit in very small, subtle ways. Sometimes problems dis- appear when the emulator is connected, and other times they only appear when the emulator is used. Reported bugs could also be a result of improper use of the instrumentation. *Frequency:* Less common. *Severity:* Low.

Other bugs
Syntax/typing
Compilers can be a considerable help in checking the accuracy of our typing. For example, some will issue a warning when assign- ments are made within a condi- tional expression: writing if (a=1) when if (a==1) was intended. But other errors defy detection.

No compiler will warn you about misspelling a variable name when the misspelling is also a valid symbol (for example, hiRes_speed , instead of hiRev_speed).
Frequency: Less common.
Severity: Varies.

Interface
Complex interfaces are a common source of errors. Interfaces can be external to the processor or internal. The modules interfaced to could be hardware or software. Documentation that is missing, incomplete, ambiguous, or incorrect is often to blame. Hardware or software changes that aren't properly communicated to all the appropriate people also produce interface problems. These types of bugs include protocol errors and timing or sequence problems. Examples: incorrect EEPROM erase/write sequence, improper use of LCD controller chip commands, wrong sequence in reading/writing serial communication interface registers.
Frequency: Common.
Severity: High to critical.

Memory allocation/ deallocation
Using memory management routines can greatly simplify the efficient use of available memory. It can also be an added source of errors. Examples: not checking for successful allocation before using the memory, not freeing memory when it's no longer needed (memory leak).
Frequency: Common only with high level languages, and only when using routines such as malloc() and free(). Less common with languages that do more memory management automatically (use constructors, destructors, and references). Severity: Varies. Sometimes a small memory leak may go unnoticed. Not checking an allocation before using the memory can crash the system.

Peripheral register initialisation
Most embedded systems have peripheral hardware devices

that they use to perform some necessary work. These peripherals often have many different modes of operation, increasing the number of applications for which they're useful. This can complicate the initialisation and use of these devices, producing another source of errors.
Frequency: Less common.
Severity: Critical.

Watchdog servicing
Watchdog timers help ensure that if something in the system goes exceptionally wrong, it will fail in a safe, or at least predict-

able, manner. Most software FMEAs make use of watchdog timers to mitigate risks. However, with every added complexity comes yet another potential source of problems. Servicing the watchdog timer must be done properly and at the right time. The watchdog must be enabled, and set to timeout at the correct interval. The watchdog servicing must be guaranteed to occur frequently enough, under every correctly operating scenario, to prevent a timeout. Otherwise, the device intended to mitigate serious problems becomes a

