Strategies for Preventing and Detecting Errors in Embedded Software Development

ParaSoft®

Introduction

As every embedded software developer knows, it is extremely difficult to find and fix errors in embedded software.

One reason that debugging embedded software is so difficult is that code is typically written on the host machine, then cannot be executed or tested until it is transferred to the target system. This means that most traditional software testing techniques cannot be applied to embedded software development.

Another reason for this difficulty is that embedded software developers have considerably fewer development tools at their disposal than other software developers-- not because there is no demand for such tools, but rather because it is expensive and difficult to create development tools for this development sector. Only a relatively small number of developers produce software for embedded systems, and those who do develop embedded software work with a wide variety of target devices. As a result, this development sector is composed of a very small number of developers who are spread out across multiple targets and compilers. The size and fragmentation of this sector discourage development tool companies from supporting embedded software development. Developing for such a small, fragmented market is not cost-effective: development tools are difficult to build and port, so they typically only cater to the largest development sectors.

Consequently, embedded software developers-- developers of software in which there is typically little room for error-- are left with a monumental debugging task, and few or no tools that help them perform this task. We believe that the solution to this dilemma lies in techniques and tools that:

- **Prevent as many errors as possible:** Every error that was prevented is an error that you do not need to find and fix, and an error that has no chance of eluding testing and being shipped with the system. These benefits are especially important to embedded software developers because of the difficulty involved in debugging embedded software and the demand for error-free applications.
- **Are general, not target/compiler specific:** Software development tools that were designed to be general (i.e., not compiler/target specific) and easily portable are generally more beneficial to embedded software developers than compiler/target-specific tools are. General development tools tend to contain more advanced error detection technologies, and can be used on a variety of different projects (as developers move to new projects, they do not need to invest time and effort in learning new tools, but rather can leverage their previous investments).

Research has shown that the two most beneficial error prevention techniques are coding standards enforcement and unit testing. This paper will describe how these techniques can benefit embedded software developers and introduce software development tools that automate these techniques.

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, we will focus on C coding standards, but coding standards are also available for other languages, including C++ and Java.

To maximize your error prevention efforts, you should enforce 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 magnitude 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 magnitudes. If the developer of this code had followed the coding standard "Do not declare the magnitude of a single dimensional array in an argument declaration" (taken from Motorola's 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)
{
   .
   .
   .
}
```

Another coding standard that could help embedded software developers prevent errors is "The `const` data type should be used for pointers in function calls if the pointer is not to be changed" (also from Motorola's C coding standards). The `const` specifier guarantees that the value of the variable cannot be changed. The compiler will report an error if the value is changed when `const` is used. Following this coding standard will prevent you and other developers from mistakenly modifying the value of this variable, and thus prevent errors that may otherwise result from attempts to modify the value of this variable.

Here is yet another example of code with potential problems that could have been prevented by enforcing coding standards:

```
void Foo (int *ptr1, char *ptr2, float *ptrF)
{
    *ptr1 = 10; // possibly null pointer dereference
    ptr2 = 0;
    if( ptrF==0) {
        return;
    }
    *ptrF = 0; // OK
    return;
}
```

This code deferences a possibly null pointer; such dereferencing is a common cause of program failure. This problem could have been prevented by enforcing the coding standard "Don't pass possibly null pointers as parameters." Violations of this coding standard do not always result in an error, but it is much less time-consuming to enforce the coding standard and check if violations will cause a problem than it would be to later trace strange application behavior back to a specific line of code.

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:

```
#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.

The best way to enforce these and other coding standards is to enforce them automatically, with a tool that offers a set of meaningful industry-wide coding standards as well as a way to easily create and enforce custom coding standards. The best example of a C tool that incorporates all of these vital features is CodeWizard. CodeWizard automatically enforces a comprehensive set of built-in coding standards that prevent errors in embedded development; it also enforces any number of custom coding standards that you develop.

