Black-box, White-box Testing | 9.1 Debuggin... https://courses.edx.org/courses/UTAustinX/UT...
(https://www.edx.org)
UTAustinX: UT.6.01x Embedded Systems - Shape the World
KarenWest (/dashboard) ▼
Courseware (/courses/UTAustinX/UT.6.01x/1T2014/courseware) Course Info (/courses/UTAustinX/UT.6.01x/1T2014/info)
Discussion (/courses/UTAustinX/UT.6.01x/1T2014/discussion/forum) Progress (/courses/UTAustinX/UT.6.01x/1T2014/progress)
Questions (/courses/UTAustinX/UT.6.01x/1T2014/a3da417940af4ec49a9c02b3eae3460b/)
Syllabus (/courses/UTAustinX/UT.6.01x/1T2014/a827a8b3cc204927b6efaa49580170d1/)

Black-box testing is simply observing the inputs and outputs without looking inside. Black-box testing has an important place in debugging a module for its functionality. On the other hand, **white-box testing** allows you to control and observe the internal workings of a system. A common mistake made by new engineers is to just perform black box testing. Effective debugging uses both. One must always start with black-box testing by subjecting a hardware or software module to appropriate test-cases. Once we document the failed test-cases, we can use them to aid us in effectively performing the task of white-box testing.

A print statement is a common example of a debugging instrument. Using the editor, one adds print statements to the code that either verify proper operation or illustrate the programming errors. If we test a system, then remove the instruments, the system may actually stop working, because of the importance of timing in embedded systems. If we leave debugging instruments in the final product, we can use the instruments to test systems on the production line, or test systems returned for repair. On the other hand, sometimes we wish to provide for a mechanism to reliably and efficiently remove all instruments when the debugging is done. Consider the following mechanisms as you develop your own unique debugging style.

Help

• Place all instruments in a unique column, so you can easily distinguish instruments from regular programs.
• Define all debugging instruments as functions that all have a specific pattern in their names. In this way, the find/replace mechanism of the editor can be used to find all the calls to the instruments.
• Define the instruments so that they test a run time global flag. When this flag is turned off, the instruments perform no function. Notice that this method leaves a permanent copy of the debugging code in the final system, causing it to suffer a runtime overhead, but the debugging code can be activated dynamically without recompiling. Many commercial software applications utilize this method because it simplifies “on-site” customer support.
• Use conditional compilation (or conditional assembly) to turn on and off the instruments when the software is compiled. When the assembler or compiler supports this feature, it can provide both performance and effectiveness.

The emergence of concurrent languages and the increasing use of embedded real-time systems place further demands on debuggers. The complexities introduced by the interaction of multiple events or time dependent processes are much more difficult to debug than errors associated with sequential programs. The behavior of non-real-time sequential programs is reproducible: for a given set of inputs their outputs remain the same. In the case of concurrent or real-time programs this does not hold true. Control over repeatability, event selection, and event isolation is even more important for concurrent or real-time environments.

CHECKPOINT 9.2

Consider the difference between a runtime flag that activates a debugging command versus an assembly/compile-time flag. In both cases it is easy to activate/deactivate the debugging statements. For each method, list one factor for which that method is superior to the other.

Hide Answer

Runtime debugging can be activated in final production systems. Runtime debugging is quicker to activate/deactivate because an edit/assemble/download cycle is not needed. Assembly-time debugging produces a final production system that runs faster and requires less memory.

CHECKPOINT 9.3

What is the advantage of leaving debugging instruments in a final delivered product?

Hide Answer

We are sure we debugged the exact system that is being manufactured. The debugging statements can be used to evaluate the proper operation of systems before they are shipped. The instruments can also be used to diagnose and repair systems.

Observation: There are two important components of debugging: having control over events and being able to see what is happening. Remember: **control** and **observability**!

Common Error: The most common debugging mistake new programmers make is to simply observe the overall inputs and outputs system without looking inside the device. Then they go to their professor and say, “My program gives incorrect output. Do you know why?”

Help