UEFI Driver Development Training Testing and Debugging

Fang Hua
UEFI Development
Intel

Agenda

- EFI Shell Testing
- Debugging code statements
- COM cable usage
- American Arium usage
- Testing methodologies

EFI Shell usage

- The simplest way to test UEFI Drivers
  - Pros
    - No hardware to install
    - Can control driver selection process
    - Some additional tools exist (i.e. DrvCfg)
  - Cons
    - No source level debugging
    - Extra code required for some (i.e. printf())
    - In NT32 emulator Windows will interfere.
  - Hints
    - The '-b' command line parameter stops each page and waits

Testing Main Driver Functions

- DH
- load / unload
- connect / disconnect / reconnect
- DrvCfg / DrvDiag

Testing Driver Functions

Driver.c

ComponentName.c

```
GetDriverName()
GetControllerName()
```

DriverConfiguration.c

DriverDiagnostics.c

```
Shell> dh –d DriverHandle
Shell> Drivers
```

Testing Driver Functions

Driver.c

ComponentName.c

```
GetDriverName()
GetControllerName()
```

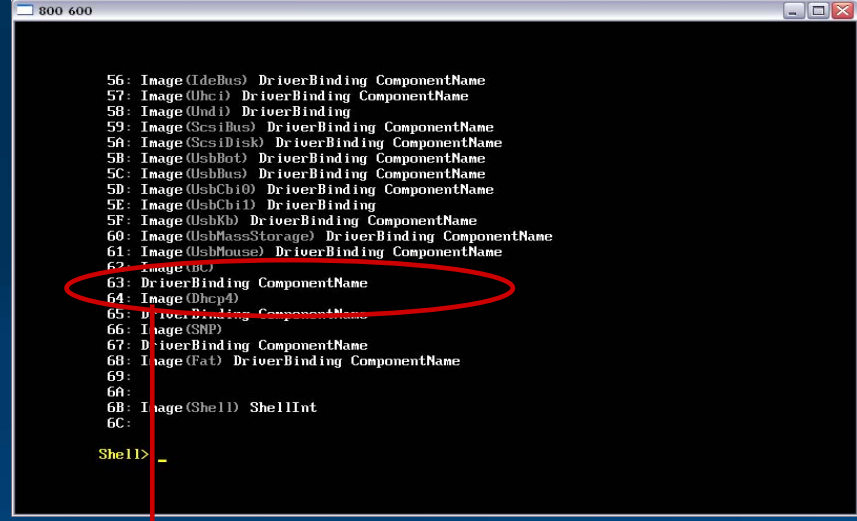
DriverConfiguration.c

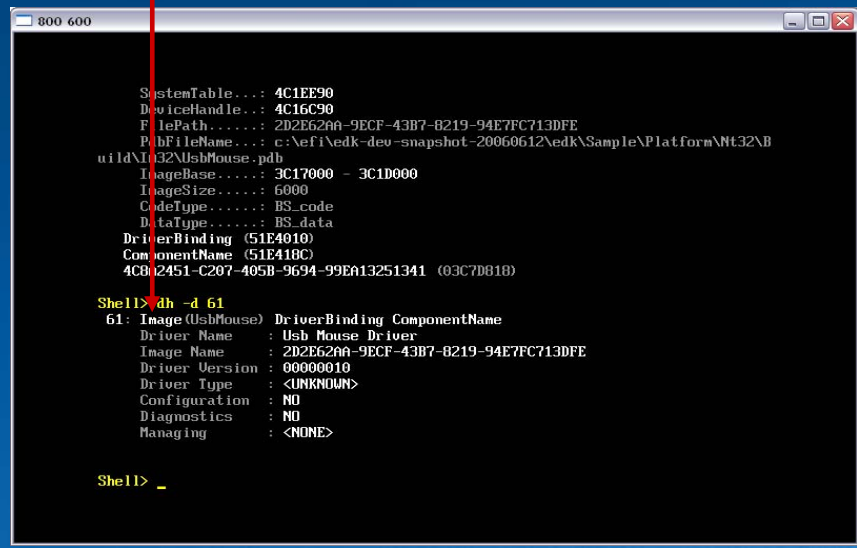
DriverDiagnostics.c

```
Shell> dh –d DeviceHandle
Shell> Devices
Shell> DevTree
```

