Microsoft

# Device Driver Best Practices in Windows Embedded Compact 7

Douglas Boling

Boling Consulting Inc.

Windows Embedded Compact 7

# About Douglas Boling

- Independent consultant specializing in Windows Mobile and Windows Embedded Compact (Windows CE)
  - On-Site Instruction
  - Consulting and Development
- Author
  - Programming Embedded Windows CE
    - Fourth Edition

# Agenda

- Basic drivers
- General Tips
- Threading and Drivers
- Stable Drivers

# What is a Windows Embedded Compact Driver?

- It's a DLL
  - Loaded by the Device Manager
- Kernel mode or User mode drivers
  - Kernel mode drivers
    - Faster
  - User mode drivers
    - More secure
- Most drivers expose a "Stream Interface"
  - How the OS interfaces with the driver

# Major Stream Interface Functions

- xxx_Init xxx_PreDeinit xxx_Deinit
  - Called when initializing/deinitializing the device driver
- xxx_Open xxx_PreClose xxx_Close
  - Called when an application opens and closes a device
- xxx_Read xxx_Write xxx_Seek
  - Called to read and write data to and from the device
- xxx_IOControl
  - Catchall API for functions not anticipated by standard interface

# “Plug and Play” Loading?

- Drivers are loaded (and unloaded) as needed
  - Boot time
  - Device Discovery
    - USB Device plug in / remove
    - SD Card insert / remove
    - PC Card insert / remove
  - Application request
- OS ‘finds’ drivers using the registry

# Driver Tips

# Use Debug Zones

- Debug Zones are runtime configurable diagnostic messages
  - Configurable at
    - Compile time
    - Load time
    - Interactively
- Zones in all the Microsoft code
  - Use to learn what is going on
  - Zones allow your driver to be debugged
- Take care with performance using too many messages

# Use the Registry

- Interrupt thread priority
- System Interrupt / hardware interrupt mapping
- Base Address of hardware
- Size of hardware window(s)

# Using the Registry

- The xxx_Init function is passed the name of the Active key
- Open this key and read the “Key” value
  - This is the name of the registry key that caused the driver to load
  - This can be
    - \drivers\builtin\<drivername>
    - \drivers\usb\clientdrivers\<drivername>
    - Others.  All under HKEY_LOCAL_MACHINE
- Store configuration information under this key

# xxx_Init Code Must Be Quick

- The boot sequence serializes load of drivers
  - One thread calls each _Init function in sequence
- Don't block
  - Major parts of the OS API isn't available at driver load during boot
  - Kernel, Filesys, some driver support available
  - GWES, Shell, everything else not available
- Don't spend time waiting on hardware
  - Spin off separate thread to wait on hardware
  - Fail open of device until secondary thread completes wait

# Never Use Global Data in a Driver

- Drivers are DLLs loaded by the device manager
- Multiple instances of the driver can be loaded
  - But only one copy of the code is loaded
- Global data will be shared across all instances
  - Probability not what you want
- Use ‘handle’ returned by xxx_Init to store ptr to instance data
  - Handle is passed back in other calls (Open, DeInit, PreDeinit)
  - Open returns ‘handle’ that can is returned in other calls

# Quick Note on Performance

- Drivers are DLLs
- System notifies DLLs when threads are created and destroyed
- Typically, drivers don't need this information
- In LibMain,  in Process Attach notification call

```
BOOL DisableThreadLibraryCalls (HMODULE hLibModule);
```

# Threading and Drivers

- Drivers are implicitly multithreaded
  - Multiple applications can call a driver at the ‘same time’
- Drivers must protect internal data structures from corruption
  - Use Critical Sections to protect shared data structures
- Drivers must have a method of unblocking blocked threads
  - When the driver closes
  - When the application unexpectedly terminates

# Use PreClose and PreDeinit

- Allows driver to free threads blocked within driver code
- PreClose is called before Close
  - Driver should unblock any blocked threads within driver
  - Driver should cancel any pending IO
- PreDeinit is called before Deinit
  - Allows driver to unblock any threads
  - Allows driver to inform others it is unloading
    - Driver already unhooked from driver list when Deinit called

# Unexpected Application Termination

- Applications that die unexpectedly can cause problems
- Application threads blocked with drivers can prevent process from being removed from memory
- When an application is terminated
  - Drivers will receive IOCTL_PSL_NOTIFY IoControl call
    - Once for each time driver opened by application
  - Driver will then receive one PreClose call

# Know the Thread Priority APIs

# Priority Bug 1

- What is wrong with this code?

```
// save current priority
int oldPri = GetThreadPriority (hThread);

// Elevate our thread priority
CeSetThreadPriority (hThread, 100);

// Do work...


// Restore original priority... (really?)
CeSetThreadPriority (hThread, oldPri);
```

