



## Design-for-Test Strategy Eases Unit Level Testing

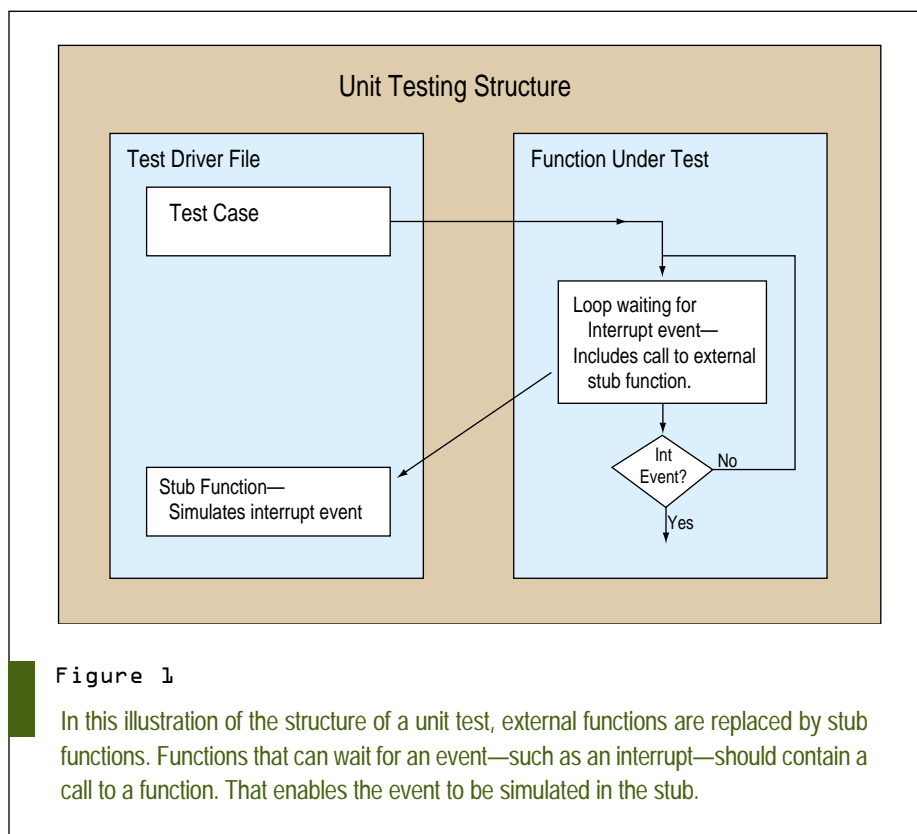
Unit level test boosts reliability, but can be costly and time consuming. By crafting code for testability up front, those hurdles can be overcome

Steve Tuttle, Chief Technology Officer  
Quality Checked Software

There's no doubt that software is playing an expanding role in today's commercial and military products. As that software grows more complex, the pressure is on for engineering teams to find ways to improve product reliability while reducing development costs. Unit level testing continues to attack one side of those efforts. It continues to be very effective in improving software reliability. Yet many teams are put off by the perception that unit level testing is both tedious and expensive.

Unit level testing verifies correct functionality of individual software units: a procedure or function in a procedural language, or a class in an object-oriented language. Any external functions called are replaced with stub (or wrapper) functions to allow full control of the test environment. A "test driver" program is written (or automatically generated) to initialize data, invoke functions under test and verify the results. Figure 1 illustrates the unit test program structure.

When unit level coding and design problems occur many development projects get bogged down at the integration phase. Each iteration of a system build requires significant time and resources. That's followed by the setup, execution and analysis of a system level test. Clearly such elaborate steps aren't an efficient method for detecting low level defects. Meanwhile, unit level error conditions are nearly impossible to simulate in a system



level test, leaving many error handling scenarios completely untested. With that in mind, low level defects should be detected at the unit development stage, before the integration phase.

### Why Unit Testing Is So Difficult?

Many software developers dismiss formal unit testing as being too difficult, too time consuming or both. There are four key reasons why:

**Missing (undocumented), incomplete or ambiguous requirements.** Software behavior cannot be verified if correct behavior is not known. Unit level requirements (not the same as system requirements) are a valuable business asset, typically maintained only in the developer's head—not a safe place for assets.

**Building the "test infrastructure" code takes time.** A complete test infrastructure includes test drivers, stub func-

## Test & Screening

tions, libraries for checking data, code parsing and coverage instrumentation tools. It does not make sense to regenerate this testing infrastructure for each project when off-the-shelf automated

testing tools can do this for you.