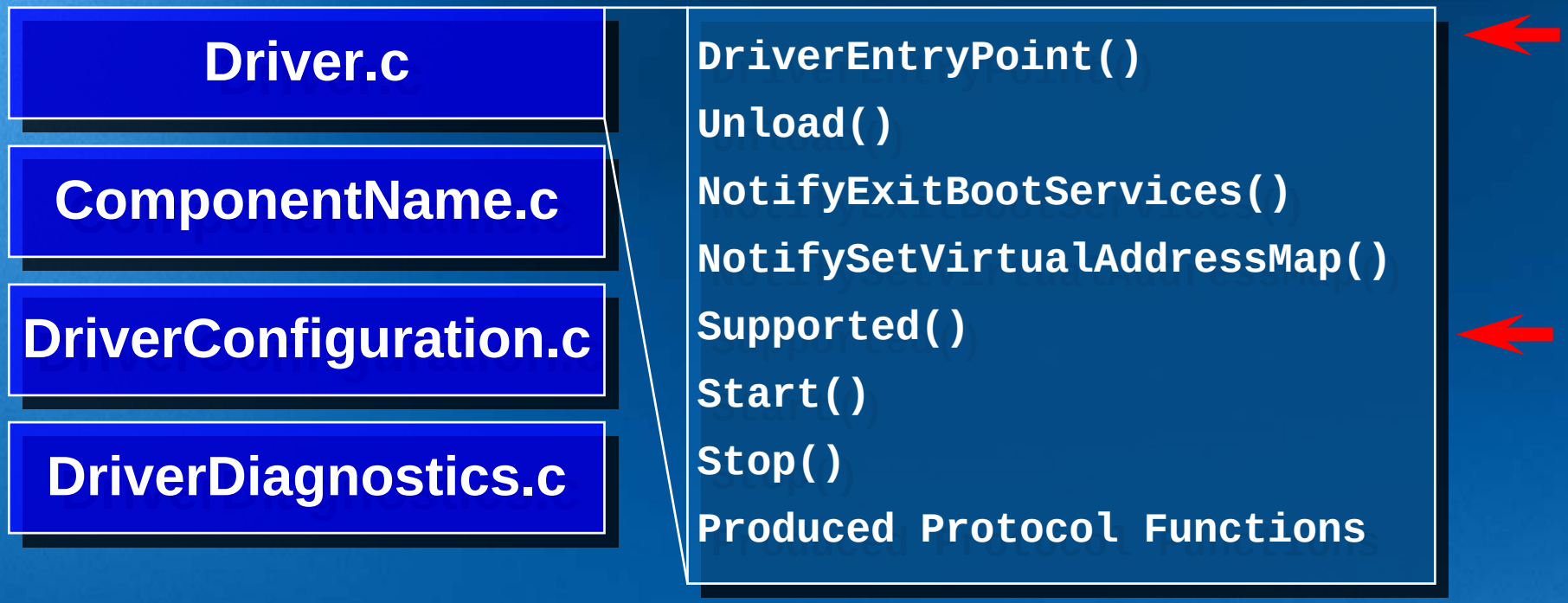
DH

- Shell> DH
  - Displays a list of handles allocated in the system with each device's handle identifier (i.e. 61)
- Shell> DH –d [handle]
  - Displays details
    - **Verify ComponentName functionality**
    - Image Info
    - Protocols consumed





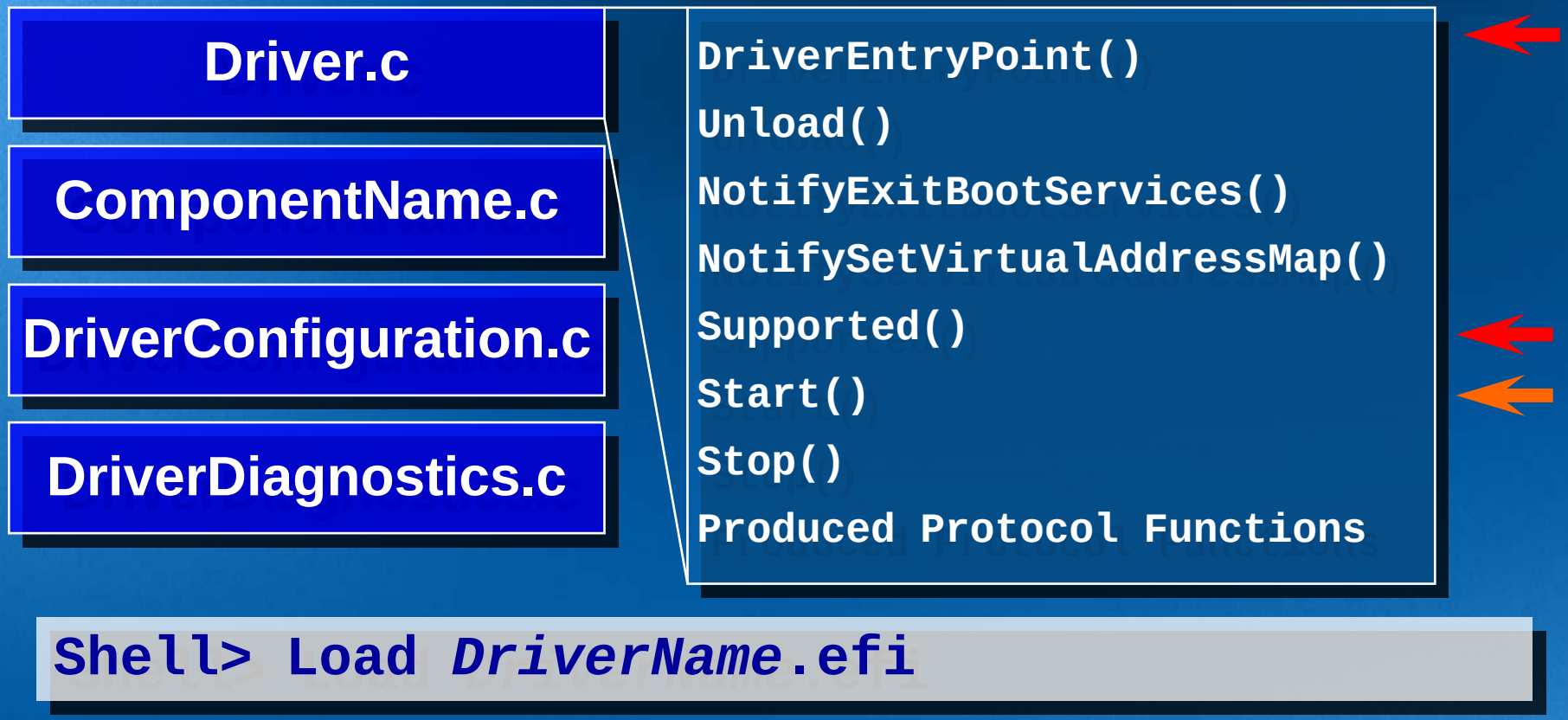
Testing Driver Functions




```
Shell> Load –nc DriverName.efi
```

