Debugging Unified Extensible Firmware Interface (UEFI) Firmware under Linux*

Laurie Jarlstrom laurie.Jarlstrom@intel.com,

Intel Corporation

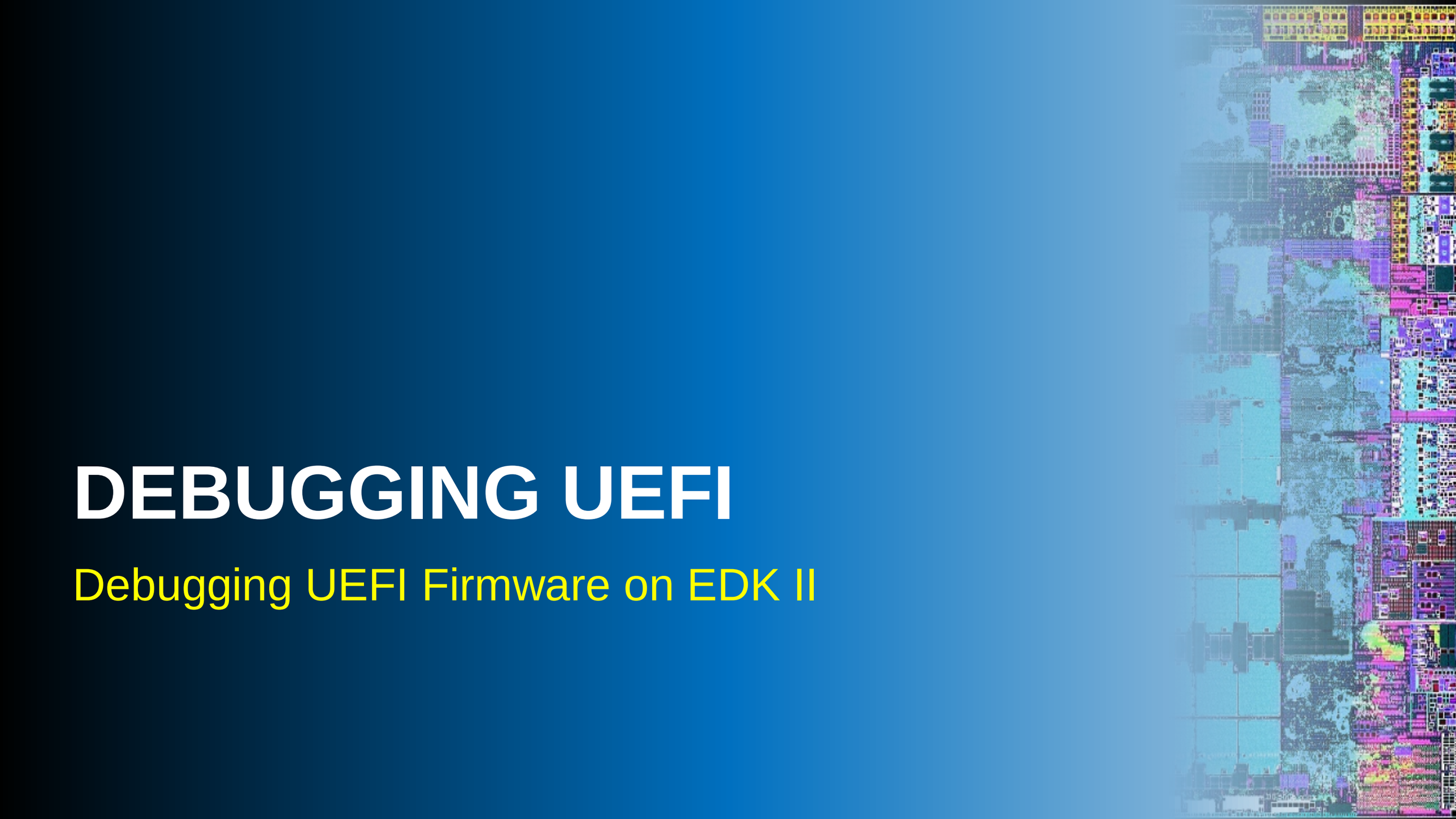
OSFC - 2018

WORKSHOP OBJECTIVE

- List the ways to debug
- Define EDK II (EFI Development Kit) **`DebugLib`** and its attributes
- Introduce the Intel® UEFI Development Kit Debugger (Intel® UDK Debugger)
- Debugging PI's phases
- Debug EDK II using Intel® UDK w/ GDB - LAB

DEBUGGING UEFI

Debugging UEFI Firmware on EDK II



Debug Methods

DEBUG and ASSERT macros in EDK II code

DEBUG instead of Print functions

Software/hardware debuggers

Shell commands to test capabilities for simple debugging

EDK II `DebugLib` Library

`Debug` and **`Assert`** macros in code

Enable/disable when compiled (**`target.txt`**)

Connects a Host to capture debug messages

Using PCDs to Configure DebugLib

MdePkg Debug Library Class

```
[PcdsFixedAtBuild. PcdsPatchableInModule]
    . . .
  gEfiMdePkgTokenSpaceGuid.PcdDebugPropertyMask|0x1f
  gEfiMdePkgTokenSpaceGuid.PcdDebugPrintErrorLevel|0x80000040
```

The DebugLib Class

Interface

MdePkg\Include\Library\DebugLib.h

Macros

(where PCDs are checked)

```
ASSERT (Expression)
DEBUG (Expression)
ASSERT_EFI_ERROR (StatusParameter)
ASSERT_PROTOCOL_ALREADY_INSTALLED(…)
```

Advanced Macros

```
DEBUG_CODE (Expression)
DEBUG_CODE_BEGIN() & DEBUG_CODE_END()
DEBUG_CLEAR_MEMORY(…)
```

Implementation

DebugLib Instances

BaseDebugLibSerialPort	(1)
UefiDebugLibConOut	(2)
UefiDebugLibStdErr	(3)
PeiDxeDebugLibReportStatusCode	(4) *

**Default for most platforms

Changing Library Instances

Change common library instances in the platform DSC by module type

```
[LibraryClasses.common.IA32]
DebugLib|MdePkg/Library/BaseDebugLibNull/BaseDebugLibNull.inf
```

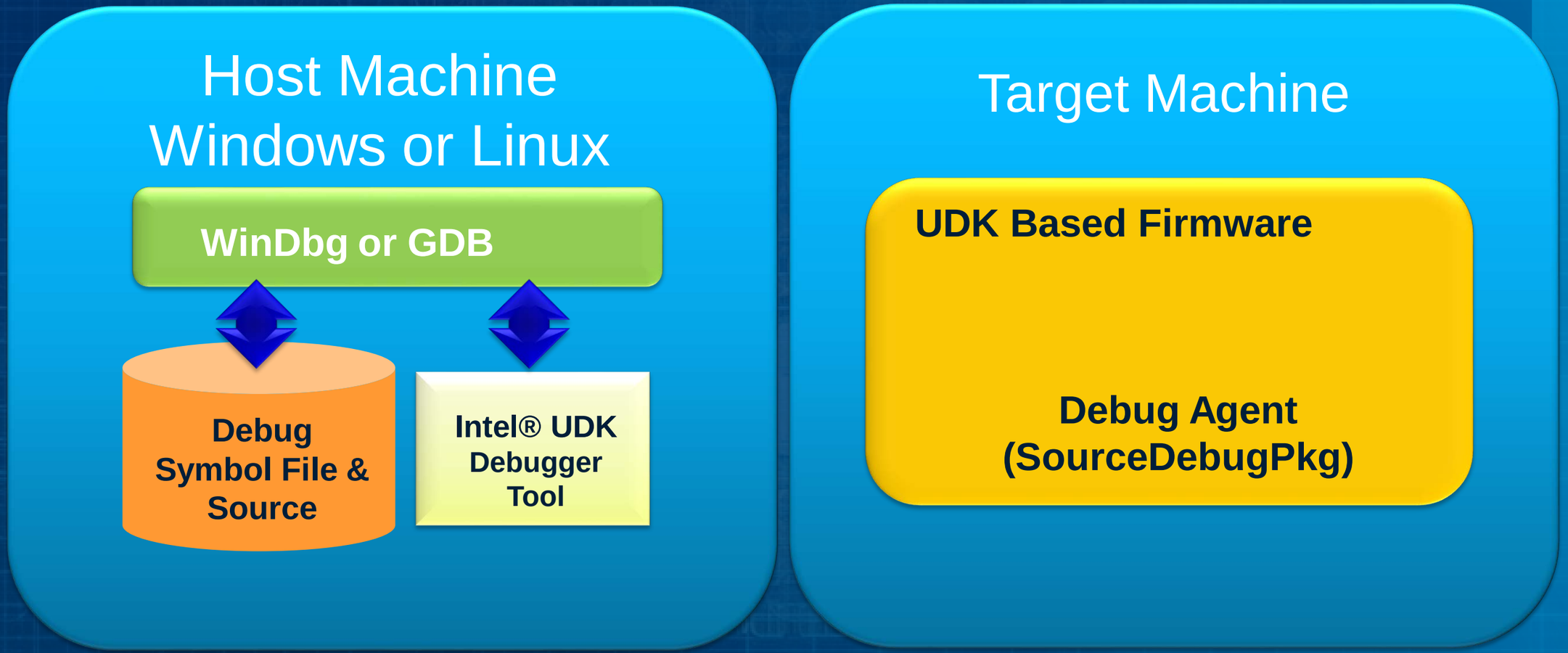
Change a single module's library instance in the platform DSC

```
MyPath/MyModule.inf {
<LibraryClasses>
DebugLib|MdePkg/Library/BaseDebugLibSerialPort.inf
}
```

