Debugging Device Drivers

Can you name a developer that can develop a bug-free software unit excluding “Hello World” program? I thought not. The main reason that the development cycle of software is such a lengthy process is errors in the code implementation, better known as “bugs” (with “debugging” becoming the word to describe the process of sieving these bugs out of the software). The end result of the existence of bugs in software is that the executable will fail at one time or another. The complexity of developing software that manages hardware is directly accountable for bugs that are manifold more complex and hard to fix. The most difficult bugs to fix are bugs that occur infrequently and cannot be easily reproduced. These are usually the logical bugs rather than the coding bugs. Kernel mode device drivers present the added danger of crashing the system if failure occurs in their code.

In this chapter

- Overview of the available tools
- Using the simplest tools
- Debug zones
- Kernel debugger
- Hardware assisted debugging
- Post mortem debugging

Overview of Debugging Tools and Techniques

Writing robust code is the key to catching and fixing bugs. Adding error handling code whenever there is a possibility of an error occurring is not just good coding practice but is a must while writing device driver code. Whenever you call an API check the return values and handle error recovery. The same applies to code that manipulates memory and memory pointers, as well as arithmetic computations that may fail, like the divide by zero classic. Having said that, a variety of debugging tools are available to debug device drivers because either coding bugs or logical bugs will still be in the code (at least in my code there are).
Debugging Techniques

The simplest of all debugging techniques is outputting debug messages to a debug shell. This technique is extremely valuable for debugging software such as device drivers. This technique is least intrusive as it does not stop the execution and flow of the device driver. All Windows CE versions including Windows Embedded Compact 7 provide an extremely helpful tool to output conditionally debug messages using debug zones. This is helpful because it curbs the barrage of messages that are displayed and therefore allowing you to view messages related only to a specific area of code you are debugging. So for example if you are debugging a device driver’s IST, you can limit the debug messages to just the ones that are triggered in the IST code so you don’t get hundreds of messages from every function in the device driver.

The other techniques involve software and hardware debuggers. While software debuggers such as kernel debugger is very useful when debugging initialization, opening a device driver instance and de-initializing and closing device driver instances, it may be quite insufficiently capable to debug device driver code that handles I/O data transactions. This would require hardware debuggers that take advantage of chip debugging capabilities.

Debugging Tools

The tools range from tools provided with Platform Builder such as

- Debug Zones
- kernel debugger
- Target control

Lauterbach’s TRACE32, is the third-party tool that interfaces with the debugging capabilities built in the CPU hardware.

- TRACE32 debugger
- eXDI hardware debugging driver by Lauterbach

Simple and Effective Debugging Techniques

As mentioned above, there are quite a few debugging techniques; however, to be effective and efficient it is wise to match the technique to the specific code that we debug. Device drivers are especially sensitive to debugging techniques because of the tight interaction with the hardware and timing sensitivity may not always comply with stopping the flow of execution using break points and examine the code trying to isolate bugs.

While it is fair to assume that initialization and de-initialization code should present no problem using a software debugger such as kernel debugger and set breakpoints within functions such as the XXX_Init entry point, it may very well be an issue if we have to debug the device driver’s IST. To debug timing sensitive code and code that directly addresses hardware registers we may prefer to resort to less obtrusive techniques such as debug messages or hardware tracing techniques.

While debugging device drivers, it is best to load the device driver manually. This means that instead of allowing Device Manager to load the device driver while the system is booting, and forcing you to perform the debugging when the system loads the device driver, you want to control loading when you are ready to debug. To do this, simply create a small application that will call ActivateDeviceEx which loads the device driver.