CodeWizard contains a set of coding standards designed specifically for C embedded developers. These coding standards were taken from C gurus as well as embedded development industry experts. Coding standards included in this special “embedded development” set include the coding standards discussed above, as well as over 75 additional coding standards such as:

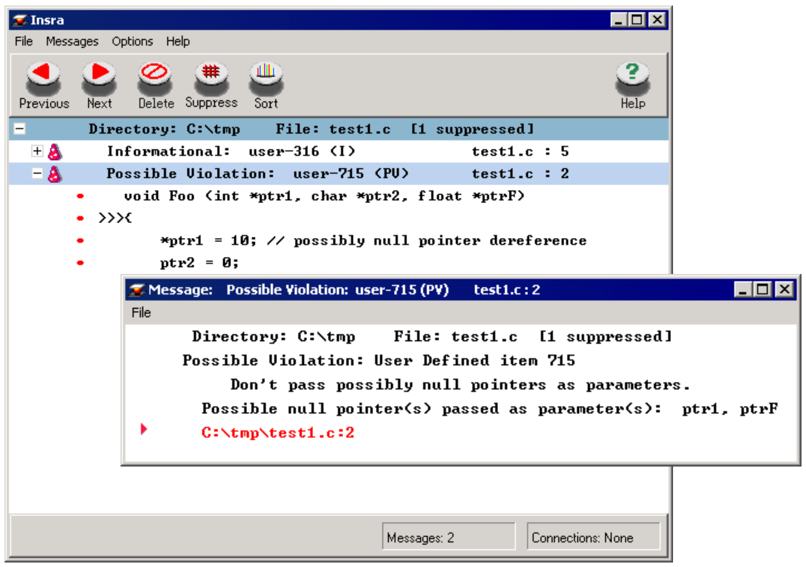
- Avoid breaks in for loops.
- Do not compare chars to constants out of char range.
- Do not check floats for equality; check for greater than or less than.
- Avoid assignment in if statement condition.
- Avoid functions that modify the global variable.
- Avoid constants out of range for the unsigned char type.
- The comma operator shall only be used in for statements and variable declarations.

- Do not initialize unsigned integer variables with signed constants.

To enforce coding standards with CodeWizard, you simply need to specify which file or files you want to check, then enter a simple command or click a button. You can also check coding standards in the background by integrating CodeWizard into your makefile.

For example, if you wanted to automatically enforce coding standards on the above file that dereferences a possibly null pointer, you could simply open the file in a supported IDE, then click the **Analyze File** button, or you could enter `codewizard test1.c` at the prompt.

CodeWizard would then analyze the file and report the following violations.


Insra
File Messages Options Help
Previous
Next
Delete
Suppress
Sort
Help
Messages: 2
Connections: None
Message: Possible Violation: user-715 (PV) test1.c:2
File

```
Directory: C:\tmp    File: test1.c   [1 suppressed]
  Informational:  user-316 (I)            test1.c : 5
  Possible Violation:  user-715 (PV)      test1.c : 2
    void Foo (int *ptr1, char *ptr2, float *ptrF)
  >>>{
        *ptr1 = 10; // possibly null pointer dereference
        ptr2 = 0;
```

```
  Directory: C:\tmp    File: test1.c   [1 suppressed]
Possible Violation: User Defined item 715
     Don't pass possibly null pointers as parameters.
  Possible null pointer(s) passed as parameter(s):  ptr1, ptrF
  C:\tmp\test1.c:2
```

Figure 1: Automatically Preventing Errors With CodeWizard

It is easy to customize CodeWizard to suit your development team's projects and priorities. CodeWizard divides its coding standards into five severity categories, so it is easy to determine which coding standards you do and do not want to enforce. Once you decide which coding standards you want to enforce, you can customize the tool by creating flexible suppression options or disabling/enabling specific coding standards.

CodeWizard also lets you create custom coding standards. With CodeWizard's RuleWizard GUI, you can create custom rules by graphically expressing the pattern that you want CodeWizard to look for as it analyzes code. Rules are created by selecting a main "node," then adding additional elements until the rule expresses the pattern that you want to check for. Rule elements are added by pointing, clicking, and entering values into dialog boxes. You can also use this tool to customize many of the built-in coding standards included with CodeWizard. You can write rules that are applicable to any C/C++ project, or you can create rules that are specific to a certain project or type of project (for example, rules for software designed for a certain type of PDA or smart phone).