UEFI DEBUGGER OVERVIEW

Intel® UEFI Development Kit Debugger Tool

Intel® UEFI Development Kit Debugger Tool



Source Level Debugger for UEFI

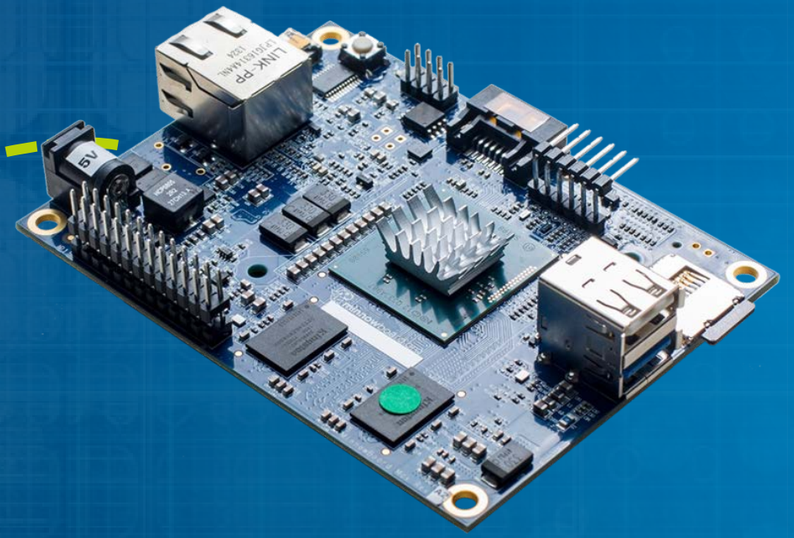
Host & Target Debug Setup

Host

Null Modem Cable or

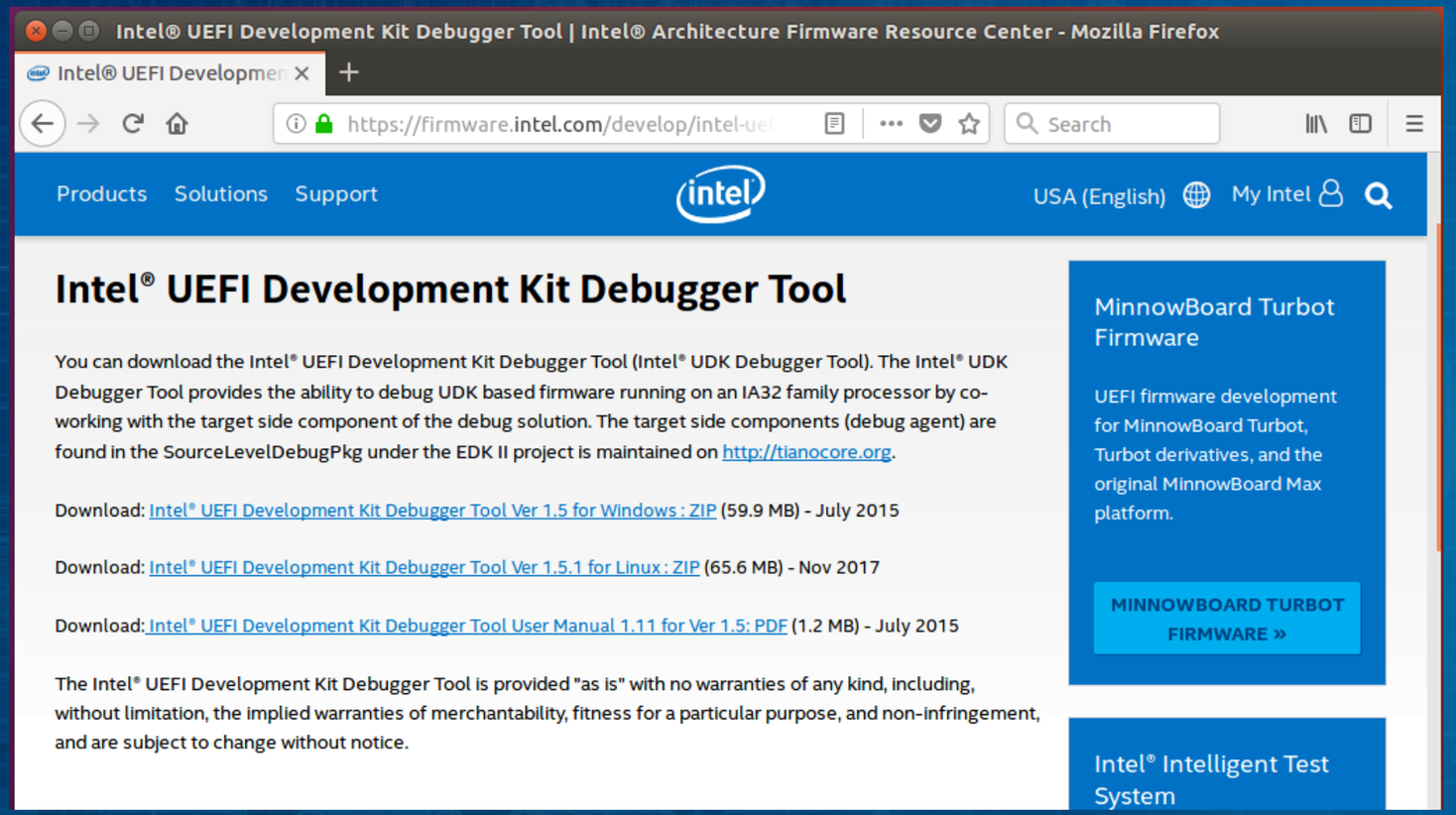
USB 2.0 Debug Cable or USB 3.0

Target



Distribution

Download application: http:/firmware.intel.com - Develop - Tools



Target source: `SourceLevelDebugPkg` at TianoCore.org

Host Configuration Requirements

Microsoft Windows

- XP with Service Pack 3 and Windows 7 and Windows 10
- Debug Tool (WINDBG) x86, version 6.11.0001.404
- Intel UDK Debugger Tool
- WinDBG Extensions in **edk2.dll**

Host Configuration Requirements

Linux

- Ubuntu 16.04 LTS client (x64 build)- validated and examples shown
- GNU Debugger (GDB) - with Expat library
- Intel UDK Debugger Tool 1.5.1

Host Configuration Requirements-GDB

Check for the configuration of GDB that is installed

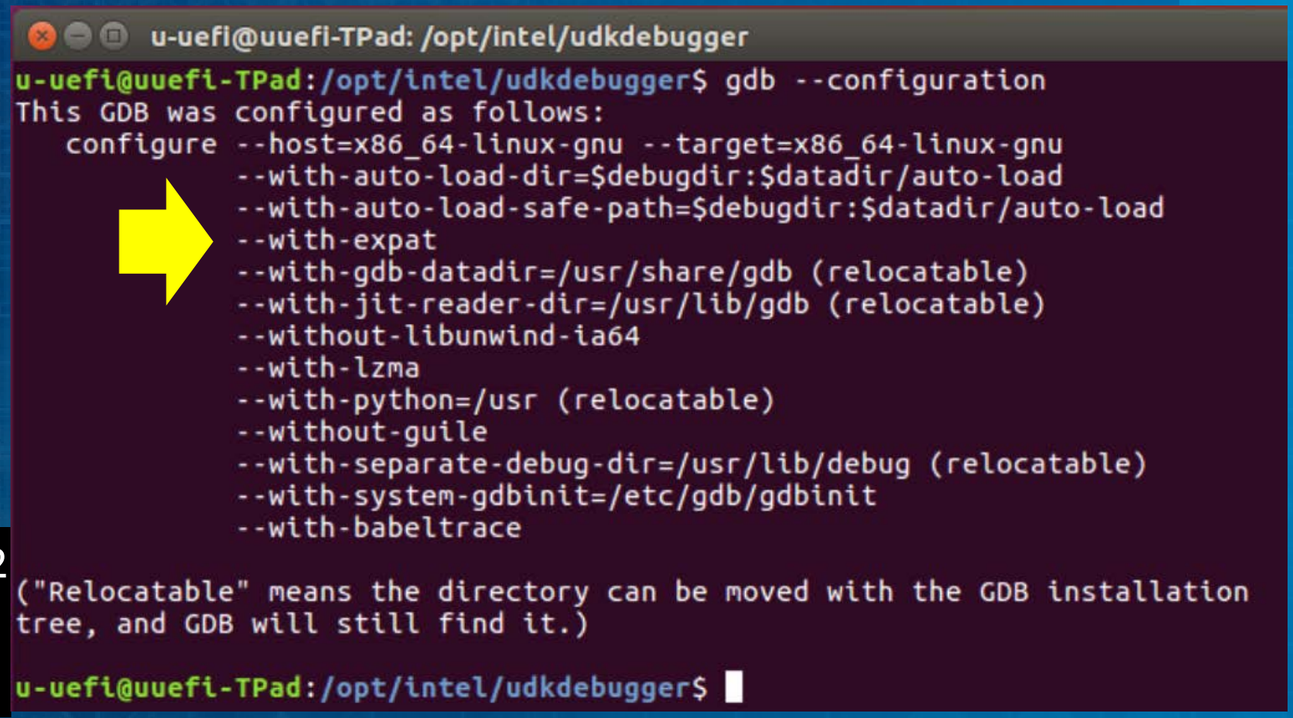
```
bash$ gdb –configuration
```

Install gdb if not installed

```
bash$ sudo apt-get update
bash$ sudo apt-get install gdb
```

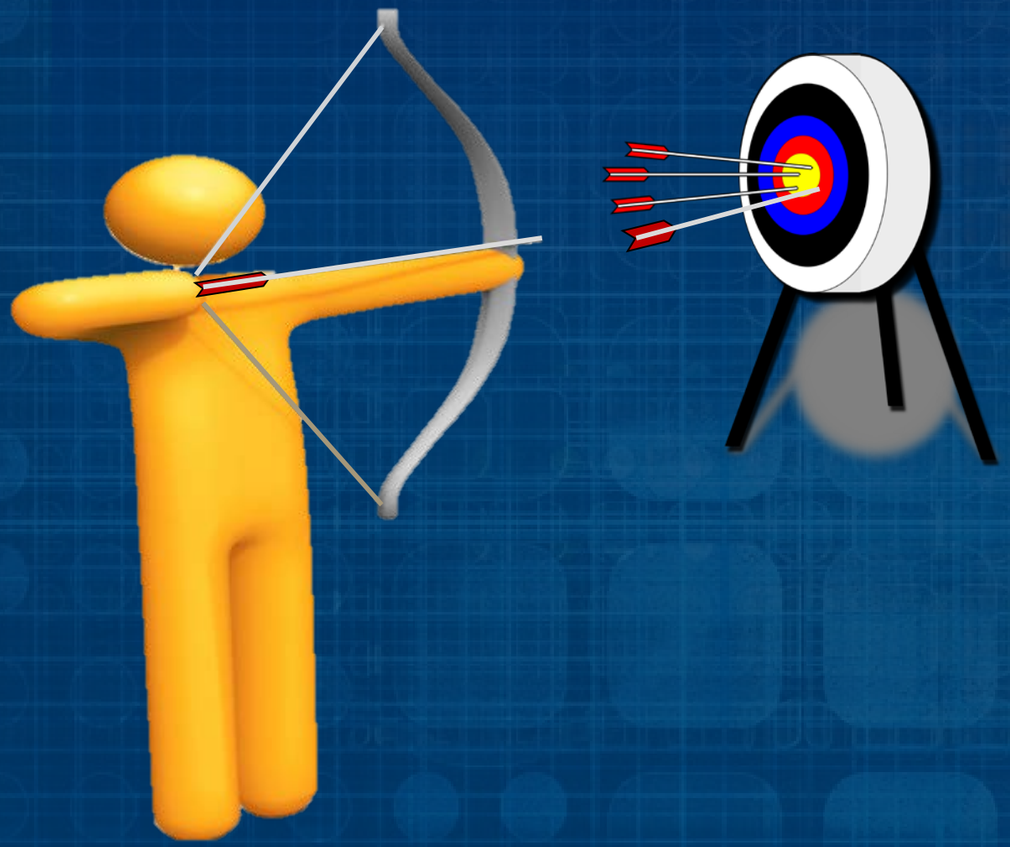
Download gdb source and compile with Expat library if there is **no** "--with-expat" as on the screen shot here

```
bash$ ./configure --target=x86_64-w64-mingw32
--with-expat
bash$ make
```



Changes to Target Firmware

Goal: Minimize changes to target firmware

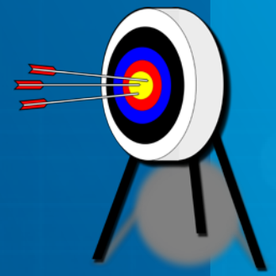


Add call to new library class (**`DebugAgentLib`**) In SEC, DXE Main, and SMM CPU Mod.

Or if you don’t want to add one

A **NULL** implementation of **`DebugAgentLib`** is checked into open source

Configure (Target)



- Add Symbolic Debug to platform DSC (Build Switch)

  `-D SOURCE_DEBUG_ENABLE`
- Change Debug Agent Library appropriately (SEC | DXE | SMM)
- Configure target to use COM port via PCD
- Ensure COM port not used by other project modules/features
- Simple “ASCII Print” though COM port is allowed

DEBUGGING UEFI

Debugging UEFI Firmware using Intel® UDK w/ GDB

Source Level Debug Features

View call stack

Go

Insert **CpuBreakpoint**

View and edit local/global variables

Set breakpoint

Step into/over routines

Go till

View disassembled code

View/edit general purpose register values

Launching UDK Debugger- Linux

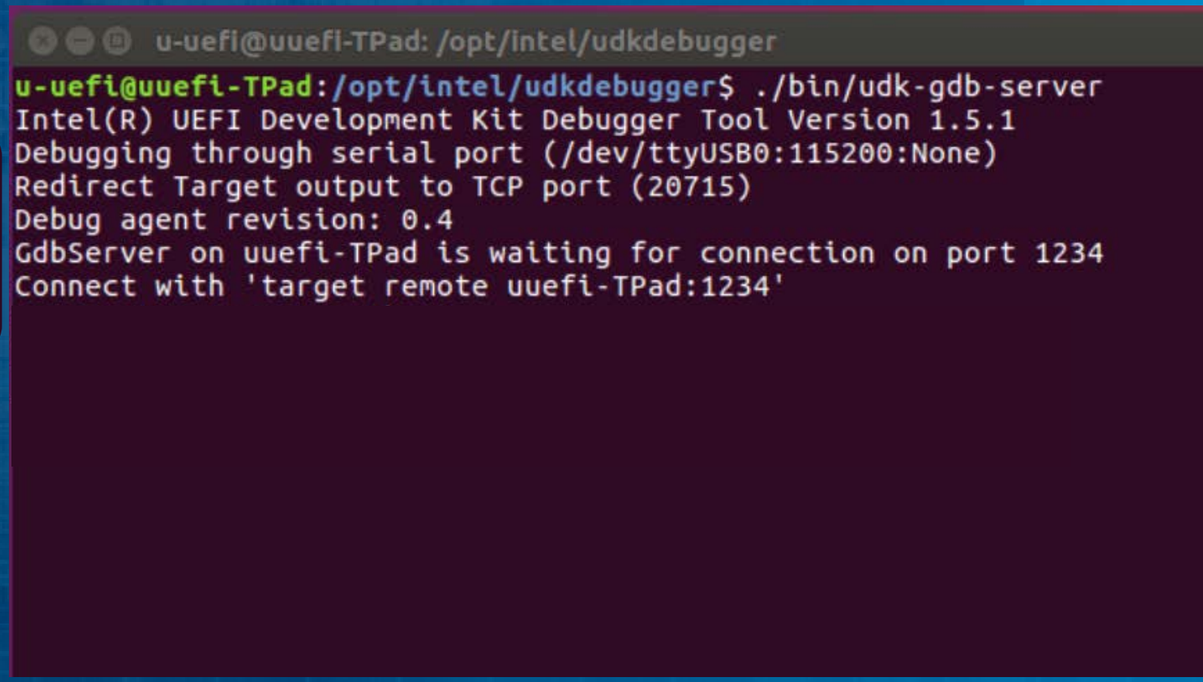
Example showing Ubuntu 16.04 LTS with GDB