# Priority Bug 1 Answer

```
// save current priority (typically normal == 3)
int oldPri = GetThreadPriority (hThread);

// Elevate our thread priority
CeSetThreadPriority (hThread, 100);



// Do work...



// Using saved priority (3) with new API. BADBAD!
CeSetThreadPriority (hThread, oldPri);
```

- Code mixing new and old APIs, thread ends up at priority 0 - 7

# Bug 1, Proper Code

```
// save current priority
int oldPri = CeGetThreadPriority (hThread);

// Elevate our thread priority
CeSetThreadPriority (hThread, 100);



// Do work...



// Properly restoring original priority
CeSetThreadPriority (hThread, oldPri);
```

- Moral:  Don’t mix the priority APIs!

# Priority Bug 2

- What is wrong with this code?

```
CeSetThreadPriority (h, THREAD_PRIORITY_NORMAL);
```

# Priority Bug 2 Answer

- Using constants defined for SetThreadPriority

```
CeSetThreadPriority (h, THREAD_PRIORITY_NORMAL);
```

- Defines are used by SetThreadPriority, not CeSetThreadPriority

```
#define THREAD_PRIORITY_TIME_CRITICAL        0
#define THREAD_PRIORITY_HIGHEST              1
#define THREAD_PRIORITY_ABOVE_NORMAL         2
#define THREAD_PRIORITY_NORMAL               3
#define THREAD_PRIORITY_BELOW_NORMAL         4
#define THREAD_PRIORITY_LOWEST               5
#define THREAD_PRIORITY_ABOVE_IDLE           6
#define THREAD_PRIORITY_IDLE                 7
```

# Basic Steps to Stable Drivers

- You must check return codes
- You must check memory allocations
- Never trust a pointer
  - Most data is passed to the driver via pointers
  - There is sure way to test the validity of a pointer
  - The driver must be able to tolerate a bad pointer
- The OS catches all thread exceptions at the call to the driver
  - The driver should catch them first

# The __try __except Block

```
__try
{
   // guarded code
}
__except ( expression )
{
   // exception handler code
}
```

- Expression values
  - EXCEPTION_CONTINUE_EXECUTION
  - EXCEPTION_CONTINUE_SEARCH
  - EXCEPTION_EXECUTE_HANDLER

# Use __try __except

- Wrap every entry point in your code
  - Callbacks
  - DLL entry points
  - Driver entry points
  - Around all accesses to buffers passed to the driver
- Don’t use C++ exception handling in drivers / OAL code

# What is Wrong Here?

```
CRITICAL_SECTION cs;
__try
{
    EnterCriticalSection (&cs);

    // Some code here

    LeaveCriticalSection (&cs);
}
__except (EXCEPTION_EXECUTE_HANDLER)
{
    // Process exception
}
```

# The __*try* __*finally* Block

```
CRITICAL_SECTION cs;
__try
{
    EnterCriticalSection (&cs);

    // Some code here


}
__finally
{
    LeaveCriticalSection (&cs);
}
```

# The __try __finally Block

```
CRITICAL_SECTION cs;
__try
{
    EnterCriticalSection (&cs);

    // Some code here
    __leave;

    // Code here not executed

}
__finally
{
    LeaveCriticalSection (&cs);
}
```

# Use __try __finally

- __finally clause always executes
  - If you fall out the end of the __try block
  - If you return in the middle of the __try block
  - If an exception occurs
- Don’t return out of a __try, __finally block
  - It generates lots of code
    - When it works at all
  - Use the __leave keyword

# Much Better Code

```
CRITICAL_SECTION cs;
__try
{
    EnterCriticalSection (&cs);
    // Some code here
}
__except (EXCEPTION_EXECUTE_HANDLER)
{
    // Exception handling code here
}
__finally
{
    LeaveCriticalSection (&cs);
}
```

# Good Drivers Are Written in C not C++

- I'm not a language bigot
  - I just speak from experience
- One of the goals of C++ is to obfuscate interior function
  - If the hardware is new (or failing) its this interior function that is failing

"In other words, the only way to do good, efficient, and system-level and portable C++ ends up to limit yourself to all the things that are basically available in C. "

Linus Torvalds

# Summary

- Basic stream interface makes drivers simple
- Use the registry to query configuration information
- Check xxx_Init performance
- Take care to protect data structures from thread conflicts
- NEVER TRUST A POINTER!

# Questions…

Doug Boling

Boling Consulting Inc.

www.bolingconsulting.com

dboling @ bolingconsulting.com

Microsoft®