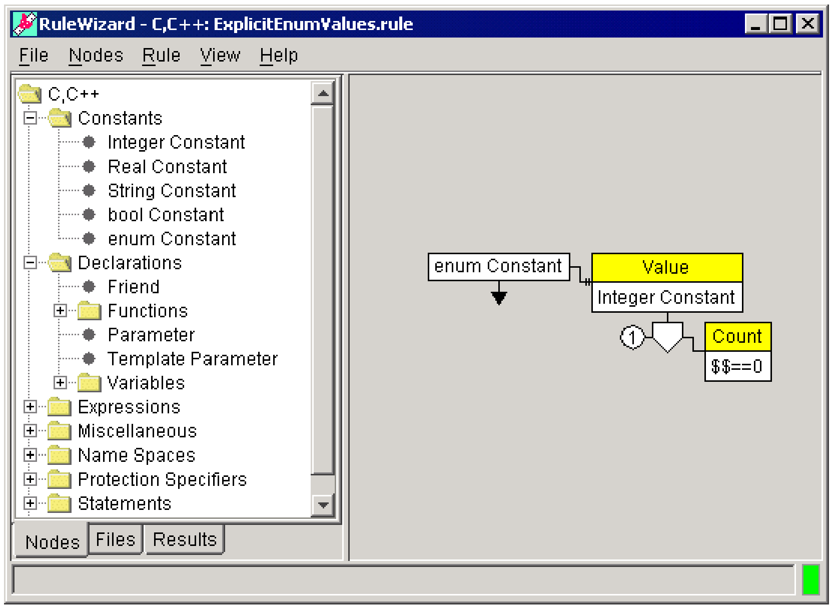


Figure 2: Creating Custom Coding Standards With RuleWizard

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 we want to concentrate on here. When we use the term “unit testing,” we are 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. Unit testing facilitates error detection by bringing you closer to the errors. Figures 3 and 4 demonstrate how unit testing does this.

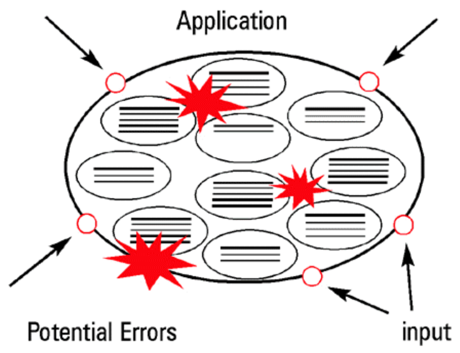


Figure 3: Application Testing

Figure 3 shows a model of testing an application containing multiple functions. The application is represented by the large oval, and the functions it contains are represented by the smaller ovals. External arrows indicate inputs. Starred regions show potential errors.

To find errors in this model, you need to modify inputs so interactions between functions will force the function to hit the potential errors. This is incredibly difficult. Imagine standing at a pool table with a set of billiard balls in a triangle at the middle of the table, and having to use a cue ball to move the triangle's center ball into a particular pocket-- with one stroke. This is how difficult it can be to design an input that finds an error within an application. As a result, developers that rely only on application testing may never reach many of the functions, let alone uncover the errors that they contain.

Testing at the unit level offers a more effective way to find errors. This is demonstrated by Figure 4.

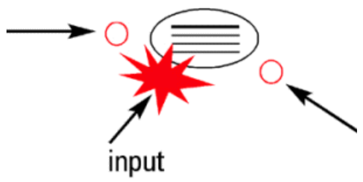


Figure 4: Unit Testing

As Figure 4 illustrates, when you test one function apart from the others, you can reach potential errors much easier because you are much closer to the errors. The difficulty of reaching the potential errors when the function is tested as an isolated unit is comparable to the difficulty of hitting one billiard ball into a particular pocket with a single stroke.

For the same reason, it is also easiest to achieve complete coverage at the unit level. While 100% coverage at the application level is an unrealistic goal, 100% coverage is indeed possible at the unit level because it is so much easier to design inputs that thoroughly cover each function when you are testing at the unit level.

The second way that unit testing facilitates error detection is by relieving you from having to wade through problem after problem to remedy what began as a single, simple error. Because bugs build upon and interact with one another, pinpointing and remedying one error at higher levels often involves finding one problem after another. When you test at a higher level, your original error is like the innermost layer of an onion, and the additional errors are like the many layers that enclose that innermost layer: you can't even see the innermost layer until you peel away every layer that envelops it. When you test every function as soon as it is compiled, errors have little chance to build upon one another and interact to cause strange behavior. To extend the onion analogy, reaching an error at the unit level is as easy as reaching an innermost onion layer that is not enveloped by any additional layers.