Need to open 3 Terminal windows
First Terminal(1) is the UDK debugger server

```
bash$ cd opt/intel/udkdebugger
bash$ ./bin/udk-gdb-server
```

Power on the Target and wait 2-3 seconds

Terminal (1)



Launching UDK Debugger- Linux

Example showing Ubuntu 16.04 LTS with GDB

Open a second Terminal(2) for GDB

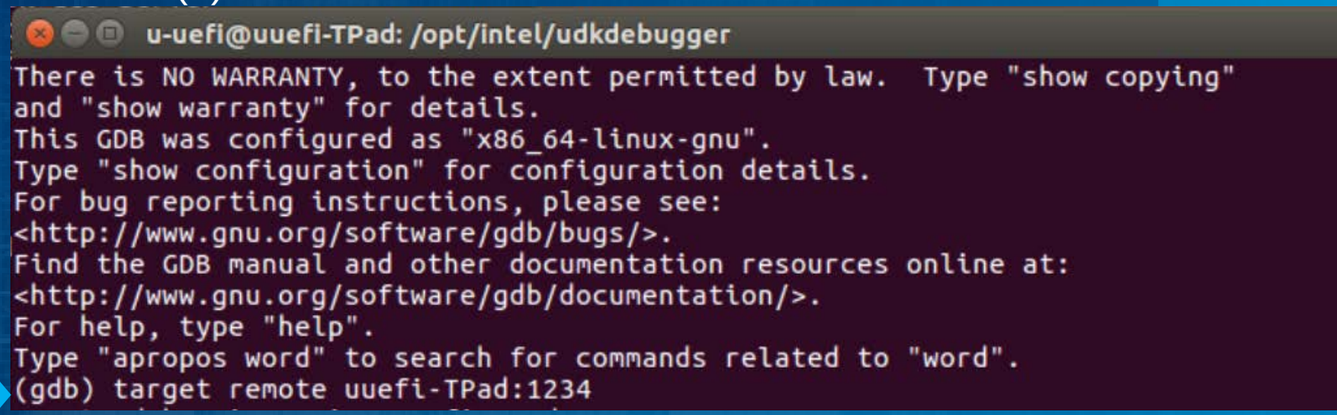
```
bash$ cd opt/intel/udkdebugger
bash$ gdb
```

Attach to the UDK debugger

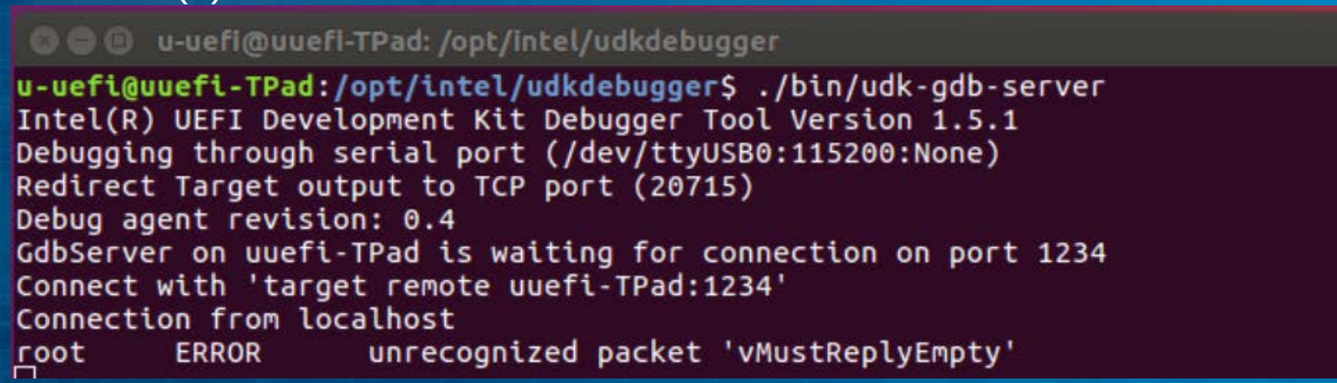
```
(gdb) target remote <HOST>:1234
```

Terminal(1) will show "Connection from localhost" message

Terminal (2)



Terminal (1)



Launching UDK Debugger- Linux

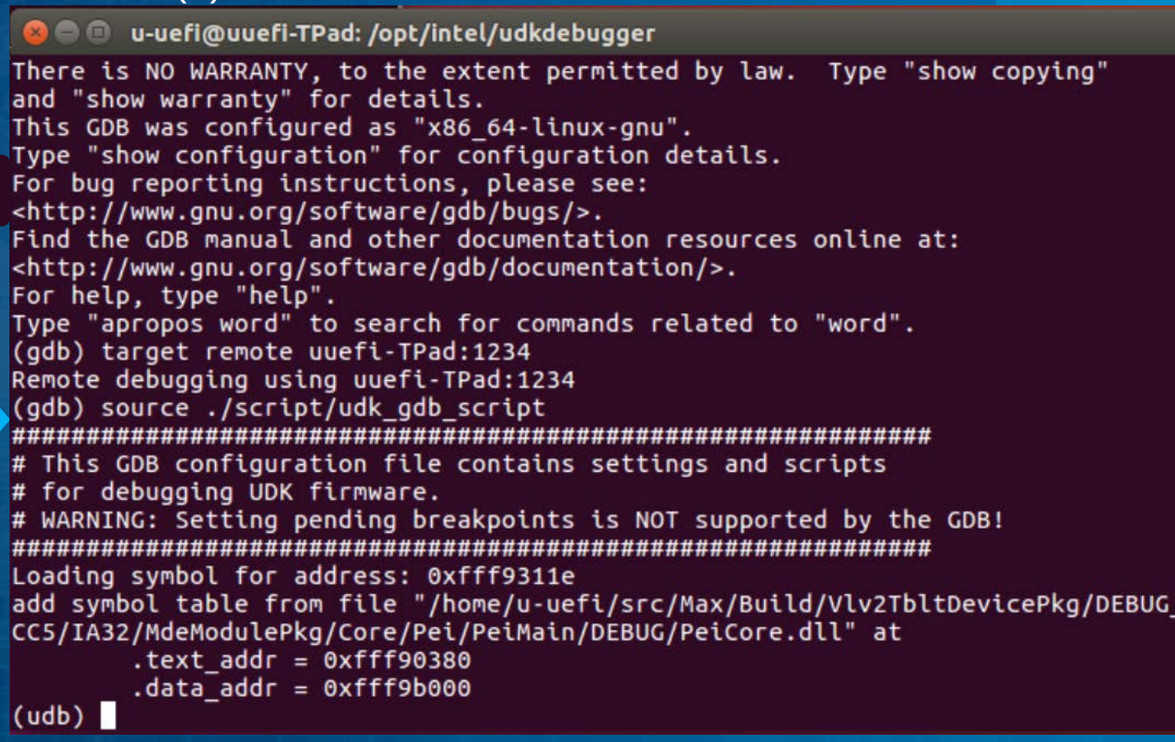
Example showing Ubuntu 16.04 LTS with GDB

Open the udk scripts in GDB – Terminal(2)

```
(gdb) source ./script/udk_gdb_script
```

The prompt changes from "(gdb)" to "(udb)"

Terminal (2)

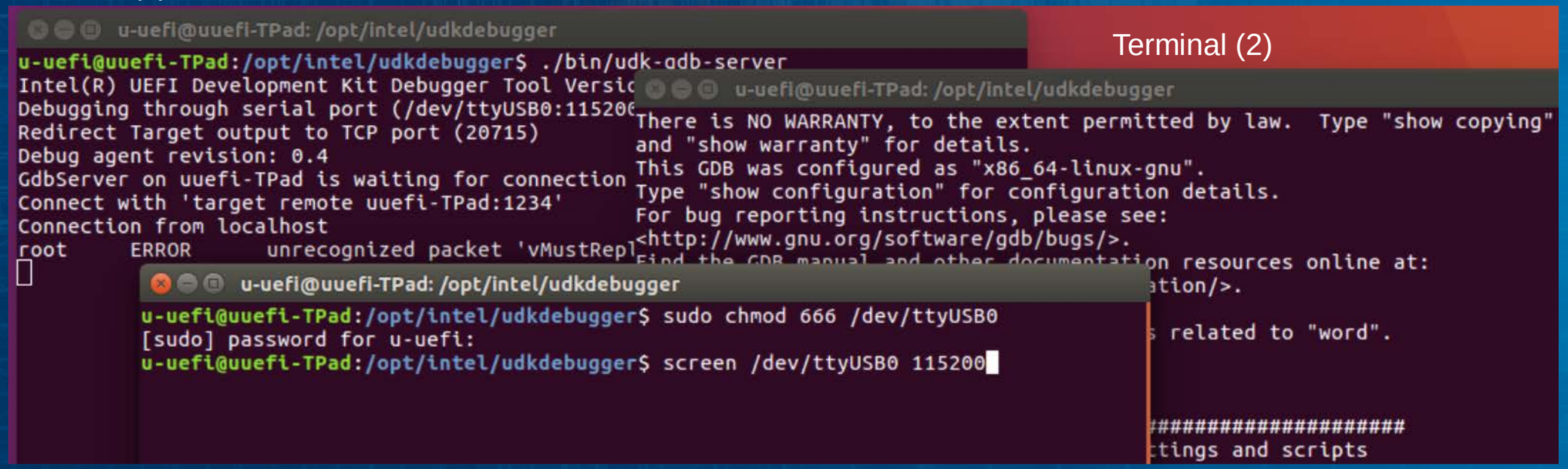


Launching UDK Debugger- Linux

Optional - open a 3rd Terminal(3)

- Example showing "screen" terminal program with “real” hardware
- Or cat debug.log with QEMU

Terminal (1)



Terminal (3)

UDK Debugger – Setting break points

Example showing Ubuntu 16.04 LTS with GDB

Terminal(2) Breakpoint at PeiDispatcher

```
(udb) b PeiDispatcher
```

Break at Port 0x80

```
(udb)iowatch/b 0x80
```

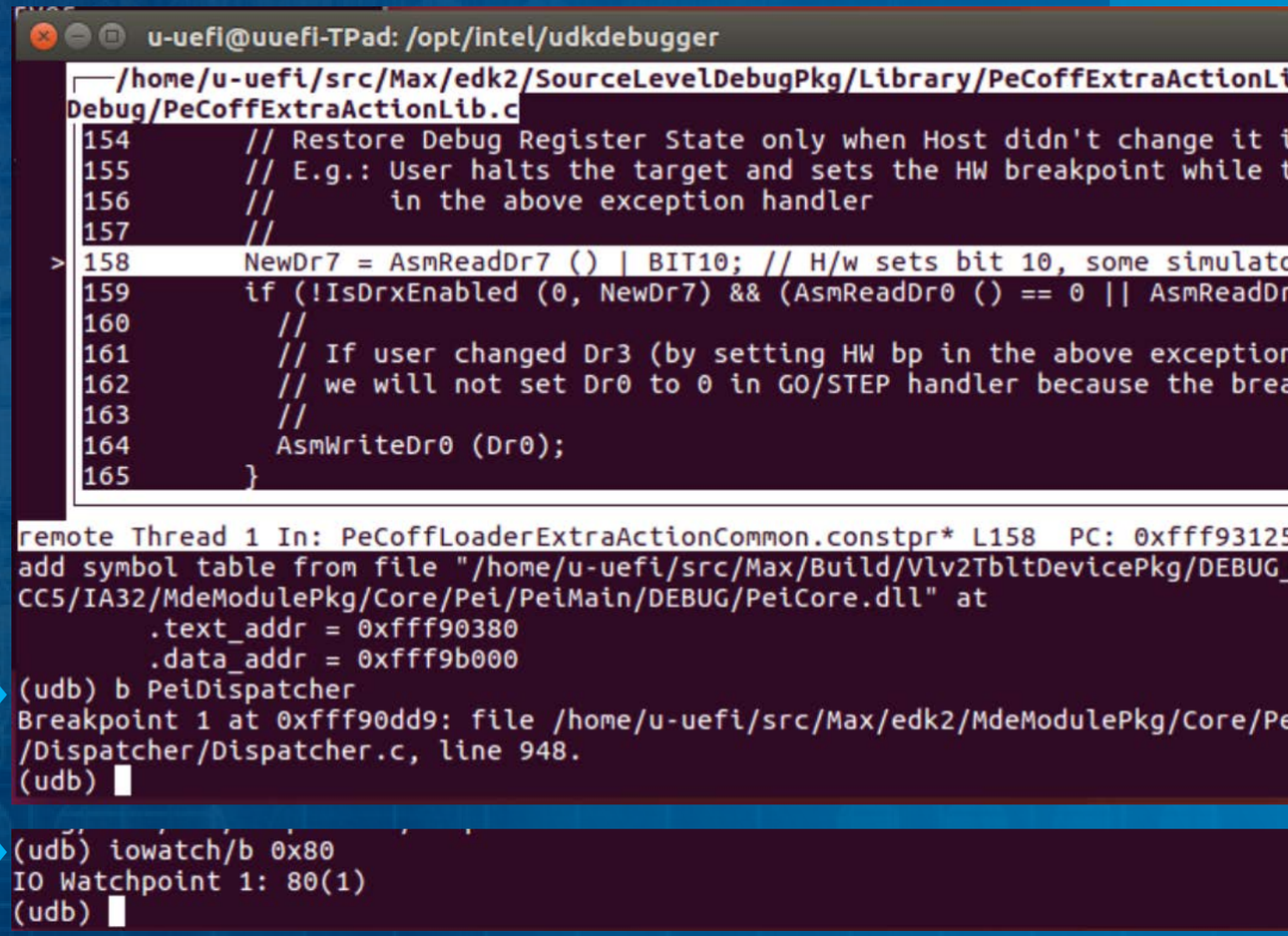
Break at absolute address

```
(udb)b *0xfff94a68
```