source of them. This implies that a thorough understanding of the timing characteristics of the system is necessary to ensure proper use of the watchdog timer. One last note: some programmers have used a periodic interrupt to ensure that the watchdog servicing is done on time. As long as the periodic interrupt is at a higher priority than whatever is going wrong, this effectively prevents the watchdog from "watching" the code, rendering its service useless.
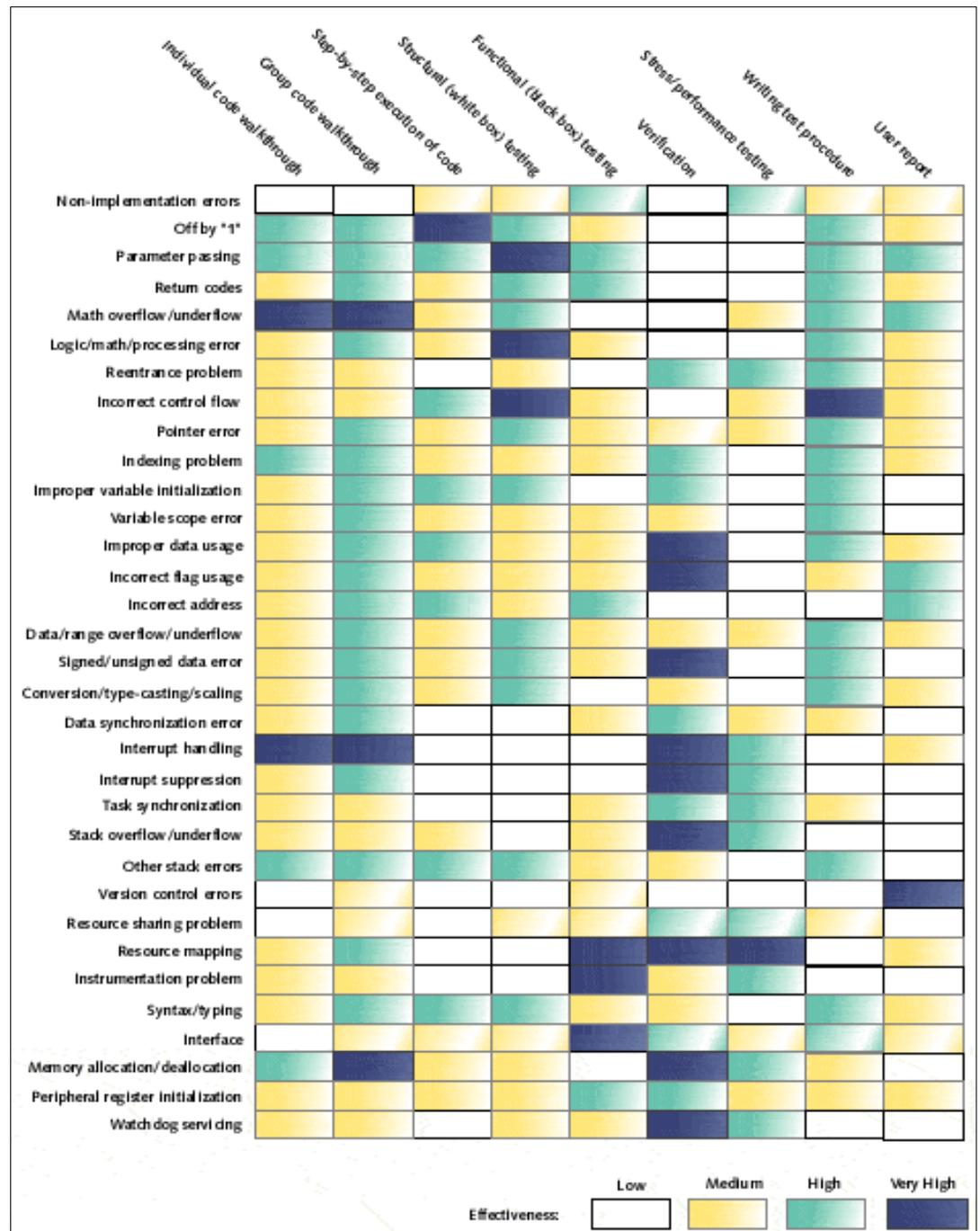*Frequency:* Less common.
Severity: Critical.

Figure 1: Effectiveness of various bug-finding techniques

Finding hidden problems

This long list of errors begs the question "How do I find these potential bugs?" (Perhaps a better question is "How do I prevent them?" That's a topic for another article.) Many techniques and tools can be used to find bugs. Some of these are more expensive than others are, both in time and material cost.

Costs can only be described relative to each other. An emulator typically costs more than a simulator. However, there's a big difference in cost between an emulator for an 8-bit microcontroller and an emulator for a 32-bit DSP. For the purposes of this article, costs will be grouped in four categories: none, low, moderate, and high. General effectiveness of a testing technique will be categorised as either low, medium, high, or very high. The effectiveness of a particular technique for a specific software error type is given in Figure 1.

Individual code walkthrough

I realise it's probably not accurate to call this a test, but it is so effective I would be remiss not to mention it here. No other technique is as effective at identifying bugs as a good walkthrough. This starts with the individual programmer carefully examining his code for potential mistakes, omissions, misunderstandings, and adherence to the project coding standards. This is best done many hours (if not a day or more) after the code is originally written. That provides the opportunity for a fresh perspective. This walkthrough is best done before the engineer tests his code on the target.
Cost: Time-very low, money-none.
Effectiveness: Very high

Group code walkthrough

A good walkthrough can find more bugs than any other single activity. Conducting group walkthroughs is as much art as it is science. Group and team social skills (or lack thereof) become apparent. The goals of a group code walkthrough include finding potential problems, ensuring adherence to the software design, looking for subtle effects on the rest of the system, and identifying improvements. Ego has no place in this activity. It must be okay to have others identify your mistakes (and vice versa). The best walkthroughs are short, and done frequently. *Cost:* Time-low, money-none. *Effectiveness:* Very high. Affected by the effort, experience, and attitudes of the review team.