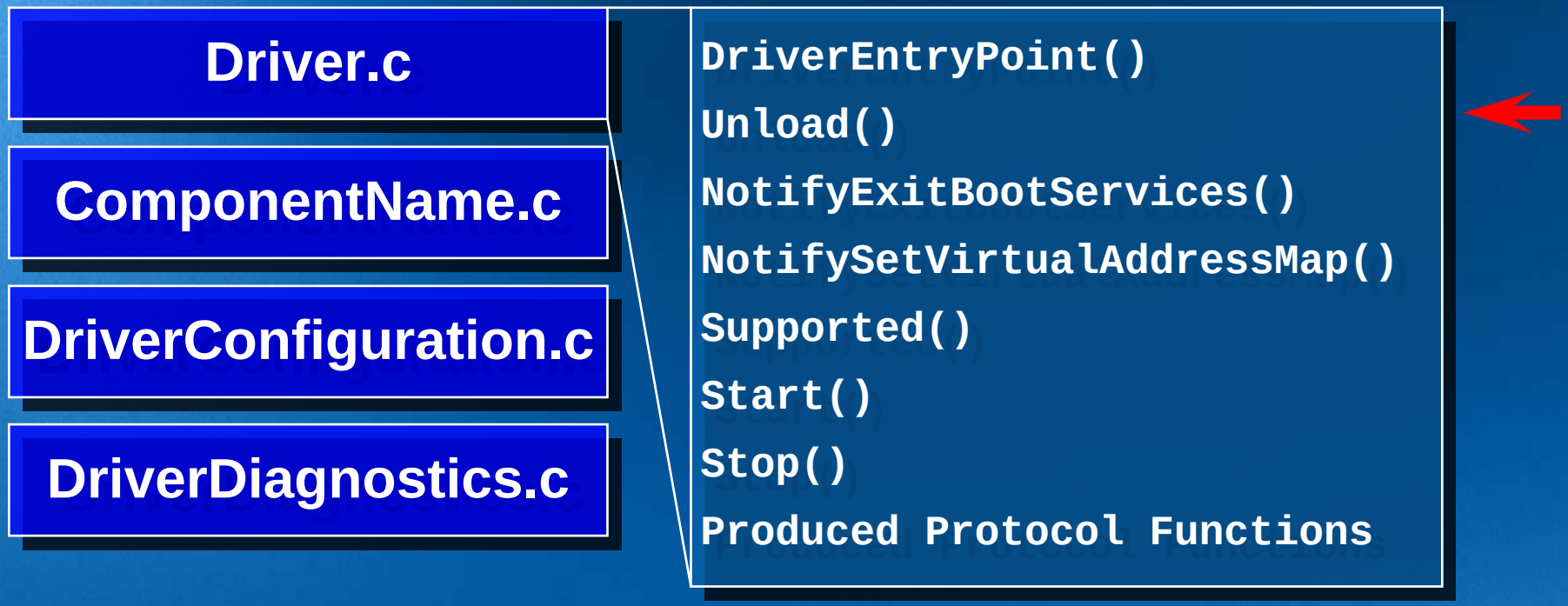
- Other drivers may connect, but not that one