The most significant effect of this easier error detection is its ability to slash development time and cost. There are several reasons why unit testing can reduce development time and cost at the same time that it improves application quality. First, because errors are easier to find, you consume less time and resources finding them. Second, because you are detecting and fixing errors as soon as you have written a function, you do not need to waste time relearning the function, as you would if you had delayed testing until later in the development process. Finally, the most important reason: because functions interact with and build upon one another, fixing one error at the unit level involves changing only the original function, while fixing one error at a higher level might mean changing the design and functionality of multiple program components. The later the problem is discovered, the more code must usually be changed in order to repair that error. And as the amount of code changed increases, two other factors also increase:

- The amount of time and money required to fix each error.
- The chance of introducing new errors into the code.

Study after study confirms that the time and cost involved in finding software errors rises dramatically the later a problem is detected. Consider the following data reported by Watts Humphrey (Humphrey 1995):

- IBM: An unpublished IBM rule of thumb for the relative costs to identify software defects: during design, 1.5; prior to coding, 1; during coding, 1.5; prior to test, 10; during test, 60; in field use, 100.
- TRW: The relative times to identify defects: during requirements, 1; during design, 3 to 6; during coding, 10; in development test, 15 to 40; in acceptance test, 30 to 70; during operation, 40 to 1000 [Boehm 81].
- IBM: The relative time to identify defects: during design review, 1; during code inspections, 20; during machine test, 82 [Remus].

- JPL: Bush reports an average cost per defect: $90 to $120 in inspections and $10,000 in test [Bush].
- Freedman and Weinberg: They report that projects that used review and inspections had a ten-fold reduction in the number of defects found in test and 50% to 80% reductions in test costs, including the costs of the reviews and inspections [Freedman].

Moreover, research has also confirmed that coding errors-- the types of problems that unit testing exposes-- account for a significant portion of the total errors found and of the total cost of fixing errors. Consider this data from Steve McConnell's *Code Complete* (McConnell 1993):

- Construction errors account for 45-75 percent of all errors on even the largest projects.
- In one study of coding errors on a small project (1000 lines of code), 75 percent of the defects resulted from coding, compared to 10 percent from analysis and 15 percent from design. (Jones 1986a). This error breakdown appears to be representative of many small projects.
- A study of two very large projects at Hewlett-Packard found that the average construction defect cost 25 to 50 percent as much to fix as the average design error (Grady 1987). When the greater number of construction defects was figured into the overall equation, the total cost to fix construction defects was one to two times as much as the cost attributed to design defects.

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 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.

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).

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 without an automatic unit testing tool.

The first step in performing unit testing on embedded software is constructing a type of simulator 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.

Creating scaffolding involves creating a new function that cannot be used for anything other than testing the original function. According to Hunt and Thomas (Hunt, Thomas 2000), scaffolding should include the following features:

- A standard way to specify setup and cleanup.
- A method for selecting individual tests or all available tests.
- A means of analyzing output for expected (or unexpected) results.
- A standard form of failure reporting.

In order to test the function thoroughly and accurately, you need to design scaffolding that fully exercises the function under test; several modifications or rewrites may be required to create such scaffolding. Once the scaffolding is created, you must examine it carefully to ensure that it does not contain any errors. An error in the scaffolding can sabotage the test, but because you cannot test a function in isolation (the original problem), you cannot test the scaffolding either.

If your function references any external resources (such as external files, databases, and CORBA objects) that are not yet available or accessible, you must then create stubs that return values similar to those that the actual external resources could return. A stub is typically a table which essentially says, "For this input, return this value, and for that input, return that value." When creating these stubs, you need to choose stub return values that will redirect the program's instruction pointer:

- In whatever paths must be executed in order to test the function's functionality.
- In enough directions to provide thorough coverage of the function.

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 we 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 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 preserve precious and costly developer time because you do not need to have a developer examine the entire test suite to determine which test cases need to be run and which do not. Even if you need to add additional systems to run all of your test cases in a reasonable period of time, you will save money: it is always cheaper to add systems than to pay developers good money to perform menial tasks.

Fortunately, there are ways of integrating unit testing into your development process so that 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 C++Test to automate your unit testing. As soon as you have written and compiled a function, you can simply run it through C++Test, and with the click of a button, the tool instantly makes the function executable and testable on the host system by automatically designing any necessary scaffolding or test stubs that let you test the software independent of the hardware. You can then use this tool to perform white-box, black-box, and regression testing.