Break in a line of code

```
(udb)b myapp.c:78
```

Terminal (2)



UDK Debugger UEFI Scripts

Info modules

Lists information about the loaded modules or the specified module

py mmio

Access the memory mapped IO space

py pci

Display PCI devic list

py mtrr

Dump the MTRR Setting of the current processor

py DumpHobs

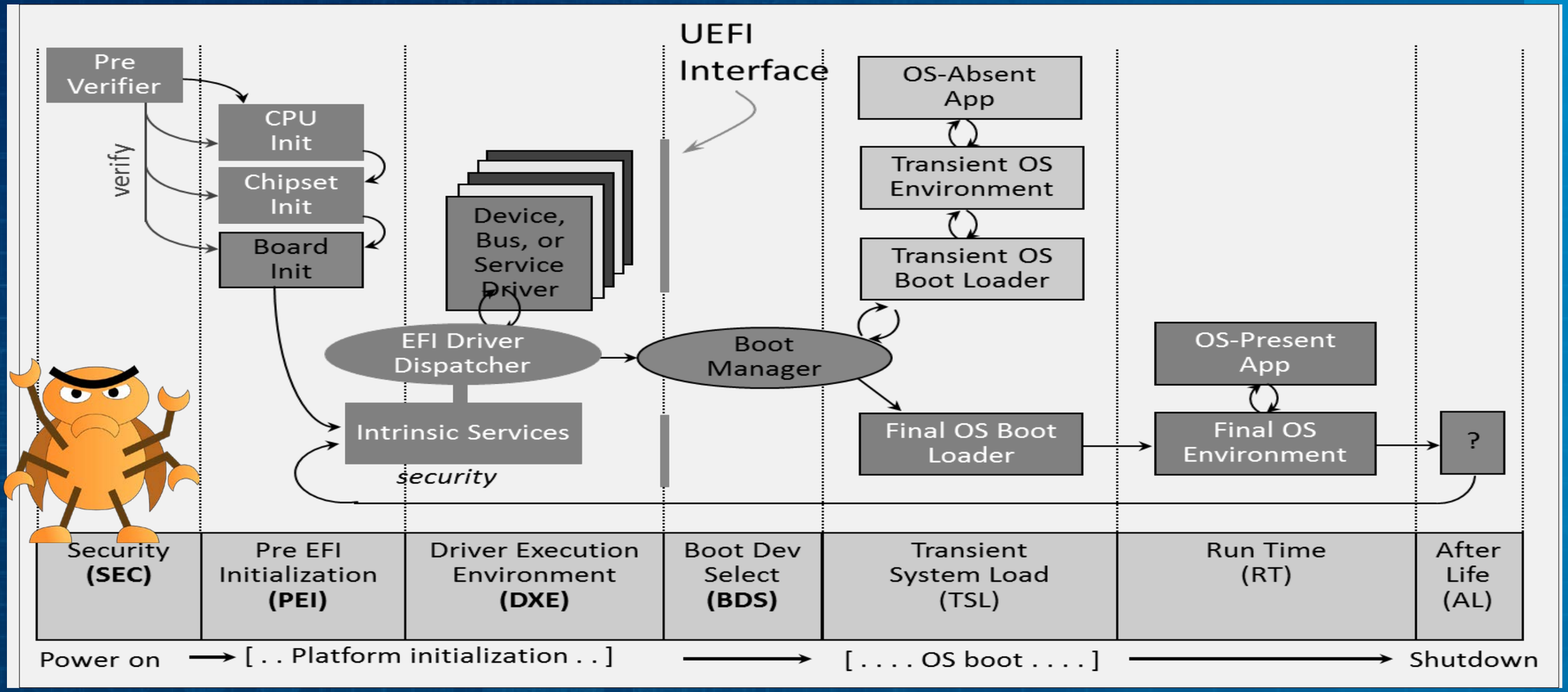
dump contents of the HOB list

resettarget

DEBUGGING BOOT FLOW

Debug through the UEFI Firmware Boot Flow

Debugging the Boot Phases

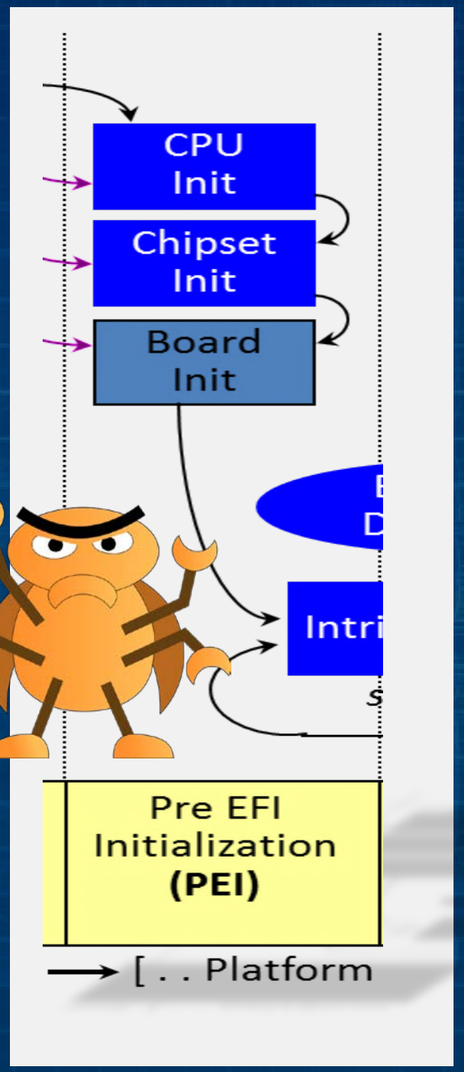


Debugging the Boot Phases - SEC

Debugging Sec Phase

SORRY – Requires a hardware debugger

Debugging the Boot Phases - PEI



- Use debugger prior to PEI Main
- Check proper execution of PEI drivers
- Execute basic chipset & Memory init.
- Check memory availability
- Complete flash accessibility
- Execute recovery driver
- Detect DXE IPL

PEI Phase: Trace Each PEIM

There is a loop function in :

MdeModulePkg/Core/Pei/Dispatcher/Dispatcher.c

Add CpuBreakpoint(); before launching each PEIM

```
VOID
PeiDispatcher (
  IN CONST EFI_SEC_PEI_HAND_OFF  *SecCoreData,
  IN PEI_CORE_INSTANCE           *Private
  )
{ // ...
        // Call the PEIM entry point
        //
        PeimEntryPoint = (EFI_PEIM_ENTRY_POINT2)(UINTN)EntryPoint;
        PERF_START (PeimFileHandle, "PEIM", NULL, 0);
// Add a call to CpuBreakpoint();  approx. line 1004
        CpuBreakpoint();
        PeimEntryPoint(PeimFileHandle, (const EFI_PEI_SERVICES **) &Private->Ps);
```

Check for transition from PEI to DXE

Critical point before calling DXE in:

MdeModulePkg/Core/Pei/PeiMain.c

Add CpuBreakpoint(); before entering DxeIpl

```
VOID
EFIAPI
PeiCore (
  IN CONST EFI_SEC_PEI_HAND_OFF        *SecCoreDataPtr,
  IN CONST EFI_PEI_PPI_DESCRIPTOR      *PpiList,
  IN VOID                              *Data
  )
{ // ...
  // Enter DxeIpl to load Dxe core.
  //
  DEBUG ((EFI_D_INFO, "DXE IPL Entry\n"));
// Add a call to CpuBreakpoint();  approx. line 468
  CpuBreakpoint();
  Status = TempPtr.DxeIpl->Entry (
                             TempPtr.DxeIpl,
                             &PrivateData.Ps,
                             PrivateData.HobList
```

Check for transition from DxeIpl to DXE

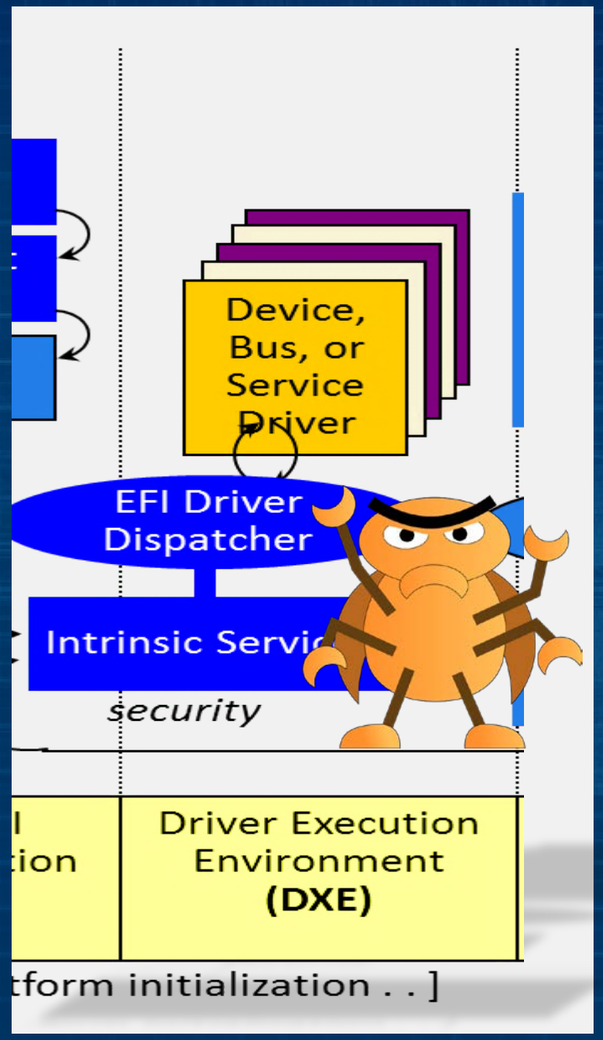
Critical point before calling DXE Core in:

MdeModulePkg/Core/DxeIplPeim/DxeLoad.c

Before entering Dxe Core (Notice also this is a standalone module - DxeIpl.efi)

```
EFI_STATUS
EFIAPI
DxeLoadCore (
  IN CONST EFI_DXE_IPL_PPI *This,
  IN EFI_PEI_SERVICES      **PeiServices,
  IN EFI_PEI_HOB_POINTERS  HobList
  )
{ // ...
  // Transfer control to the DXE Core
  // The hand off state is simply a pointer to the HOB list
  //
// Add a call to CpuBreakpoint();  approx. line 790
  CpuBreakpoint();
  HandOffToDxeCore (DxeCoreEntryPoint, HobList);
  //
  // If we get here, then the DXE Core returned.  This is an error
```

Debugging the Boot Phases - DXE



- Search for cyclic dependency check
- Trace ASSERTs caused during DXE execution
- Debug individual DXE drivers
- Check for architectural protocol failure
- Ensure BDS entry call