Testing Driver Functions



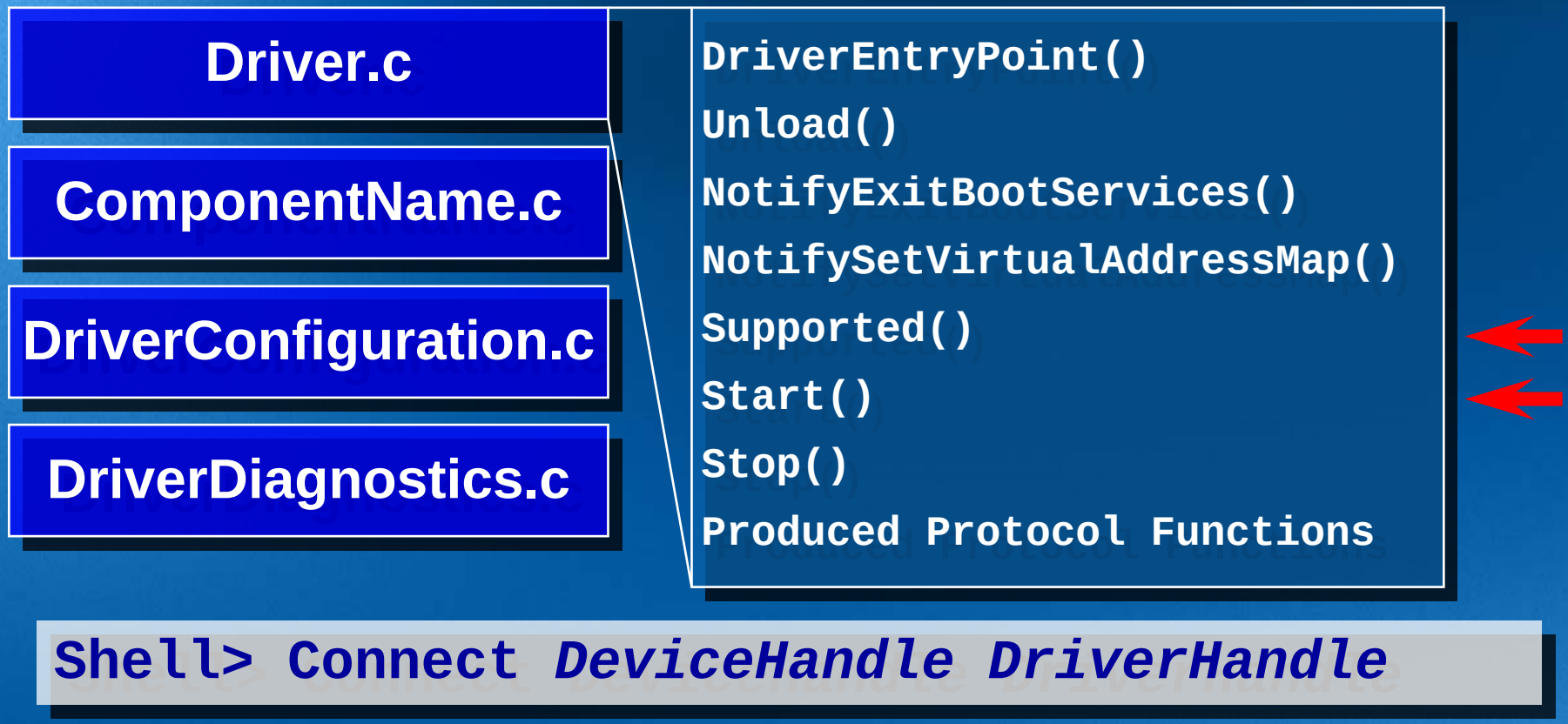

```
Shell> Load DriverName.efi
```

Testing Driver Functions



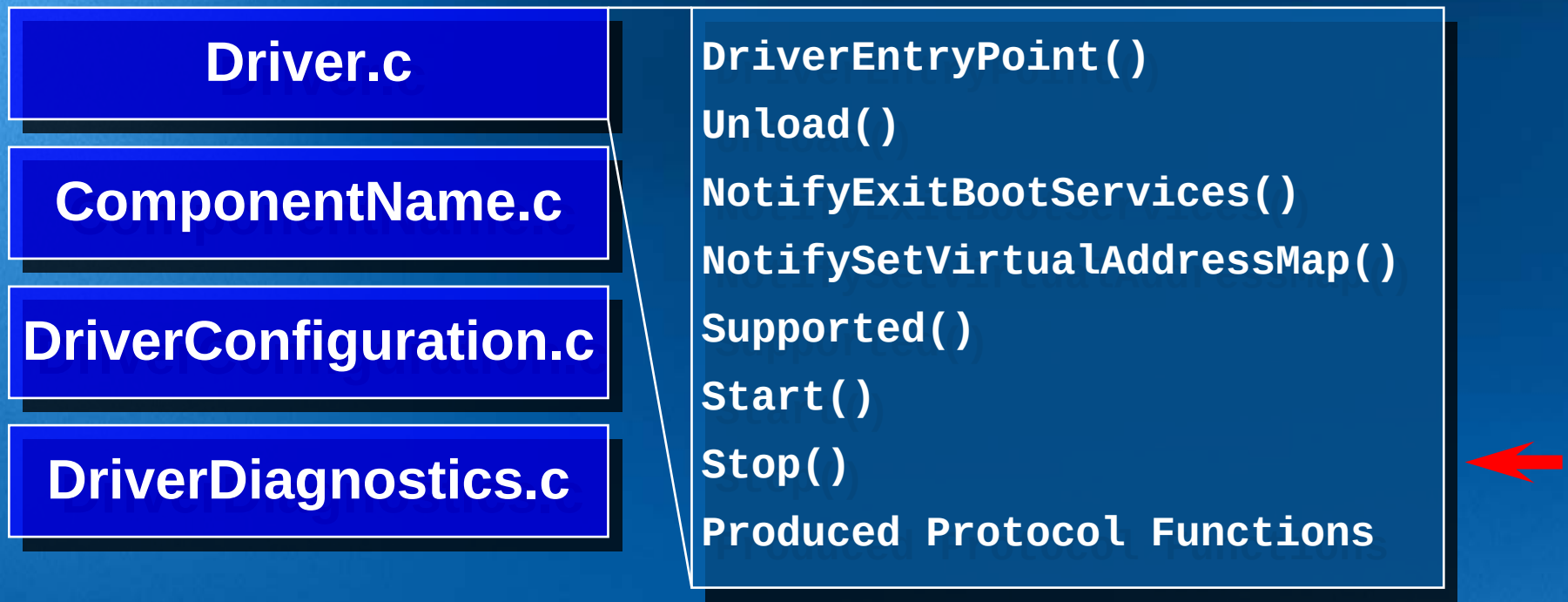

```
Shell> Unload DriverHandle
```

Testing Driver Functions



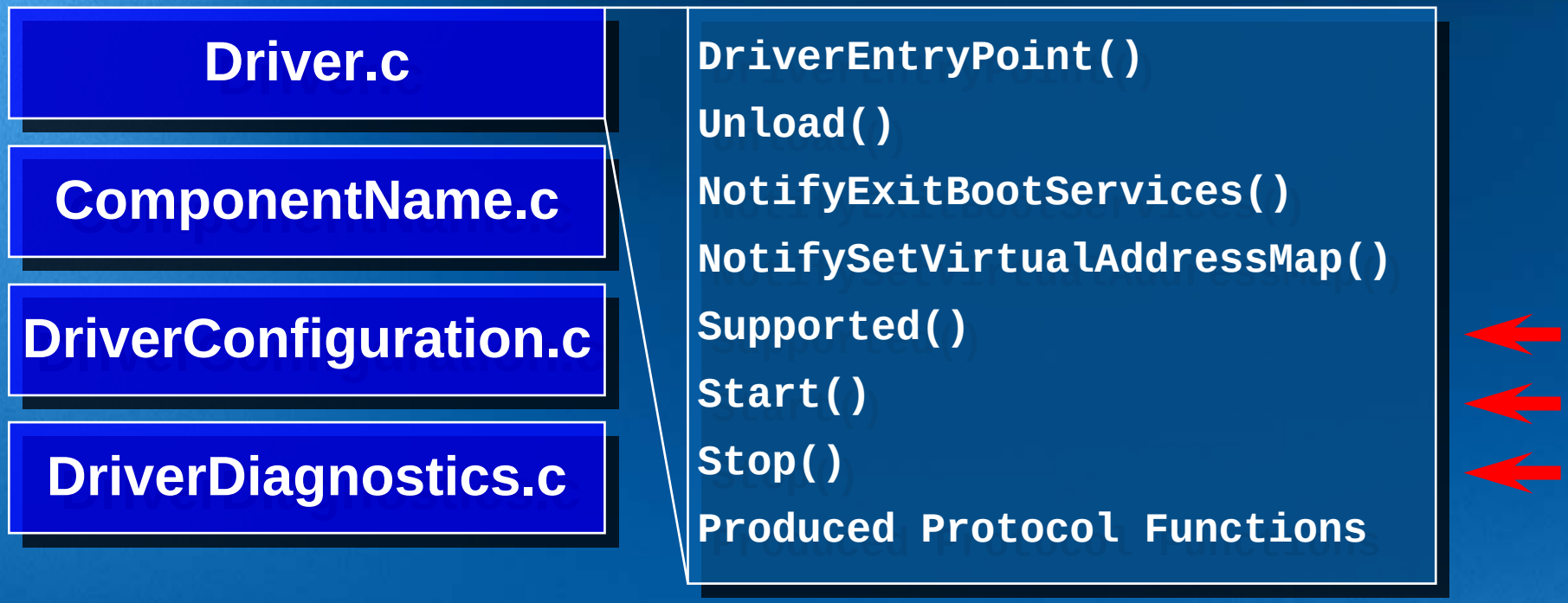

```
Shell> Connect DeviceHandle DriverHandle
```