Step-by-step execution of code

This type of testing attempts to find problems not just by walking through the code, but by executing every line of the code. This code execution can be done on a simulator or on the target. Code execution can be controlled with an in-circuit emulator or observed with a logic analyser. Sometimes every branch and condition is executed, not just every line of code.

The purpose of this type of testing is to observe the correct operation of the code. This implies that the correct operation is well understood, if not documented. The goal is not to find particular types of errors, or to observe particular behaviours. This type of testing is sometimes done on a single module or file of source code. Other times it's performed on the entire software application. Cost: Time-can be high, especially when the entire system is being examined. Always tedious, since most of the code operates perfectly. Money-low to moderate. Simulators can be inexpensive, but ICEs can be quite expensive. Effectiveness: Moderate, depending on the effort of those performing the test, and how well the intended behaviour is understood by them.

Structural (white box) testing

Sometimes called "glass-box testing," this activity uses an unobstructed view of how the code does its work. This type of test is usually performed on a single unit of the software at a time. Test procedures are written to exercise all the important elements of the code under test. This may include exercising all the paths in the module. It often involves many executions of the same code, with different values of data. Boundary conditions are typically exercised. This test is also used to determine the consistency of a component's implementation with its design. *Cost:* Time-high. It takes time to write the procedures, and examine what is critical to test. Of course, once the test procedures are written, they can be reused to retest the same module later when minor modifications are performed. Money-depends on how the testing is done. Simulators are less expensive than ICEs. *Effectiveness:* high. If a module passes a thorough white box test, the level of confidence is high that it won't cause problems later.

Functional (black box) testing

In functional test, the program or system is treated as a black box. This implies the tester has no knowledge of what's in the box, only its inputs and outputs. The system is exercised by varying the inputs and observing the outputs. This type of test is usually performed on the entire software system. In complex applications, the software is broken down into components or sub-systems, which are tested individually. All the individual parts are then brought together (usually one at a time) and the integrated system is tested.

Test procedures are usually written to describe the expected behaviour in response to a given environment and input stimulation. In the absence of formal test procedures, some engineers simply go through the system or software's actual behaviour to its specified behaviour without regard to the software implementation. Cost: Time-medium. This type of testing is almost always done to one degree or another. Ultimately, the customer will exercise all the features of the program; it's better to find any obvious bugs before he does. Money-varies depending on how easy it is to manipulate the inputs and observe the correct outputs. Sometimes the entire test can be done standalone; other times, custom equipment must be constructed in order to verify correct behaviour. Effectiveness: Medium. Many parts of the system are difficult to exercise with a black box test. Error conditions (especially due to hardware failure) can be difficult, if not impossible, to generate. Some combinations of inputs are difficult to produce, especially those with unique timing characteristics.

Verification

Different organisations mean different things when they use the term verification. Some use "verification" and "validation" interchangeably; others make a clear distinction between the terms. The IEEE defines validation as "the process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements." This type of activity is described in the previous section. Verification, on the other hand, is defined as "(1) The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase. (2) Formal proof of program correctness."

In this article, I use the term verification to refer to the detailed analysis of software, independent of its function, to determine if common errors are present. It is generally performed on the code as a whole, not on individual units. It's very white box-like, in the sense that verification examines the structural integrity of the code, and not functionality. Examples of typical verification checks include: stack depth analysis, proper watchdog timer usage, power-up/power-down behaviour (in the sense of proper initialisation and clean up), singular use of each variable, and proper interrupt suppression. Cost: Time-medium. Some checks

can be automated, or partially automated. Others are tedious. Money-low to medium, depending on what is checked. Most checks can be done by hand. Others, such as verifying interrupt timing, require test equipment. Effectiveness: High. This is the only way to develop confidence in some areas of the code. It's difficult to generate tests that will produce certain conditions (for example, worst-case stack depth, maximum interrupt latency). However, by analysing the code, the engineer can determine what the worst case condition is and show that the system has been designed to handle it.

Stress/performance testing

Can the system handle its intended load? How close to the edge of acceptable performance does the system come? Stress tests are designed to load the system to the maximum specified load and then beyond, to see where it breaks. "Load" could be number of users, number of messages per unit time, frequency of a periodic interrupt, number of dynamically allocated tasks, and so on. Knowing at what point the system fails tells us how much overhead (factor of safety) we have. Performance testing is conducted to measure the system's performance. This may be important to verify that a requirement is met, to ensure the design is adequate, or to determine resources available to add more features. (Examples: processor throughput, interrupt latency, worst case time to complete a given task, and so on.)
Cost: Time-low. Money-low to high, depending on what's measured.
Effectiveness: High. There's nothing like knowing how close to the edge you're really running.