DXE: Trace Each Driver Load

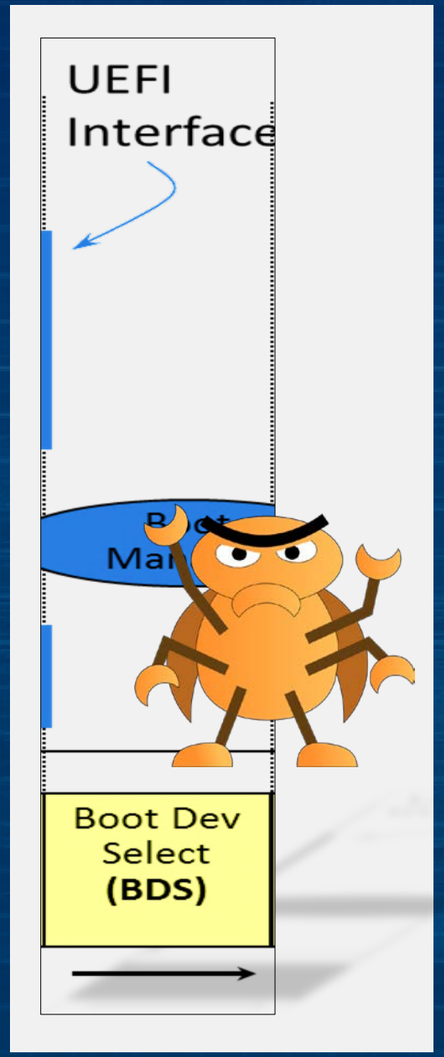
DXE Dispatcher calls to each driver's entry point in:

MdeModulePkg/Core/Dxe/Image/Image.c

Break every time a DXE driver is loaded.

```
EFI_STATUS
EFIAPI
CoreStartImage (
  IN EFI_HANDLE  ImageHandle,
  OUT UINTN      *ExitDataSize,
  OUT CHAR16     **ExitData  OPTIONAL
  )
{ // ...
    //
    // Call the image's entry point
    //
    Image->Started = TRUE;
// Add a call to CpuBreakpoint();  approx. line 1673
    CpuBreakpoint();
    Image->Status = Image->EntryPoint (ImageHandle, Image->Info.SystemTable);
```

Debugging the Boot Phases - BDS



- Detect console devices (input and output)
- Check enumeration of all devices' preset
- Detect boot policy
- Ensure BIOS "front page" is loaded

BDS Phase – Entry Point

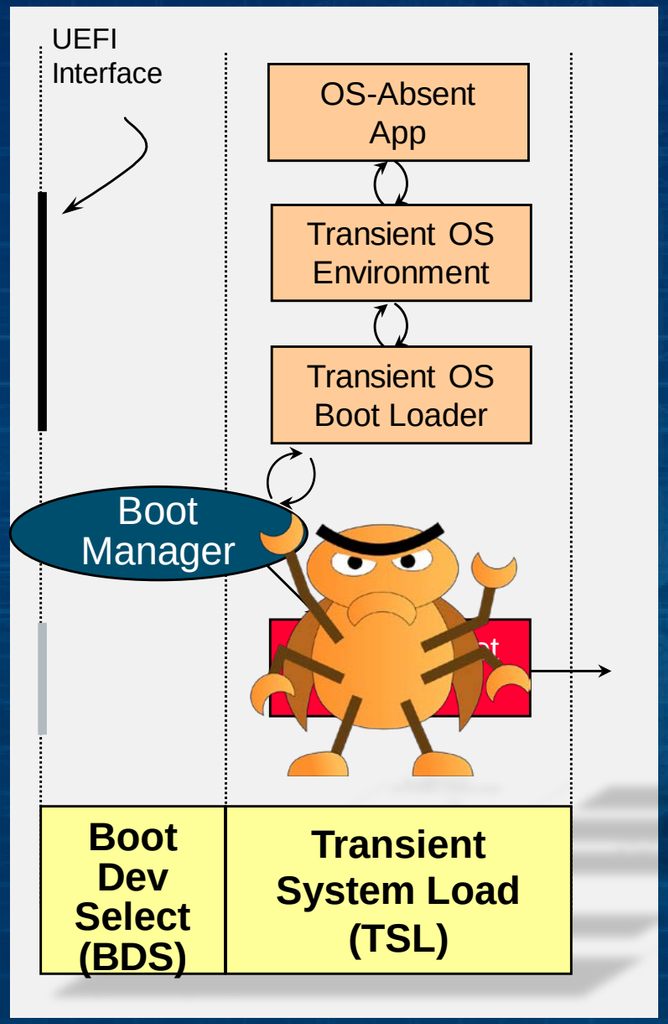
DXE call to BDS entry point in:

MdeModulePkg/Core/Dxe/DxeMain/DxeMain.c

Add CpuBreakpoint(); to break before BDS.

```
VOID
EFIAPI
DxeMain (
  IN  VOID *HobStart
  )
{ // ...
  // Transfer control to the BDS Architectural Protocol
  //
// Add a call to CpuBreakpoint();  approx. line 554
  CpuBreakpoint();
  gBds->Entry (gBds);

  //
  // BDS should never return
  //
  ASSERT (FALSE);
  CpuDeadLoop ();
```

Debugging the Boot Phases - Pre-Boot



- "C" source debugging
- UEFI Drivers
  - Init
  - Start
  - Supported
- UEFI Shell Applications
  - Entry point
  - Local variables
- **`CpuBreakpoint()`**

Debug in Pre-Boot – UEFI Shell Application

Add CpuBreakpoint() to SampleApp.c near the entry point

Add SampleApp.inf to the platform .dsc file

```
bash$ cd <edk2 workspace directory>
bash$ . edksetup.sh
bash$ build -m SampleApp/SampleApp.inf
```

Copy the binary SampleApp.efi to USB drive

SampleApp.c(~/Max/edk2/SampleApp) - gedit

Save

```
EFI_STATUS
EFIAPI
UefiMain (
  IN EFI_HANDLE        ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable
  )
{
        UINTN              EventIndex;
        BOOLEAN            ExitLoop;
        EFI_INPUT_KEY      Key;
        DEBUG((0xffffffff, "\n\nUEFI Base Training DEBUG DEMO\n"));
        DEBUG((0xffffffff, "0xffffffff USING DEBUG ALL Mask Bits Se

        CpuBreakpoint();
```

Debug in Pre-Boot – UEFI Shell Application

Use UDK Debugger and GDB to debug SampleApp

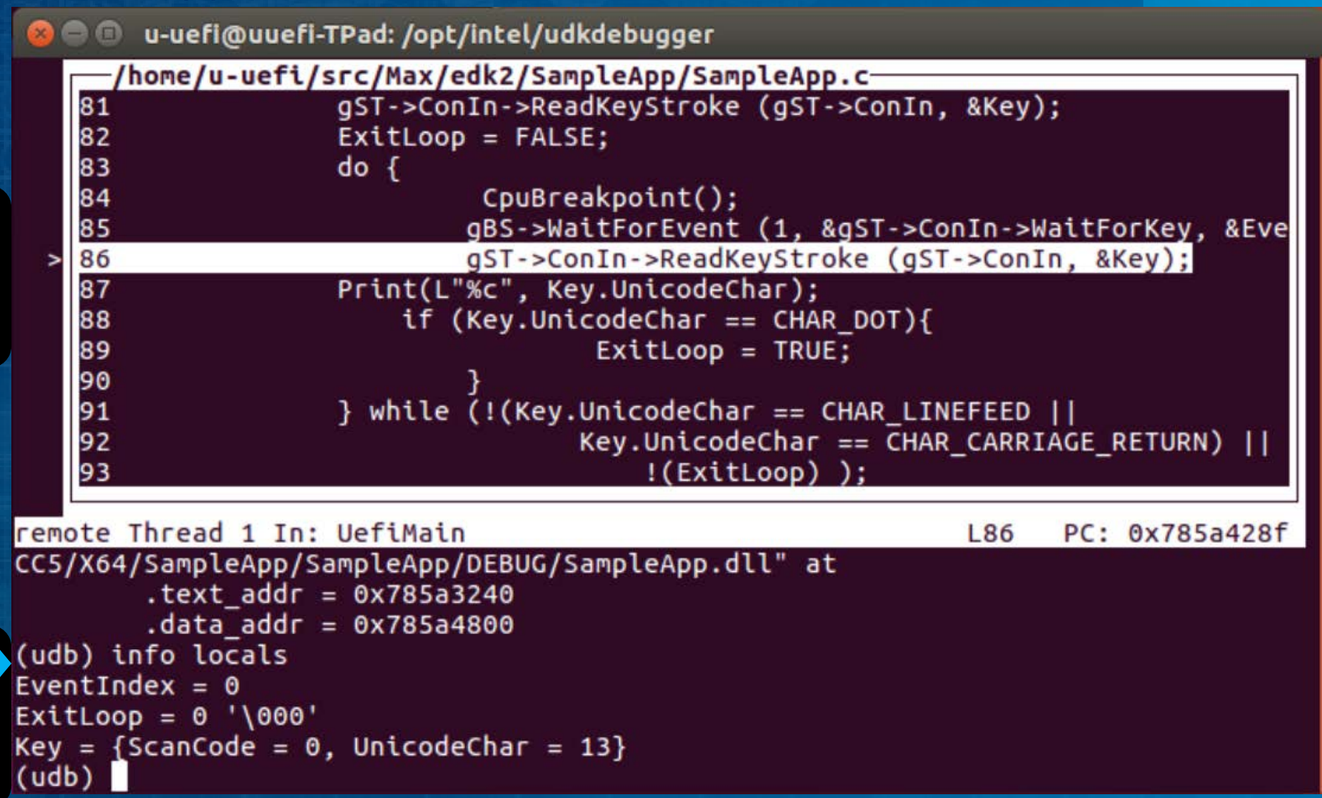
At the UEFI shell prompt on the target invoke SampleApp

```
Shell> Fs0:
FS0:/> SampleApp
```

GDB will break at the CpuBreakpoint
Begin debugging SampleApp

```
(udb) layout src
(udb) info locals
(udb) next
```

Terminal (2)



DEBUG WORKSHOP

Steps to setup the GDB & UDK with QEMU to debug UEFI Firmware

SETUP OVMFPKG

Setup OvmfPkg to build and run with QEMU and Ubuntu

EXTRACT CONTAINER

Copy “docker“ directory from usb thumb drive to local hard driver

Change permissions for Docker

```
sudo chmod a+rwx /var/run/docker.sock
sudo chmod a+rwx /var/run/docker.pid
```

Load the edk2-Ubuntu.docker

```
bash$ docker load –i edk2-Ubuntu.docker
```

For OpenSuse – also set up share

```
bash$ xhost local:root
```

Create a work directory

```
bash$ mkdir workspace
bash$ cd workspace
```

Docker run command from the docker directory

```
bash$ . ~/docker/edk2-ubuntu.sh
```

EXTRACT CONTAINER

From this point on the terminal window for Ubuntu 16.04 will be available and $HOME/workspace will be the shared directory between the host and the Docker container
window will look like:

```
[edk2-ubuntu] workspace #
```

Open Multiple terminal windows run the script OpenEdk2Window.sh

```
bash$ docker exec –it edk2 bash
```

WORKSHOP MATERIAL

1. Copy the Workshop_Material from the thumb drive to $HOME/workspace

Directory Workshop_Material/ will be created

```
~/workspace/Workshop_Material/
 - Documentation
 - edk2
 - SampleCode
 - Presentations
```

Create QEMU Run Script

1. Create Directories for invoking Qemu under the home directory

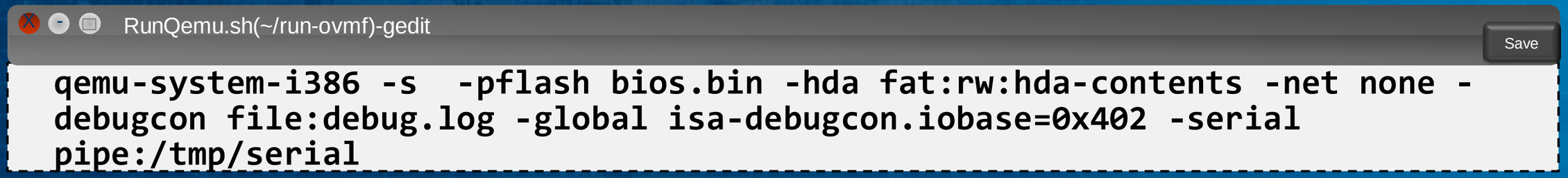
```
bash$ cd ~/workspace
bash$ mkdir run-ovmf
bash$ cd run-ovmf
bash$ mkdir hda-contents
```