Testing Driver Functions



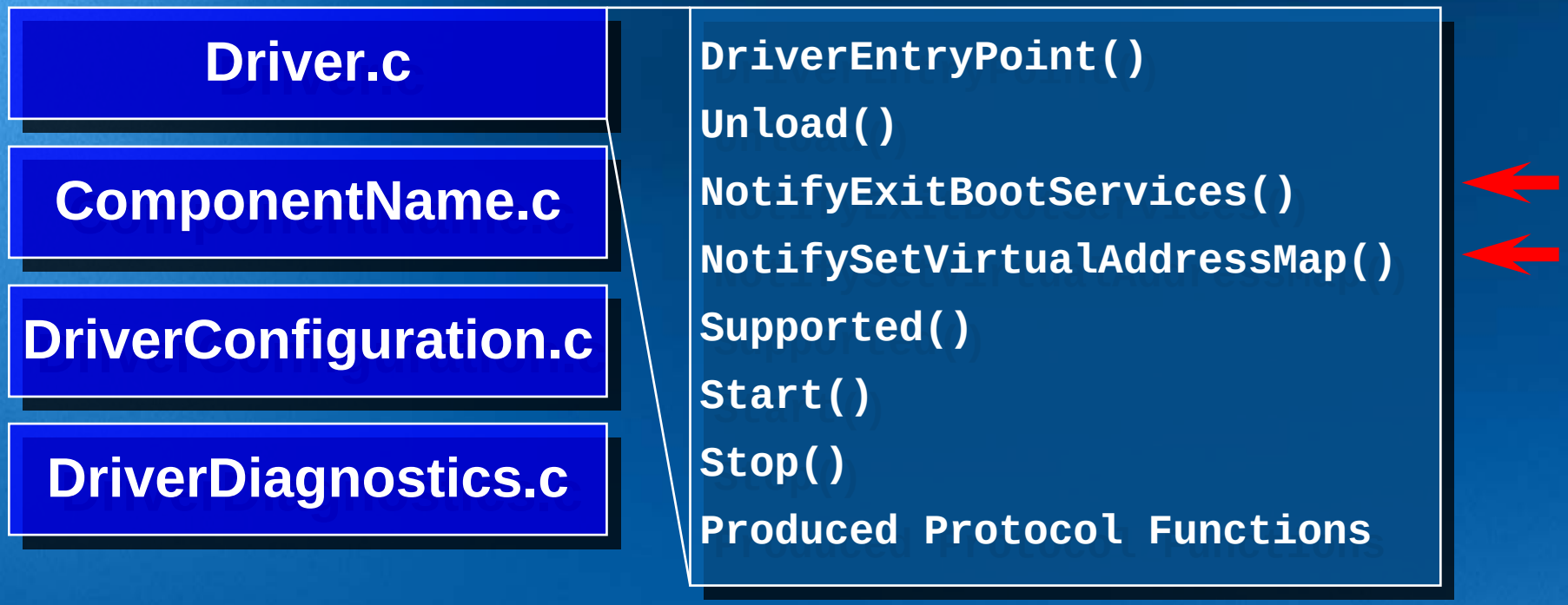

```
Shell> Disconnect DeviceHandle
```

Testing Driver Functions




```
Shell> Reconnect DeviceHandle
Shell> Reconnect -r
```

Testing Driver Functions



Install and Boot an UEFI Compliant Operating System

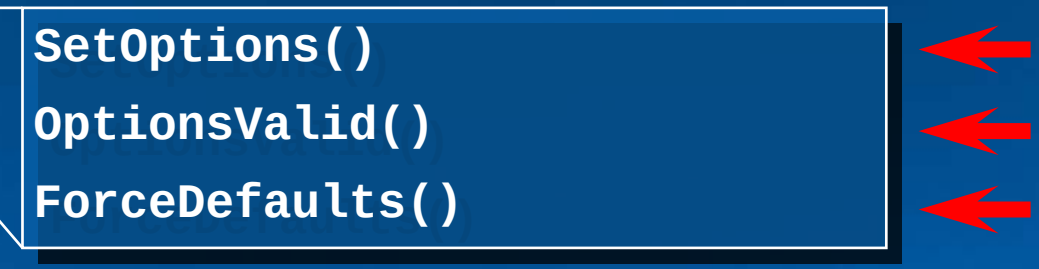
Testing Driver Functions

Driver.c

ComponentName.c

DriverConfiguration.c

DriverDiagnostics.c




```
Shell> DrvCfg –s DriverHandle DeviceHandle
Shell> DrvCfg –v DriverHandle DeviceHandle
Shell> DrvCfg –f DriverHandle DeviceHandle
Shell> DrvCfg –s DrvHndl DevHndl ChildHndl
```