Writing test procedure

Many times the greatest benefit in developing formal test procedures is not executing the tests, but simply developing good test procedures. The code and/or the requirements must be examined carefully to determine what to test and how to test it. In the light of this increased scrutiny, many problems in the code are uncovered.
Cost: Time-high. It does take time to develop good test procedures. Money-none.
Effectiveness: High.

User report

From one perspective, your product is continually being tested by those who use it. Occasionally, a user may find a bug that evaded all previous test efforts. Most customers do not enjoy being used as unpaid test engineers, and get a little testy when they find bugs. Others volunteer to test your code (beta-testers) in order to get the benefits of your product sooner. Regardless, user reports always have to be verified: sometimes alleged errors are due to improper user understanding, hardware conflicts, or other problems unrelated to the software's operation.
Cost: Time-none (of yours). Money-none. Of course, the intangible costs to your business could be much larger.
Effectiveness: Medium.

Program

Various tools are helpful to test code and find bugs. Software debuggers allow the programmer to start and stop execution, modify the data in memory and registers, and sometimes trace execution and coverage. This can be useful in stepping through the code, and forcing execution down particular paths. These techniques are often used in unit testing. Similarly, a logic analyser can be used to trace execution. It can show what happened in time, but doesn't provide a way to control the execution. An ICE (often used with a debugger itself) can do everything the software debugger does, and trace real-time execution. Its pod replaces the processor, giving the engineer complete control of the system. A simulator allows the code to execute in a virtual environment on the host. Most simulators provide a means of writing scripts, which can simulate inputs, record outputs, and otherwise control the execution of your code. They can be useful for unit testing, especially where repeatability is desired.

In addition to these basic tools, many other programs are useful in testing code. Tools exist to capture events and play them back, which is useful in automating a procedure. Other tools simply help organise the testing of complex systems. Some help in developing a traceability matrix. Other tools provide scripting capabilities, which can speed the development of procedures. Yet other tools provide a skeleton of code, which can be modified to perform unit testing on a host, another step towards automating the process.

Literally hundreds of tools and companies are out there that can help identify the bugs we assume to be in our code. However, beware of claims that sound like a "silver bullet." In my experience, there's always some set of assumptions and conditions that limit those claims to a particular class of applications or type of testing.

Program

Now that we've covered the sorts of bugs that lurk in our code, and some of the techniques useful in finding them, you may say, "I don't have time to exhaustively test every line of code six different ways. What approach should I use to test my code? How much time and money will it cost?"

The answers to those questions are "it depends." What's the purpose of your testing? Is it to complete a mandatory activity as part of your company's software development process? Is it to ensure that the code implements all the advertised features? On the other hand, is the purpose to ensure beyond a reasonable doubt that the system in which your software runs won't kill someone because of a software error? Or destroy a multi-million dollar piece of equipment? In other words, what is the risk posed by the inevitable software bugs? Testing theory states, "all software has errors." Some errors probably will never be noticed, ever. Other errors may be catastrophic. As engineers we have to balance the risk that alleged software errors pose with the cost of finding them.

For different purposes, and different types of systems, some test strategies are more appropriate than others. Figure 1 shows which tests are generally best suited to finding certain types of errors. Knowledge about the sorts of errors that may be in a program-and the risks that they pose-can then be used to select the testing strategies that should be the most effective.

Program

None of the techniques here can be used effectively outside of a good software development process. Nor will they be of help if not performed thoroughly, with appropriate management buy-in. All the ideas presented may need to be modified to suit your specific environment. In addition, here are a few parting thoughts:

- Test early and often
- Understand the real goals of the testing effort, and make your plans accordingly
- Plan your testing efforts and stick to your plan. The time to start thinking about testing is in the concept phase, when each requirement's testability should be evaluated. These plans (and later, test procedures) continue to evolve, and should be complete when the code is ready to test
- Obtain required resources (both time and equipment) early in the development cycle
- Share your successes and failures. Learn from others what works and what doesn't
- Testers and developers are part of the same team. Work cooperatively
- Test each other's code. It's difficult to see your own mistakes

Email   Send inquiry