*Using an inappropriate tool for the job.* Many developers attempt unit testing with debugging tools, exercising their code with various input conditions. This

is primarily why the unit test process has gained a reputation for being so tedious; it is a manual process that offers neither consistency of test cases nor automated regression testing. Testing (identifying defects) and debugging (tracking down the cause of defects) are not the same task and require different tools if those tasks are to be done efficiently.

*Designing without “testability” as a goal.* This is the largest obstacle to unit testing, because the software often has to be rewritten in order to build a successful test. That presents two major problems: it takes time to redesign the software, and the modifications to support testability are typically not incorporated into the official source code archive, presenting risk that the software being tested is not the same software that will be delivered in the final product.

While important, the first reason listed above—incomplete requirements—falls outside the scope of this article. The second and third reasons are easily addressed by the selection of an appropriate off-the-shelf testing tool. It's the fourth reason—designing for testability—that relates most directly to the

```
Hdw.h:      struct hw_struct_t { /* define reg structure */
            unsigned char register1;
            unsigned char register2;
        };
            /* define value for base pointer variable */
            #ifndef UNIT_TEST
            #define HW_BASE 0x200000
            #else
            #define HW_BASE &hw_structure
            extern struct hw_struct_t hw_structure;
            #endif
            extern struct hw_struct_t *hw_base;
Hdw.c:      #include "Hdw.h" /* hw_base points to reg struct */
            struct hw_struct_t *hw_base = HW_BASE;
Example.c:  #include "Hdw.h" /* This defines the structure */
            void some_function() {
                /* perform access to hardware register */
                hw_base->register1 = 0x20;
            }
```

Figure 2

This C code example shows a structure definition that models the hardware register layout. Conditional compilation directives are then used to set the value of the structure pointer. This permits code to be tested on a host machine by defining the pointer variable to point to an instance of the register structure in host memory. In this example, the `hw_structure` variable would then be defined in the test driver file as a global variable.

## Hardware Design Can Affect Software Testability

At first glance, software testability seems purely a software issue, unaffected by the design choices made by the hardware engineers. However, in embedded systems, the feasibility of software testing is very dependent on the availability of hardware resources.

The main factors affecting software testability are the selection of the CPU type, the amount and types of memory and the availability of an I/O channel for a test results I/O stream back to the user.

**CPU Selection:** In selecting the CPU (or microcontroller chip), ensure that the maximum size of the address space is at least four times the size that the application requires. This will leave enough extra space for any memory overhead required by a testing tool.

**Memory Capacity:** Software testing tools will require extra program memory—roughly 2 to 3 times the nominal program space, to allow for instrumented code

(Figure A). In addition, plan for an extra 64 Kbytes of program memory for test related runtime libraries. In terms of RAM space, expect a typical testing tool to require 8-32 Kbytes of extra data storage over and above the requirements of the software under test. To measure code coverage with minimal impact on the execution time, allow for an extra 200-300 Kbytes of RAM for runtime storage of coverage data. If an ICE (In Circuit Emulator) is available, some of this extra memory could be mapped into the target CPU address space by the emulator.

**I/O Facilities:** In an embedded environment, a communication channel must be available to transfer test results from the target to a host machine. It may be possible to use a debugger/download I/O port for this purpose if all test-result I/O happens after the execution has completed. However, if test data will be continuously piped to a host, an I/O channel should be dedicated for this purpose.

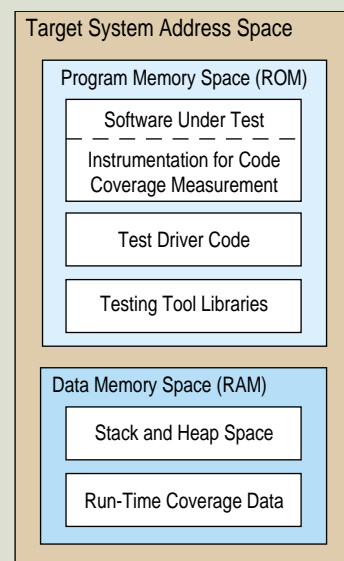


Figure  
Extra hardware resources must be available to support software testing on a target system. Shown here is the target memory usage in a unit testing context.

software development process itself. When testability is made a priority, the unit testing phase of a project becomes simpler and requires fewer resources.

### Design Characteristics That Affect Software Testability

In the context of a unit level test, a separate source file called a test driver is used to set up initial conditions, make calls to the software under test and verify the results. There are three requirements that must be met by the software under test to support the testing process:

- **Setability:** Can an external test driver initialize the data?
- **Controllability:** Can an external test driver direct the flow of control?
- **Visibility:** Can an external test driver check the data?