To perform white-box testing on a function, you simply tell the tool which function to test, then click a button. C++Test will then automatically:

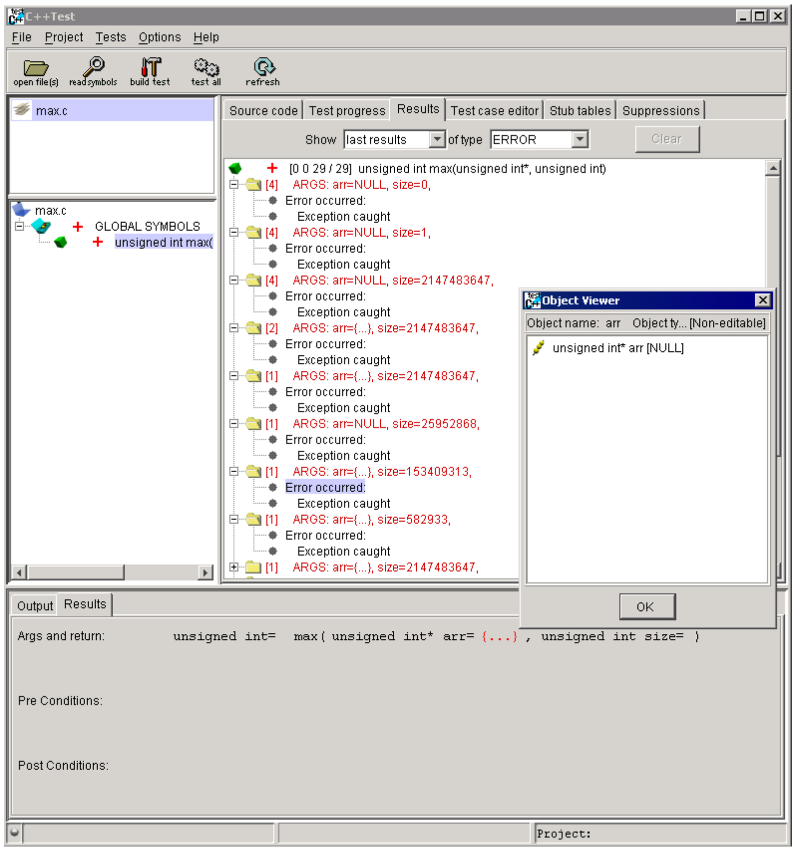
- Generate any scaffolding or stubs required for the test.
- Design and execute test cases that will thoroughly test the code's construction.
- Report errors in an easy-to-read tree structure.

You can then save test parameters and results to facilitate regression testing.

For example, let's assume that you have written the following C file and are ready to perform unit testing on it.

```
unsigned int max(unsigned int* arr, unsigned int size)
{
    unsigned int max_val = 0;
    int i;
    for (i=0; i <= size; ++i) {
        if (max_val < arr[i]) {
            max_val = arr[i];
        }
    }
    return max_val;
}
```

You can make the functions executable and testable, then expose crashes by simply loading the file into C++Test, then clicking **Test All**.


C++Test
File Project Tests Options Help
open file(s) read symbols build test test all refresh
max.c
Source code | Test progress | Results | Test case editor | Stub tables | Suppressions
Clear
Object Viewer
OK
Output | Results

Show last results of type ERROR

```
max.c
  GLOBAL SYMBOLS
    unsigned int max(
```

```
[0 0 29 / 29]  unsigned int max(unsigned int*, unsigned int)
[4]  ARGS: arr=NULL, size=0,
     Error occurred:
       Exception caught
[4]  ARGS: arr=NULL, size=1,
     Error occurred:
       Exception caught
[4]  ARGS: arr=NULL, size=2147483647,
     Error occurred:
       Exception caught
[2]  ARGS: arr={...}, size=2147483647,
     Error occurred:
       Exception caught
[1]  ARGS: arr={...}, size=2147483647,
     Error occurred:
       Exception caught
[1]  ARGS: arr=NULL, size=25952868,
     Error occurred:
       Exception caught
[1]  ARGS: arr={...}, size=153409313,
     Error occurred:
       Exception caught
[1]  ARGS: arr={...}, size=582933,
     Error occurred:
       Exception caught
[1]  ARGS: arr={...}, size=2147483647,
```

Object name: arr   Object ty... [Non-editable]

```
unsigned int* arr [NULL]
```

Args and return: `unsigned int=  max( unsigned int* arr= {...} , unsigned int size= )`

Pre Conditions:

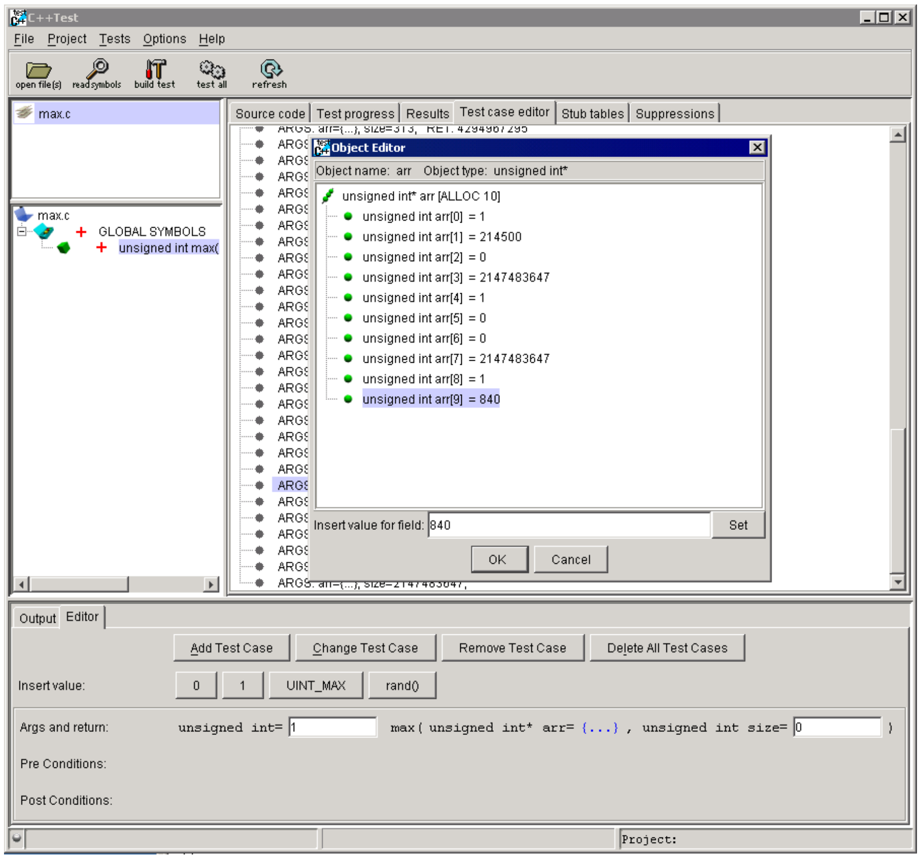
Post Conditions:

Project:

Figure 5: Automatic White-Box Testing with C++Test

When you are ready to perform black-box testing, you can start with the base black-box test suite that C++Test creates during white-box testing, then you can expand the test suite by adding your own test cases and customizing tests with user-defined stubs. You can also save black-box inputs and outcomes with a click of a button; this makes unit-level regression testing virtually effortless.

For example, you can enter a user-defined test case to the above example file by entering values in the test case skeleton and object editor.


C++Test
File Project Tests Options Help
open file(s) read symbols build test test all refresh
Source code Test progress Results Test case editor Stub tables Suppressions
Object Editor
Insert value for field:
Set
OK Cancel
Output Editor
Add Test Case Change Test Case Remove Test Case Delete All Test Cases
Insert value:
0 1 UINT_MAX rand()
Args and return:
Pre Conditions:
Post Conditions:
Project:

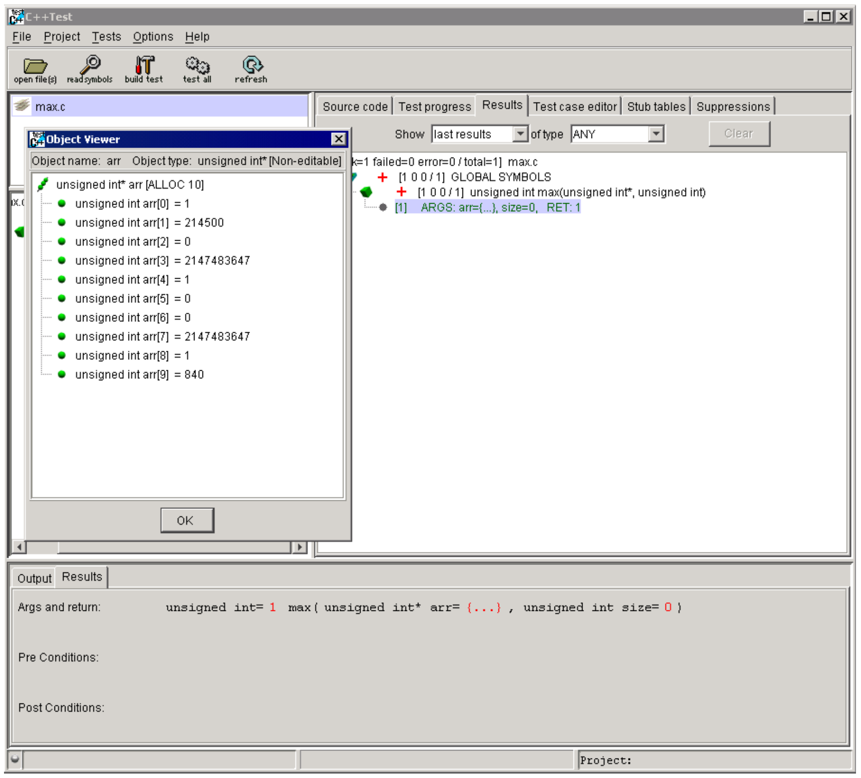
```
max.c
GLOBAL SYMBOLS
unsigned int max(

Object name:  arr    Object type:  unsigned int*
unsigned int* arr [ALLOC 10]
    unsigned int arr[0]  = 1
    unsigned int arr[1]  = 214500
    unsigned int arr[2]  = 0
    unsigned int arr[3]  = 2147483647
    unsigned int arr[4]  = 1
    unsigned int arr[5]  = 0
    unsigned int arr[6]  = 0
    unsigned int arr[7]  = 2147483647
    unsigned int arr[8]  = 1
    unsigned int arr[9]  = 840
Insert value for field: 840

Args and return:  unsigned int= 1   max( unsigned int* arr= {...} , unsigned int size= 0 )
```

Figure 6: Entering a User-Defined Test Case in C++Test

You can then run all or selected test cases with the click of a button. C++Test would report the following results for the above test case.


C++Test
File Project Tests Options Help
open file(s) read symbols build test test all refresh
max.c
Source code Test progress Results Test case editor Stub tables Suppressions
Show last results of type ANY Clear
OK
Output Results
Project:

```
Object Viewer
Object name:  arr    Object type:  unsigned int* [Non-editable]
unsigned int* arr [ALLOC 10]
    unsigned int arr[0]  = 1
    unsigned int arr[1]  = 214500
    unsigned int arr[2]  = 0
    unsigned int arr[3]  = 2147483647
    unsigned int arr[4]  = 1
    unsigned int arr[5]  = 0
    unsigned int arr[6]  = 0
    unsigned int arr[7]  = 2147483647
    unsigned int arr[8]  = 1
    unsigned int arr[9]  = 840
```

```
k=1 failed=0 error=0 / total=1]  max.c
    [1 0 0 / 1]  GLOBAL SYMBOLS
        [1 0 0 / 1]  unsigned int max(unsigned int*, unsigned int)
            [1]    ARGS: arr={...}, size=0,   RET: 1
```

```
Args and return:        unsigned int= 1  max ( unsigned int* arr= {...} , unsigned int size= 0 )

Pre Conditions:

Post Conditions:
```

Figure 7: Test Case Results

As you find errors, remember to determine the cause of each error, then design a coding standard that prevents that type of error from reoccurring. You should then fix the errors found, and re-run the same tests until all construction and functionality problems are fixed. Code with unit-level errors should not be integrated into the application.