Testing Driver Functions

Driver.c

ComponentName.c

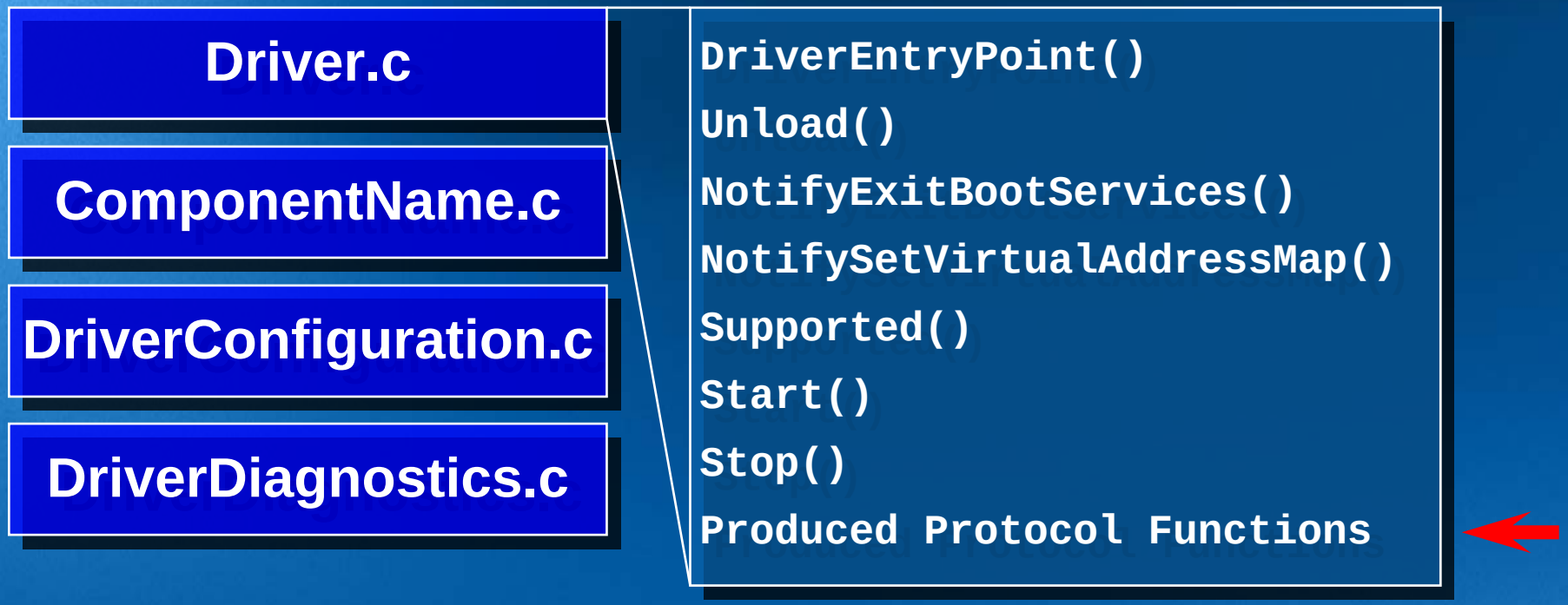
DriverConfiguration.c

DriverDiagnostics.c

RunDiagnostics()

```
Shell> DrvDiag –s DriverHandle DeviceHandle
Shell> DrvDiag –e DriverHandle DeviceHandle
Shell> DrvDiag –m DriverHandle DeviceHandle
```

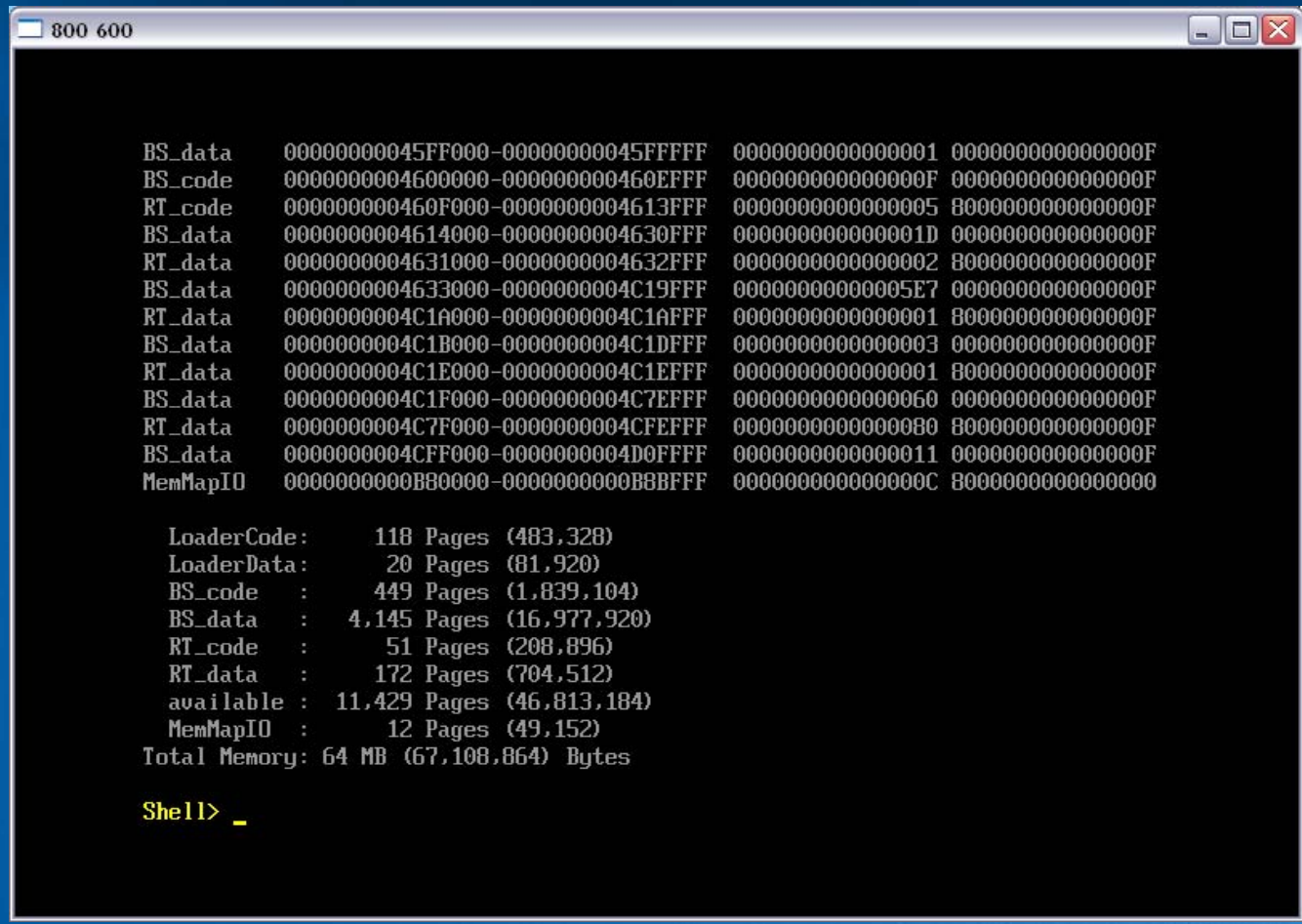
Testing Driver Functions



Depends on the protocol

Testing for leaks

- memmap
  - Displays a map of all memory in the system.
  - Can see what is allocated.