2. Create a FIFO Pipe used by Qemu and the UDK debugger

```
bash$ mkfifo /tmp/serial.in
bash$ mkfifo /tmp/serial.in
```

3. Create a Linux shell script to run the QEMU from the run-ovmf directory

```
bash$ gedit RunQemu.sh
```



4. Save and Exit

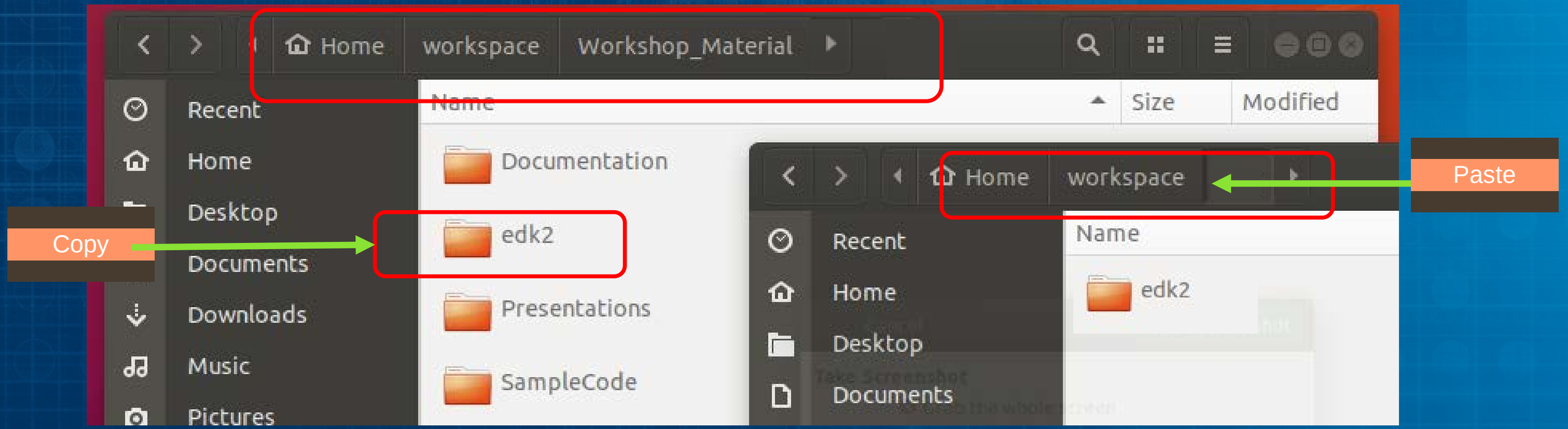
example scripts See: ~/workshop-material/SampleCode/Qemu/RunQemuDebug.sh

BUILD EDK II OVMF
-Getting the Source

From the Docker terminal window, Copy the edk2 directory to the docker ~/workspace

```
bash$ cp -R Workshop_Material/edk2 .
```

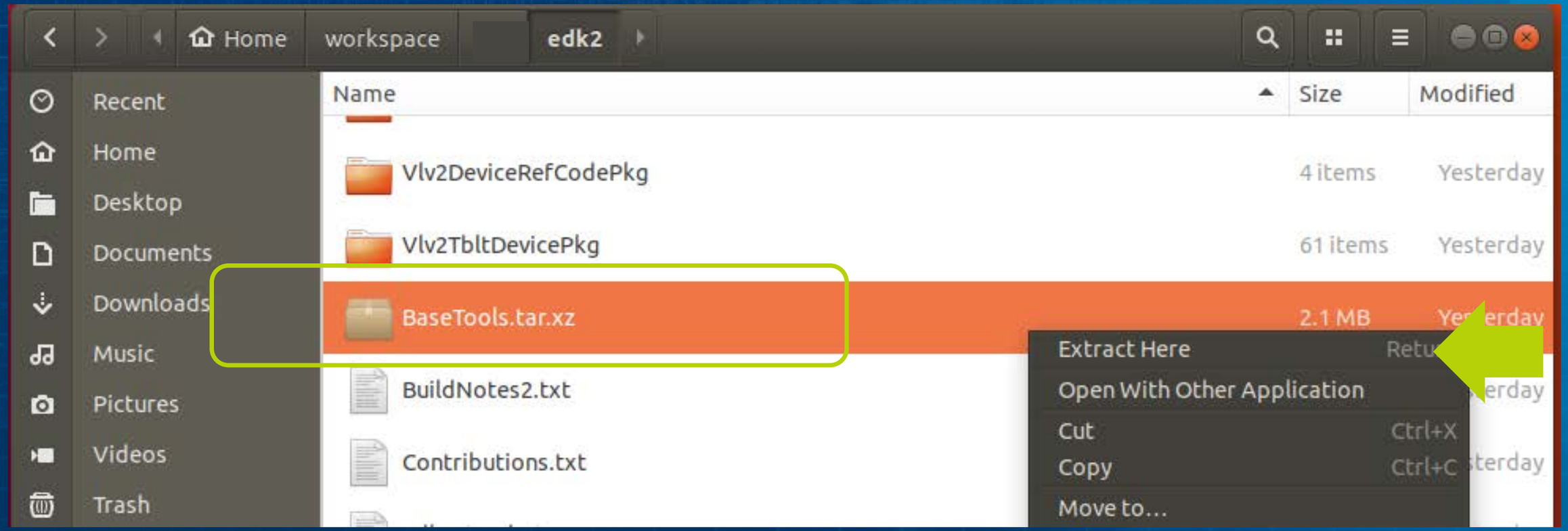
From the FW folder, copy and paste folder “~/../edk2” to ~/workspace



BUILD EDK II OVMF
-Getting BaseTools

From the folder ~/workspace/edk2, Extract the BaseTools.tar.xz to edk2 directory.

```
bash$ cd ~/workspace/edk2
bash$ tar -xf BaseTools.tar.xz
```



BUILD EDK II OVMF

- Building the Base Tools

Run Make from the Docker Terminal window

```
bash$ cd ~/workspace/edk2
bash$ make –C BaseTools
```

```
testSurrogatePairUnicodeCharInUtf16File (CheckUnicodeSourceFiles.Tests) ... ok
testSurrogatePairUnicodeCharInUtf8File (CheckUnicodeSourceFiles.Tests) ... ok
testSurrogatePairUnicodeCharInUtf8FileWithBom (CheckUnicodeSourceFiles.Tests) ..
. ok
testUtf16InUniFile (CheckUnicodeSourceFiles.Tests) ... ok
testValidUtf8File (CheckUnicodeSourceFiles.Tests) ... ok
testValidUtf8FileWithBom (CheckUnicodeSourceFiles.Tests) ... ok

----------------------------------------------------------------------
Ran 263 tests in 3.218s

OK
make[1]: Leaving directory '/home/dockeruser/workspace/edk2/BaseTools/Tests'
make: Leaving directory '/home/dockeruser/workspace/edk2/BaseTools'
[edk2-ubuntu] edk2 #
```

About 15 seconds

Run edksetup (note This will need to be done for every new Docker Terminal window)

```
bash$ . edksetup.sh
```

```
lju@lju-VirtualBox:~/src/edk2$  . edksetup.sh BaseTools
WORKSPACE: /home/lju/src/edk2
EDK_TOOLS_PATH: /home/lju/src/edk2/BaseTools
Copying $EDK_TOOLS_PATH/Conf/build_rule.template
     to $WORKSPACE/Conf/build_rule.txt
Copying $EDK_TOOLS_PATH/Conf/tools_def.template
     to $WORKSPACE/Conf/tools_def.txt
Copying $EDK_TOOLS_PATH/Conf/target.template
     to $WORKSPACE/Conf/target.txt
lju@lju-VirtualBox:~/src/edk2$
```

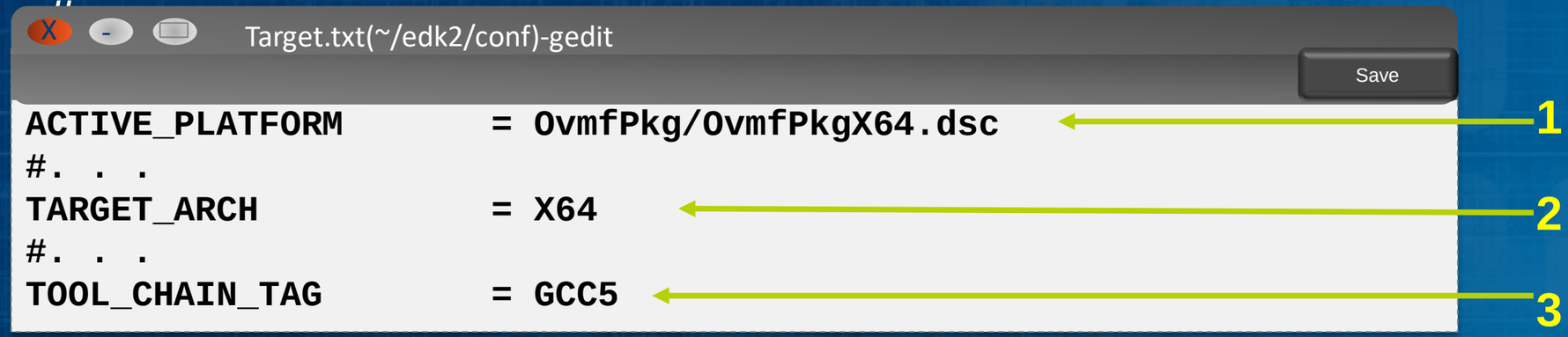
BUILD EDK II OVMF

-Update Target.txt and Build

Open Virtual Machine Firmware OVMF- Build

Edit the file Conf/target.txt

```
bash$ gedit Conf/target.txt
```



Save and Exit

Build from the edk2 directory

```
bash$ build -D SOURCE_DEBUG_ENABLE
```

or (if target.txt not updated)

```
bash$ build –p OvmfPkg/OvmfPkgX64.dsc –a X64 –t GCC5 -D SOURCE_DEBUG_ENABLE
```

Finished build

BUILD EDK II OVMF

-Inside Terminal

```
u-uefi@uuefi-TPad: ~/src/FW/edk2
BuildEnv.sh  build_rule.txt  ReadMe.txt  target.txt  tools_def.txt
u-uefi@uuefi-TPad:~/src/FW/edk2$ gedit Conf\target.txt
u-uefi@uuefi-TPad:~/src/FW/edk2$ gedit
u-uefi@uuefi-TPad:~/src/FW/ed
u-uefi@uuefi-TPad:~/src/FW/ed
Build environment: Linux-4.8.
Build start time: 11:51:26, J

WORKSPACE        = /home/u-ue
ECP_SOURCE       = /home/u-ue
EDK_SOURCE       = /home/u-ue
EFI_SOURCE       = /home/u-ue
EDK_TOOLS_PATH   = /home/u-ue
CONF_PATH        = /home/u-ue


Architecture(s)  = X64
Build target     = DEBUG
Toolchain        = GCC5

Active Platform          = /h
Flash Image Definition   = /h

Processing meta-data .....
```

```
ome/lju/src/edk2/MdePkg/Include -I/home/lju/src/edk2/MdePkg/Include/X64 /home/l
u/src/edk2/MdePkg/Library/BasePrintLib/PrintLib.c
"echo" Symbol renaming not needed for /home/lju/src/edk2/Build/OvmfX64/DEBUG_GC
44/X64/MdePkg/Library/BasePrintLib/BasePrintLib/OUTPUT/./PrintLib.obj
Symbol renaming not needed for /home/lju/src/edk2/Build/OvmfX64/DEBUG_GCC44/X64
MdePkg/Library/BasePrintLib/BasePrintLib/OUTPUT/./PrintLib.obj
"/usr/bin/ar" -cr /home/lju/src/edk2/Build/OvmfX64/DEBUG_GCC44/X64/MdePkg/Libra
y/BasePrintLib/BasePrintLib/OUTPUT/BasePrintLib.lib  @/home/lju/src/edk2/Build/
vmfX64/DEBUG_GCC44/X64/MdePkg/Library/BasePrintLib/BasePrintLib/OUTPUT/object_f
les.lst
Building ... /home/lju/src/edk2/PcAtChipsetPkg/Library/SerialIoLib/SerialIoLib.
nf [X64]
"/usr/bin/gcc" -g -fshort-wchar -fno-stack-protector -fno-strict-aliasing -Wall
-Werror -Wno-missing-braces -Wno-array-bounds -ffunction-sections -fdata-sectio
s -c -include AutoGen.h -DSTRING_ARRAY_NAME=PcAtSerialPortLibStrings -m64 "-DEF
API=__attribute__((ms_abi))" -DNO_BUILTIN_VA_FUNCS -mno-red-zone -Wno-address -
cmodel=large -mno-mmx -mno-sse -o /home/lju/src/edk2/Build/OvmfX64/DEBUG_GCC44/
64/PcAtChipsetPkg/Library/SerialIoLib/SerialIoLib/OUTPUT/./SerialPortLib.obj -I
home/lju/src/edk2/PcAtChipsetPkg/Library/SerialIoLib -I/home/lju/src/edk2/Build
Ov
ho
2/
lP
```