The first phase of a unit level test case involves setting up initial conditions. Most modern programming languages allow the hiding of internal variables from external software modules, which is in direct conflict with the requirement to initialize that internal data for testing purposes. The concept of data hiding is generally regarded as a beneficial feature of a programming language, but is actually a liability when the needs of testing are considered.

In C++, the test driver program can access private class members through the addition of a “friend” declaration to the class under test. This could be implemented as a conditional code construct in the class definition, or by the use of a testing tool that adds the friend declaration automatically.

### Variable Declarations Matter

In the C programming language, private data is generally declared as a static variable in the source file under test. To make this software testable, private data should be declared using a macro for the type modifier static. For example, use `HIDDEN` instead of using the keyword `static` in the data declaration. Then using conditional compilation, redefine `HIDDEN` to eliminate the static keyword while testing. Note that when private (static) variables are converted into a visible (global) form, a name conflict may occur with other variables in the global namespace.

This is not usually a problem because the typical unit test consists of just one or two source files, rather than the entire set of source files in the project.

Another obstacle to setting up initial conditions before test execution is the use of the type modifier `const` in the declaration of global variables. The `const` modifier tells the compiler that a variable is a constant and can only be assigned a value at the point of declaration. This violates the testing requirement for setability. The solution here is similar to the previous example of “static” data. Use a macro as a substitute for the `const` keyword.

The initialization of variables is also a problem when a developer has placed a type definition for a structure or a class inside the source file being tested, rather than in a separate header file. Most developers assume that if a variable is declared static in a source file, no other source files will need to access that data, so they place the type declaration in that same source file. But for unit testing purposes static

data will be converted to global data, to gain access to the variables. By placing the type definitions in a header file, both the test driver file and the source file under test can include that header file, and see the common type definitions.

### Control Execution Flow During Test

The second step in the test process is the execution of the function under test. The software design needs to support this execution phase by offering the test driver the opportunity to control the path of execution, including decision outcomes and loop iteration counts. Above all, the software under test needs to return control back to the test driver at the end of execution.

In embedded systems, some functions may be designed as an infinite loop. After the test driver passes control to a function that contains an endless loop, the test driver will never be able to regain control of the test. There must be loop termination condi-

## Test & Screening

tions that can be controlled by the test driver file. This could be a reference to a variable that is always true (except when testing), or a macro that can be controlled by the tester through conditional compilation.

Loops that wait for external events, such as an interrupt or the arrival of a message, should contain one or more calls to a function that can be stubbed. This stub call gives the tester an opportu-

nity to alter the flow of control by simulating the event in the body of the stub function. As shown in Figure 1, the test driver can simulate the external event by modifying global data or changing status that is returned from the stubbed function.

In general, the use of functions like *exit()*, that never return control, should be minimized. This aborts the flow of control and prevents the test driver from completing its task. The use of these types of func-

tions should be limited to only top-level modules. Lower level units should propagate error conditions up to the top-level module, rather than using a function like *exit()* directly.

Another potential source of problems is embedded target systems code that accesses hardware I/O registers. This usually causes memory protection errors when that code is tested in a host environment such as Windows or UNIX. The access to I/O registers should be done through the use of a pointer to a structure variable, rather than directly accessing an absolute address in memory. It's best to use a structure definition that models the hardware register layout, and then use conditional compilation directives to set the value of the structure pointer. This allows the software to be tested on a host machine by defining the pointer variable to point to an instance of the register structure in host memory. An example along those lines is shown in the code listing in Figure 2.

### Checking Output Data After Test

The third step in the test process is verifying the results data after execution of the software under test. This step requires visibility to data that has been modified—or not modified. The data-hiding features of the programming language can be a hindrance in this phase of testing. Follow the same guidelines that were presented in the discussion of the “setability” requirements.

Unit level testing is an effective way to increase software reliability, and should always be performed in parallel with the software design and implementation. Attempting to use a debugger as a unit level testing tool is inefficient and a misapplication of that tool. The developer needs to use a tool specifically designed for unit testing to complete the job in a timely fashion and gain the benefits of consistent, automated regression testing. Most importantly, the software must be designed with testability as a primary goal, or the result will be time lost from code rework just to make testing feasible. ■■

Quality Checked Software  
Beaverton, OR.  
(503) 645-5610.  
[www.qcsltd.com].