Checking For Load / Unload Leaks

- gBS->AllocatePool() / gBS->FreePool()
- gBS->AllocatePages() / gBS->FreePages()
- gBS->InstallMultipleProtocolInterfaces()
- gBS->UninstallMultipleProtocolInterfaces()

```
Shell> Memmap
Shell> Dh
Shell> Load DriverName.efi
Shell> Memmap
Shell> Dh
Shell> Unload DriverHandle
Shell> Memmap
Shell> Dh
```

Checking For Memory Leaks

- gBS->AllocatePool() / gBS->FreePool()
- gBS->AllocatePages() / gBS->FreePages()

```
Shell> Memmap
Shell> Connect DeviceHandle DriverHandle
Shell> Memmap
Shell> Disconnect DeviceHandle DriverHandle
Shell> Memmap
Shell> Reconnect DeviceHandle
Shell> Memmap
```

Checking For Handle Leaks

- gBS->InstallMultipleProtocolInterfaces()
- gBS->UninstallMultipleProtocolInterfaces()

```
Shell> dh
Shell> Connect DeviceHandle DriverHandle
Shell> dh
Shell> Disconnect DeviceHandle DriverHandle
Shell> dh
Shell> Reconnect DeviceHandle
Shell> dh
```

Checking For Protocol Leaks

- gBS->OpenProtocol() / gBS->CloseProtocol()

```
Shell> OpenInfo DeviceHandle
Shell> Connect DeviceHandle DriverHandle
Shell> OpenInfo DeviceHandle
Shell> Disconnect DeviceHandle DriverHandle
Shell> OpenInfo DeviceHandle
Shell> Reconnect DeviceHandle
Shell> OpenInfo DeviceHandle
```

Debugging code statements

- ASSERT()
- ASSERT_EFI_STATUS()
- CR()
- EFI_BREAKPOINT()
- DEBUG()
- Setting the error level

ASSERT()

- Standard ASSERT call.
- Evaluates the argument.
  - If false print filename and line number and halts

ASSERT_EFI_STATUS

- Same as before but evaluates argument against EFI_STATUS instead of true / false.
  - If argument is not EFI_SUCCESS prints filename and line number and halts.

CR()

- Used to verify data types
- `CR (Record, Type, Field, Signature)`
  - Calls ASSERT(FALSE) if Data Structure Signature does not match

EFI_BREAKPOINT

- Generates a CPU break point instruction
- Requires source level debugging capability

DEBUG

- Allows for printing out of strings depending on error level.
  - DEBUG(ErrorLevel, String, ...)
  - Prints string if error level is active.

Setting the error level in console

```
Shell> Err ErrorLevel
```

- some standard error levels:
  - EFI_D_ERROR 0x80000000
  - EFI_D_INIT 0x00000001
  - EFI_D_WARN 0x00000002
  - EFI_D_INFO 0x00000040
  - EFI_D_BLKIO 0x00001000
  - EFI_D_UNDI 0x00010000

COM cable usage

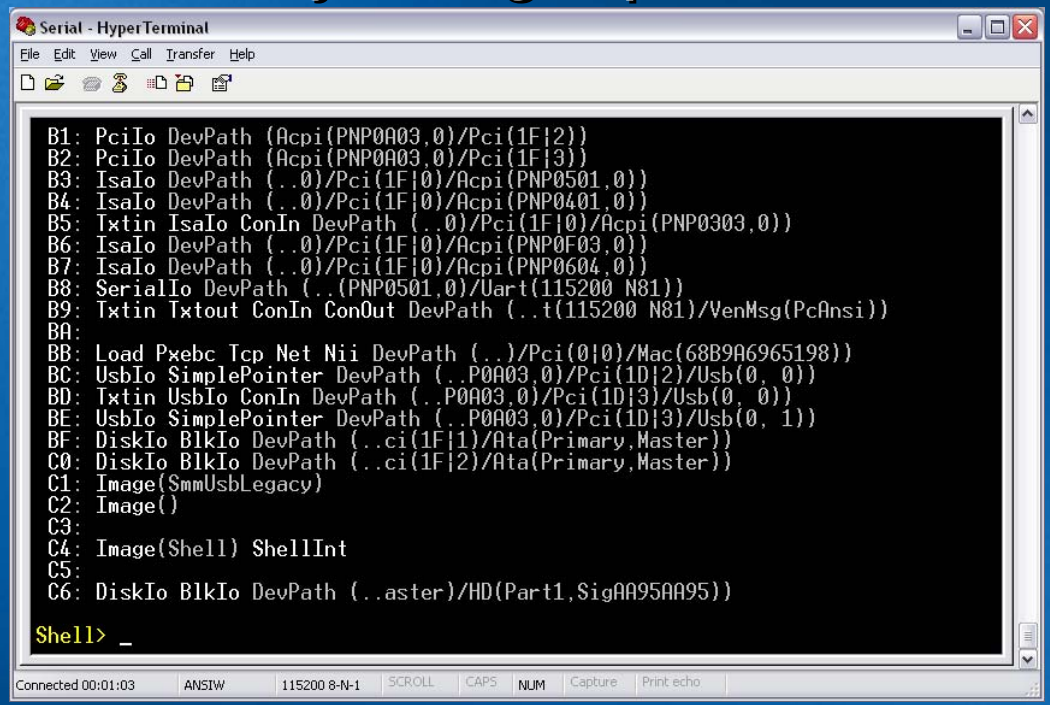
- Setup up
- Messages

COM setup

- Connect a null modem cable
- Open a connection with these settings:
  - Bits per second: 115200
  - Data bits: 8
  - Parity: None
  - Stop Bits: 1
  - Flow Control: None
- Boot the Framework / UEFI based BIOS

COM messages

- Can run DH or any other command from within serial connection
- Obviously no graphics stuff



American Arium usage

- American Arium makes a HW debugger that works with EFI.
  - Enables source level debugging
  - www.Arium.com

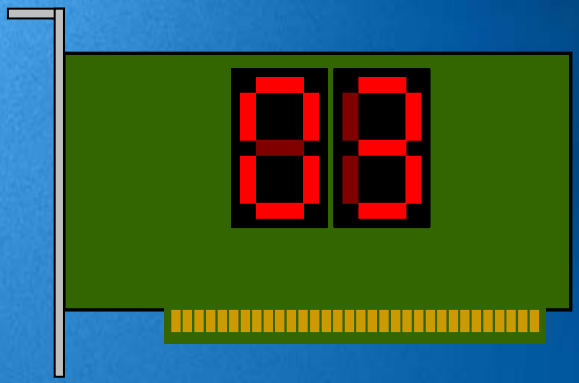
Testing methodologies

- How do I know if my driver works?
- Can I use my device?
- Can I boot off my device?
- Is my driver fully compliant?
- When am I done?