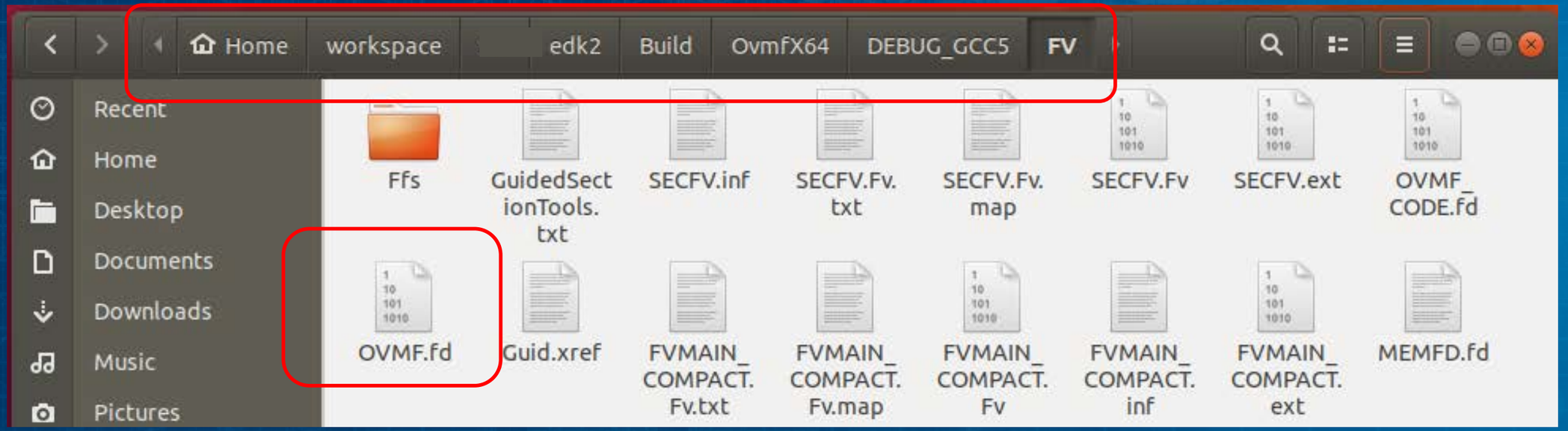
```
GUID cross reference file can be found at /home/u-uefi/src/edk2/Build/OvmfX64/DE
BUG_GCC5/FV/Guid.xref

FV Space Information
SECFV [10%Full] 212992 total, 22288 used, 190704 free
FVMAIN_COMPACT [33%Full] 3440640 total, 1157592 used, 2283048 free
DXEFV [38%Full] 10485760 total, 3990040 used, 6495720 free
PEIFV [19%Full] 917504 total, 177768 used, 739736 free

- Done -
Build end time: 09:20:45, Jun.14 2017
Build total time: 00:00:19
```

Build EDK II OVMF
-Verify Build Succeeded

OVMF.fd should be in the Build directory

- For GCC5 with X64, it should be located at

```
~/workspace/edk2/Build/OvmfX64/DEBUG_GCC5/FV/OVMF.fd
```



Invoke QEMU

1. Change to run-ovmf directory under the home directory

```
bash$ cd ~/workspace/run-ovmf
```

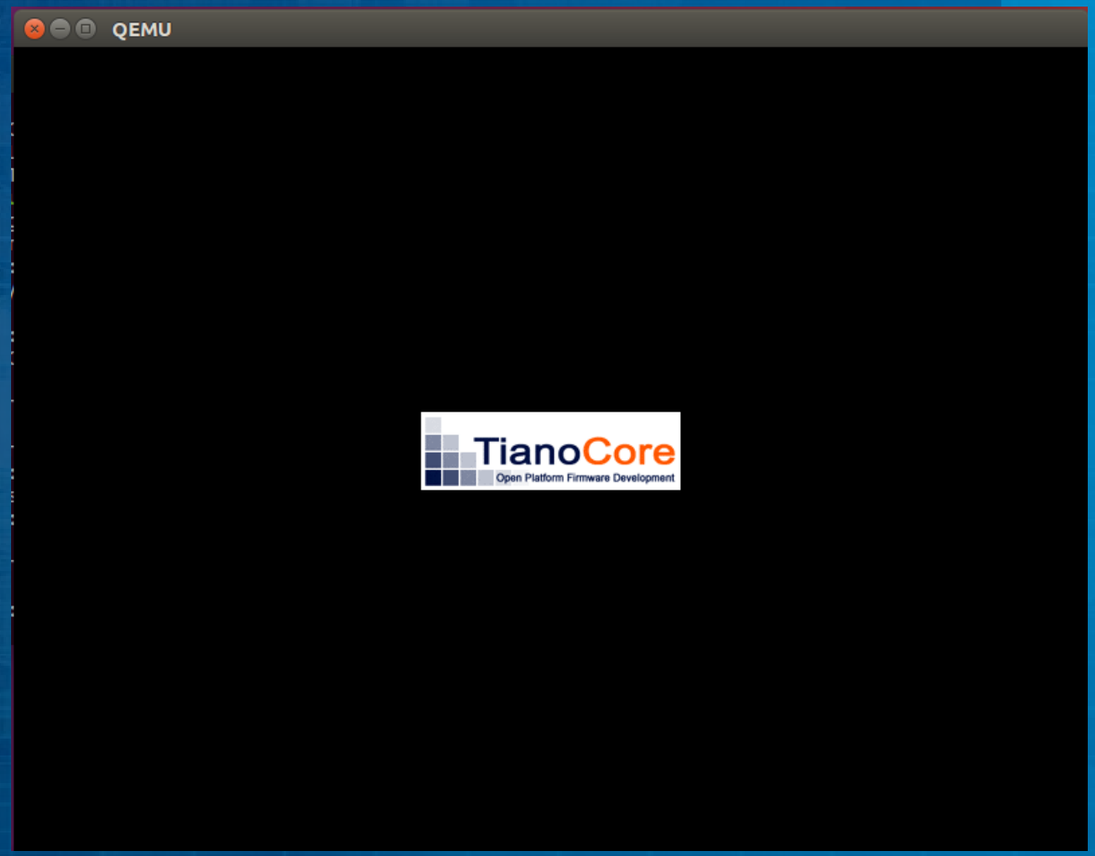
2. Copy the OVMF.fd BIOS image created from the build to the run-ovmf directory naming it bios.bin

```
bash$ cp
~/workspace/edk2/Build/OvmfX64/DEBUG_GCC5
/FV/OVMF.fd    bios.bin
```

3. Run the RunQemu.sh Linux shell script

```
bash$ . RunQemu.sh
```

4. Exit Qemu



UDK DEBUGGER

Update the Configuration for the Intel® UDK Debugger Tool

Intel® UDK Debugger Tool

Configure Debug Port Menu

Edit Configuration file: /etc/udkdebugger.conf

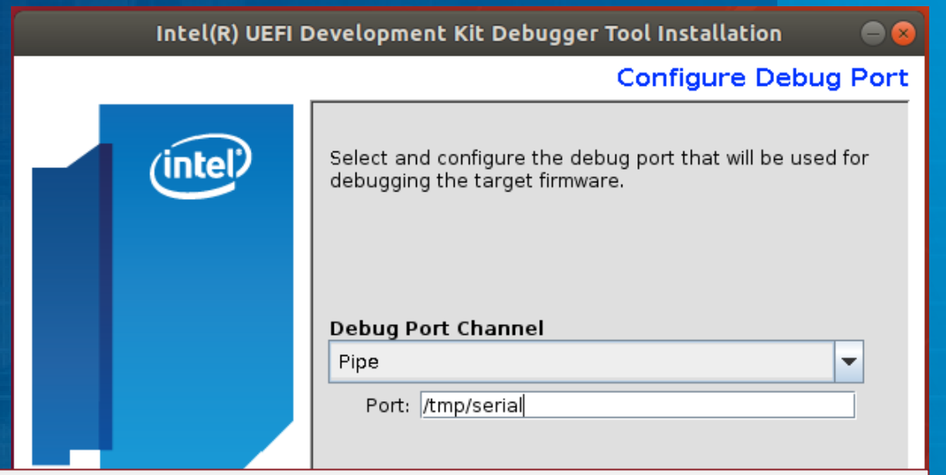
```
bash$ gedit /etc/udkdebugger.conf
```

Change `Channel = Pipe` and `Port = /tmp/serial`

Save the file

Copy the file for successive sessions

```
bash$ cp /etc/udkdebugger.conf ~/workspace
```




```
[Debug Port]
Channel = Pipe
Port = /tmp/serial
FlowControl = 1
BaudRate = 115200
Server =

[Target System]
```

File: udkdebugger.conf

DEBUG A DRIVER

Build a UEFI Driver and use the Debugger tools to Debug

Simple UEFI Driver - MyUefiDriver

MyUEFIDriver – UEFI Driver Produces:

- Driver Binding Protocol
- Component Name Protocol
- HII forms / Fonts w/ supporting Protocols
- My Dummy Protocol –

  Updates NVRAM Variable – Unicode string

  Updates a Memory buffer – Unicode char

MyApp – UEFI application interfaces with My Dummy Protocol

See Readme.md Workshop-Material/SampleCode/UEFI-Driver

Debugging `MyUefiDriver` from the UEFI Shell

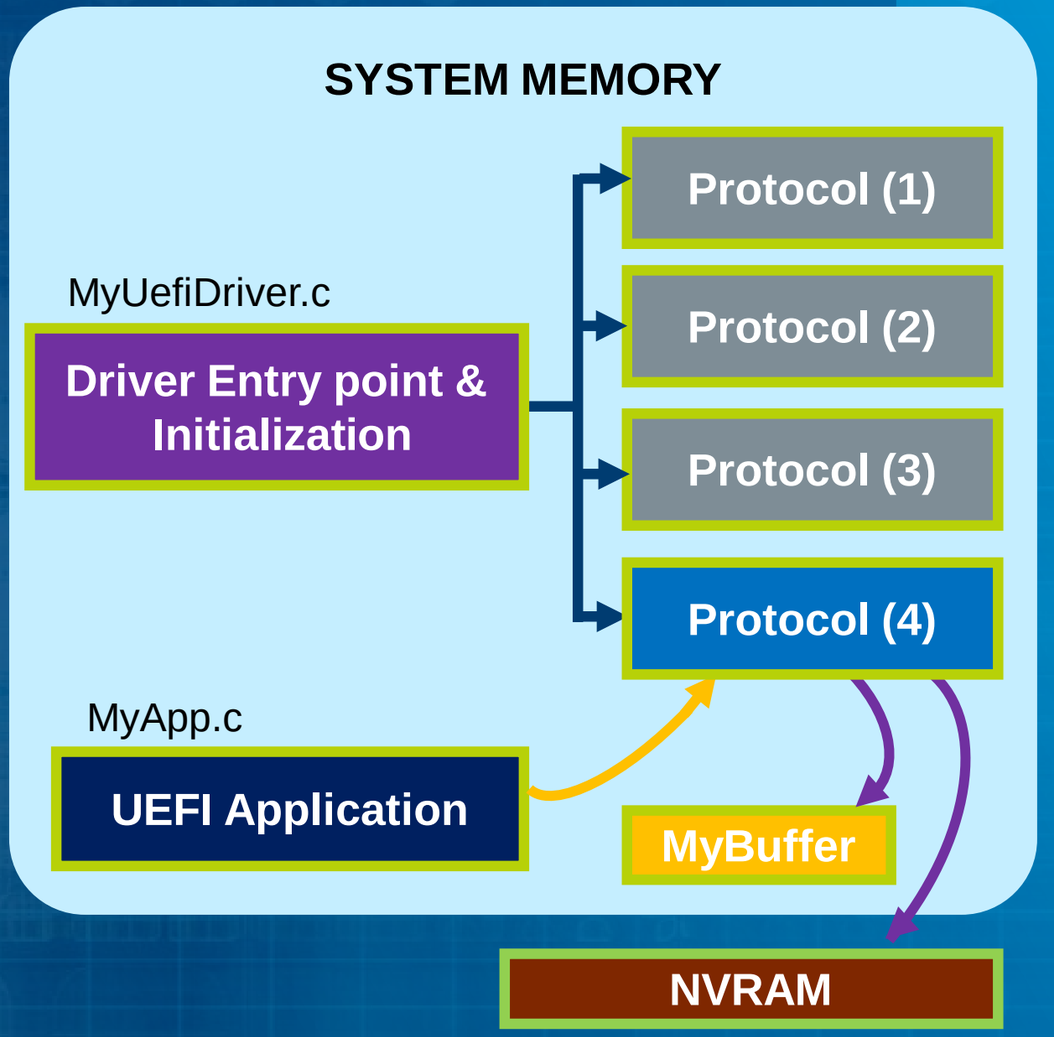
`MyUefiDriver` produces

Protocol (4) – My dummy protocol

- String write to NVRAM
- String clear NVRAM
- Char write memory buffer
- Char clear memory buffer