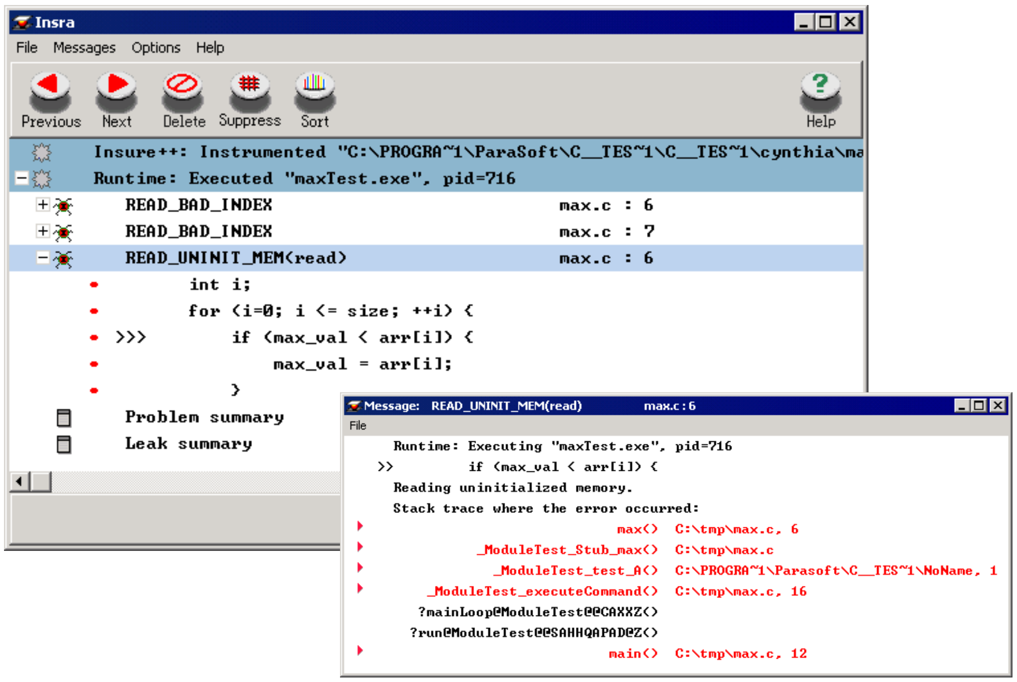
If you save your black-box and white-box test cases, you can re-run them to perform unit-level regression testing and monitor your existing code's integrity as you modify it.

You can also use C++Test to perform automatic runtime error detection at the function level. This means that you can find the following types of errors as soon as a function is compiled-- before your code is transferred to the target device:

- Memory corruption/uninitialized memory
- Memory leaks
- Memory allocation errors
- Variable definition conflicts
- I/O errors

- Pointer errors
- Library errors
- Logic errors

For example, you can instantly determine that the example file reads uninitialized memory by telling C++Test to test with Insure++ (ParaSoft's C/C++ runtime error detection tool) then clicking a button.



```
Insure++: Instrumented "C:\PROGRA~1\ParaSoft\C__TES~1\C__TES~1\cynthia\ma
Runtime: Executed "maxTest.exe", pid=716
   READ_BAD_INDEX                         max.c : 6
   READ_BAD_INDEX                         max.c : 7
   READ_UNINIT_MEM(read)                  max.c : 6
           int i;
           for (i=0; i <= size; ++i) {
   >>>         if (max_val < arr[i]) {
                   max_val = arr[i];
               }
   Problem summary
   Leak summary
```

```
Message:   READ_UNINIT_MEM(read)          max.c : 6
File
    Runtime: Executing "maxTest.exe", pid=716
  >>          if (max_val < arr[i]) {
    Reading uninitialized memory.
    Stack trace where the error occurred:
                                  max()  C:\tmp\max.c, 6
                   _ModuleTest_Stub_max()  C:\tmp\max.c
                     _ModuleTest_test_A()  C:\PROGRA~1\Parasoft\C__TES~1\NoName, 1
              _ModuleTest_executeCommand()  C:\tmp\max.c, 16
            ?mainLoop@ModuleTest@@CAXXZ()
           ?run@ModuleTest@@SAHHQAPAD@Z()
                                 main()  C:\tmp\max.c, 12
```

Figure 8: Uninitialized Memory Errors Found by Insure++ and C++Test

Conclusion

The above error prevention and error detection techniques can dramatically improve embedded software quality and reduce the time and money spent on debugging. The tools that ParaSoft offer embedded developers are designed as generally as possible so that they can easily be applied to many different types of embedded software projects. This means that you can leverage your investment and expertise in these tools and techniques as you move to new projects and different target technologies. These tools and techniques will also help you ensure that your code can easily be maintained, modified, and ported to new types of devices. In short, these tools and 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 tools and expertise needed to develop high quality applications for these technologies-- on time and on budget.

To learn more about how CodeWizard, C++Test, and other ParaSoft development tools can help your department prevent and detect errors, talk to a Software Quality Specialist today at 1-888-305-0041, or visit www.parasoft.com.

References

Humphrey, W. *A Discipline for Software Engineering* (Reading, MA: Addison-Wesley 1995).

Hunt, A. and Thomas, D. *The Pragmatic Programmer* (Reading, MA: Addison-Wesley 2000).

McConnell, S. *Code Complete* (Redmond, WA: Microsoft Press 1993).