- UEFI IHV SCT


intel®
Leap ahead™

When DEBUG() is not Available



- POST Card (I/O 0x80)
  - PCI Root Bridge I/O Protocol
  - PCI I/O Protocol

```
Value = 0x03;
Status = PciIo->Io.Write (
                      PciIo,                          // This
                      EfiPciIoWidthUint8,             // Width
                      EFI_PCI_IO_PASS_THROUGH_BAR,    // BAR
                      0x80,                           // Offset
                      1,                              // Count
                      &Value                          // Buffer
                      );
```

May not work on all platforms
May produce unpredictable results
Must be removed from production drivers

When DEBUG() is not Available

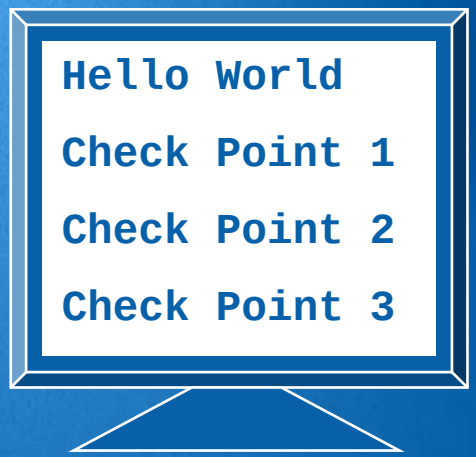
```
Hello World
Check Point 1
Check Point 2
Check Point 3
```

- UART (COM1 I/O 0x3F8-0x3FF)
- UART (Platform Specific MMIO)
  - –PCI Root Bridge I/O Protocol
  - –PCI I/O Protocol

```
Status = PciIo->PollIo (PciIo, EfiPciIoWidthUint8,
                        EFI_PCI_IO_PASS_THROUGH_BAR,
                        0x3FD, 0x20, 0x20, 1000000, &Lsr);
Status = PciIo->Io.Write (PciIo, EfiPciIoWidthUint8,
                          EFI_PCI_IO_PASS_THROUGH_BAR,
                          0x3F8, 1, &Data);
```

May not work on all platforms
May produce unpredictable results
Must be removed from production drivers

When DEBUG() is not Available



- VGA (MMIO 0xB8000-0xBFFFF)
  - –PCI Root Bridge I/O Protocol
  - –PCI I/O Protocol

```
VideoAddress   = 0xB8000 + (Row * 80 + Column) * 2;
VideoCharacter = 0x0700 | Character;
Status = PciIo->Mem.Write (PciIo, EfiPciIoWidthUint16,
                           EFI_PCI_IO_PASS_THROUGH_BAR,
                           VideoAddress, 1, &VideoCharacter);
```

May not work on all platforms
May produce unpredictable results
Must be removed from production drivers

Recommendations

- Test Functions with EFI Shell Commands
- Check for Leaks with EFI Shell Commands
- Install EFI Compliant Operating System
- Boot EFI Compliant Operating System
- Debug Macros Identify Critical Failures
- Use Same Techniques on all CPU Types
  - IA-32, Intel® 64, IA-64, EBC

Many Tools Available to Test and Debug

Testing EFI drivers

- Run drivers shell command
  - Verify driver attributes in database: handle, child controllers, device path generation, bus or device, config/diag protocol
  - Component name protocol should show name of driver, mfg, ver no. etc. to identify driver.
  - Version no. field should be set correctly for the driver

EFI Tests

- Should be done for EBC drivers and/or native drivers (on IA32, Intel® 64 and IA-64).
- Preferred drivers are EBC if they exist.
- Should test on each type of target platform (on IA32, Intel® 64 and IA-64) that the cards/drivers should work on.
- Run EFI IHV SCTs on EFI handles pertaining to the cards implementation (ie USB, Lan, SCSI etc.)
- Run manual tests below first before running EFI IHV SCTs.

Testing EFI drivers

- Do dh –d #handle on driver
  - –Verify child controllers managed
  - –Type of controller

Testing EFI drivers

- Do connect –r
- Verify with map command file systems/blkio produced by driver (or whatever driver stack the drivers should produce: usb – uhci, usbbus, usbkbd etc. Lan – Undi, snpxx etc.)
- Do disconnect on individual controllers
- Do reconnect –r (many times >100 with .nsh)
  - Disconnect /connect many times to look for resource leaks – memmap command or failures to connect. Available memory should not decrease constantly as each disconnect is run(it ok for avg to be constant but not to increase over many reconnects).
- # handles should not increase everytime reconnect occurs.

Testing EFI storage drivers

- Verify EFI file systems are usable.
  - Copy large files ranging from min block size to max lba blocks on media.
  - Verify copied files on known good system other than EFI (use windows or linux to do verify)
  - Repeat copying many times and verify with original images on known machine
  - Check first and last block (gpt) on partition
  - Do Dblk on blkio driver to check for correctness of data with known files (and see if driver works)

Testing EFI drivers

- Test DRVCFG(setup config) options on driver
  - Verify nvram settings in driver
- Test DRVDIAG options on driver
  - Verify diagnostic tests

Testing EFI drivers

- Run with console redirection on (through serial port)
- Run Drvcfg with console redirection
- Run Drvdiag with console redirection

- Remote head must be able to see/interact and record all console interactions with driver, drvcfg and drvdiag utilities.

Testing EFI drivers

- Lan drivers (non –storage)
  - Test booting to PXE server
  - Transfer large image (entire disk images)
    - Using TCP/IP stack in toolkit

Video UGA drivers

- Load UGA driver for display card
- Exercise setmode for different resolutions
- Run loadbmp and exercise blt to screen with various size pictures. Use known bmp so that all the buffer is used and pixels are verified
- Use shell mode command to change resolutions
- Cls x (0..9) verify different colors.
- Disconnect and reconnect UGA handles with console
- Verify with EFI OS (booting) that OS goes to desired setmode during OS loader.


intel®
Leap ahead™