`MyApp` consumes Protocol (4)

Use the UEFI Shell for debugging `MyUefiDriver`



Copy Simple Driver to Your Workspace

Open Directory Workshop-Material/SampleCode/UEFI-Drivers

```
bash$ cd ~/workshop/Workshop-Material/SampleCode/UEFI-Drivers
```

Copy the sample driver to the edk2 directories

```
bash$ cp –R MyUefiDriver/  ~/workspace/edk2/
bash$ cp –R Protocol/  ~/workspace/OvmfPkg/Include/
bash$ cp OvmfPkg.dec  ~/workspace/edk2/OvmfPkg/
bash$ cp OvmfPkgX64.dsc  ~/workspace/edk2/OvmfPkg/
bash$ cp –R Bin/ ~/workspace/run-ovmf/hda-contents/
```

Build UEFI Driver & Test

1. Build with Source Debugger

```
bash$ cd ~/workspace/edk2
bash$ build -D SOURCE_DEBUG_ENABLE
```

2. Change to run-ovmf directory under the home directory

```
bash$ cd ~/workspace/run-ovmf
```

3. Copy the OVMF.fd BIOS image created from the build to the run-ovmf directory naming it bios.bin

```
bash$ cp ~/workspace/edk2/Build/OvmfX64/DEBUG_GCC5/FV/OVMF.fd bios.bin
```

4. Copy MyUefiDriver and MyApp to the Qemu file system hda-contents

```
bash$ cp ~/workspace/edk2/Build/OvmfX64/DEBUG_GCC5/X64/MyApp.efi hda-
contents/.
bash$ cp ~/workspace/edk2/Build/OvmfX64/DEBUG_GCC5/X64/MyUefiDriver.efi hda-
contents/.
```

Test UEFI Driver in Qemu

1. Run QEMU

```
bash$ . RunQemu.sh
```

2. Load the UEFI Driver at the shell prompt

```
Shell> fs0:
FS0:> Load MyUefiDriver.efi
```

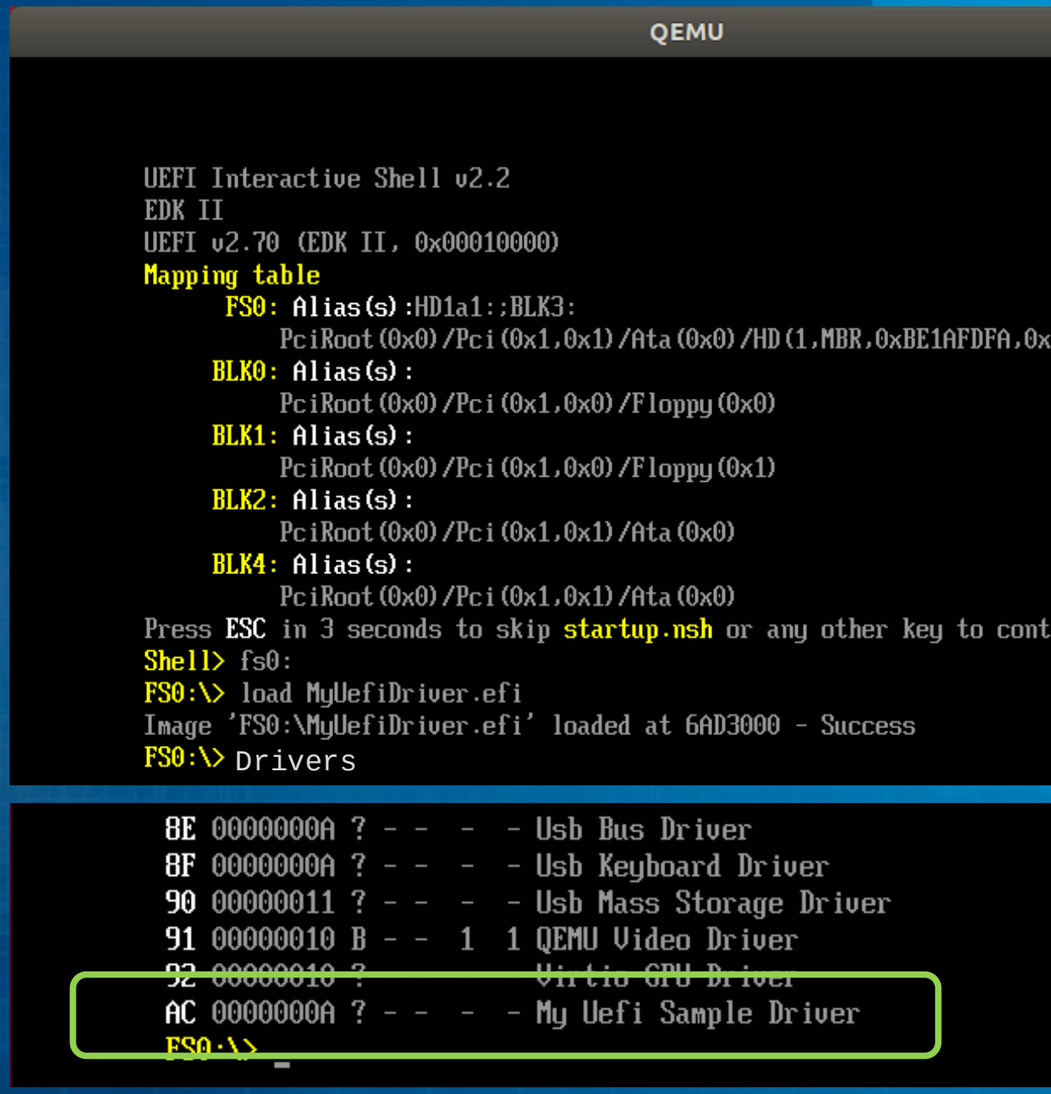
3. Test with Drivers command

```
FS0:> Drivers
```

See “My Uefi Sample Driver” in the list

4. Run the MyApp that calls the protocols

```
FS0:> MyApp.efi
```



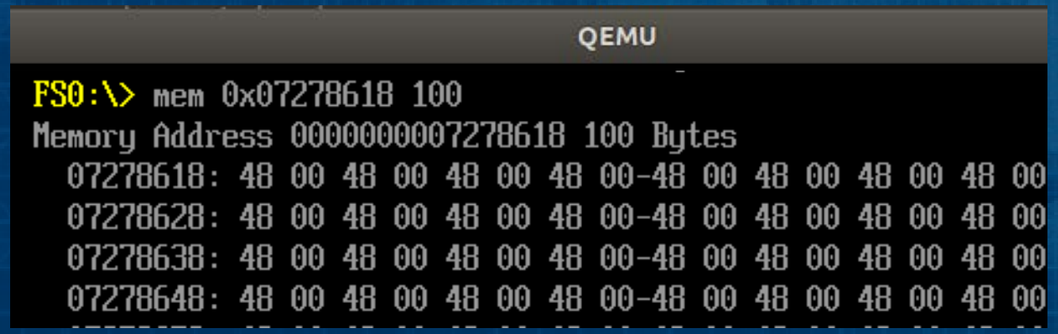
Test UEFI Driver in Qemu

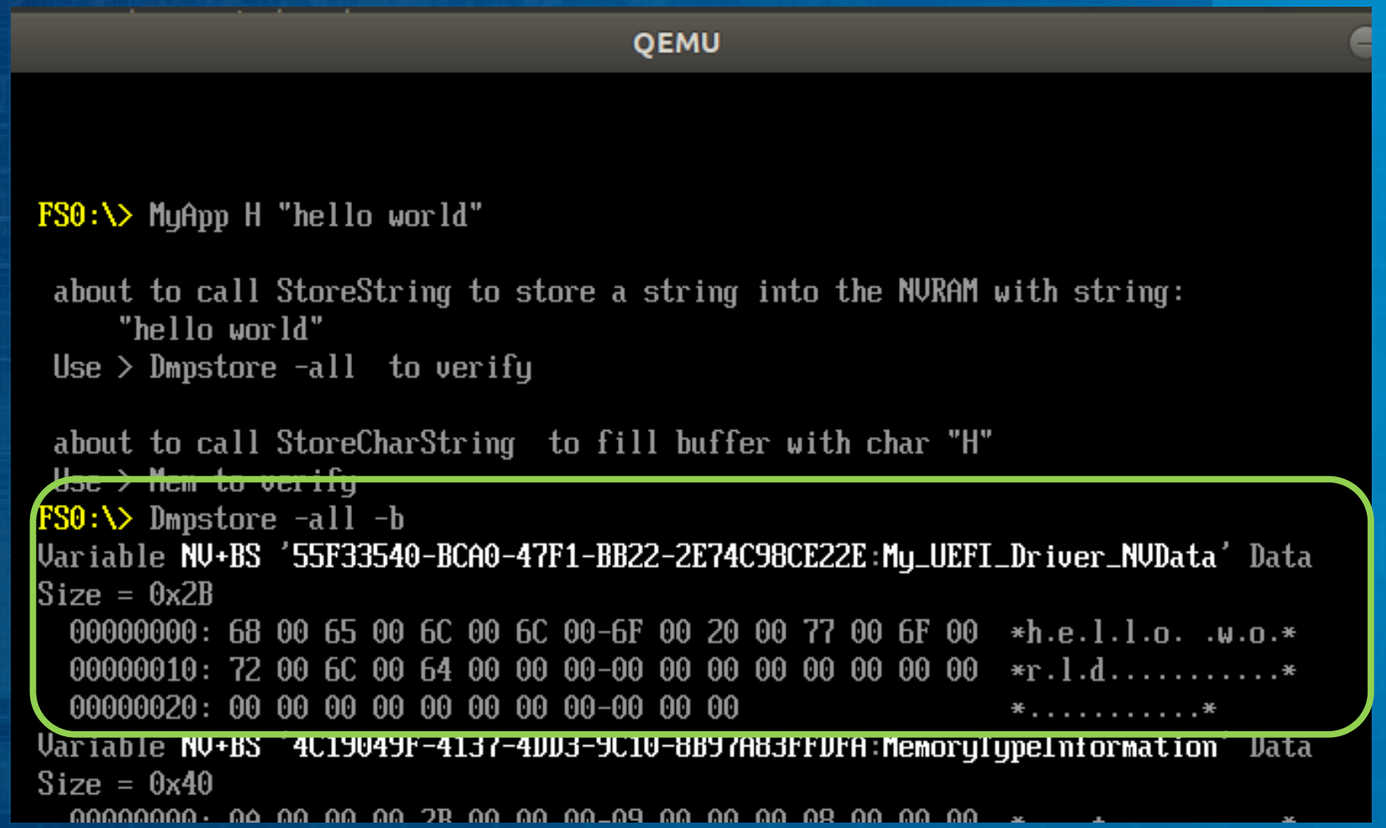
Test the MyApp interface with the dummy protocol

```
FS0:> MyApp.efi H “hello world”
FS0:> Dmpstore –all –b
```

Then use Mem on the address found in `debug.log`

```
FS0:> Mem 0x07278618
```





Debugging with DEBUGLIB

The MyUefiDriver has Debug print statements included

Check debug.log in run-ovmf dir

```
bash$ cat debug.log
```

- Notice all the Debug statements from the Supported function and only one from the Start function.
- Notice the Memory buffer location that is created in the start.
- Notice the install of MyDummyProtocol in the Start

```
>>>>>[MyUefiDriver] mMyUefiDriverDummyDevice data at: 0x067E5B
[MyUefiDriver]   Variable My_UEFI_Driver_NVData created in NVR
[MyUefiDriver] Not Supported
[MyUefiDriver] Supported SUCCESS
[MyUefiDriver] Buffer 0x07278618
InstallProtocolInterface: 09576E91-6D3F-11D2-8E39-00A0C969723B
InstallProtocolInterface: 5DA5C341-3970-4055-9168-57C3C8E0DD3B
[MyUefiDriver] Not Supported
[MyUefiDriver] Not Supported
[MyUefiDriver] Not Supported
[MyUefiDriver] Not Supported
[MyUefiDriver] Not Supported
[MyUefiDriver] Not Supported
[MyUefiDriver] Not Supported
[MyUefiDriver] Not Supported
[MyUefiDriver] Not Supported
[MyUefiDriver] Not Supported
[MyUefiDriver] Not Supported
[MyUefiDriver] Not Supported
[MyUefiDriver] Not Supported
[MyUefiDriver] Not Supported
```

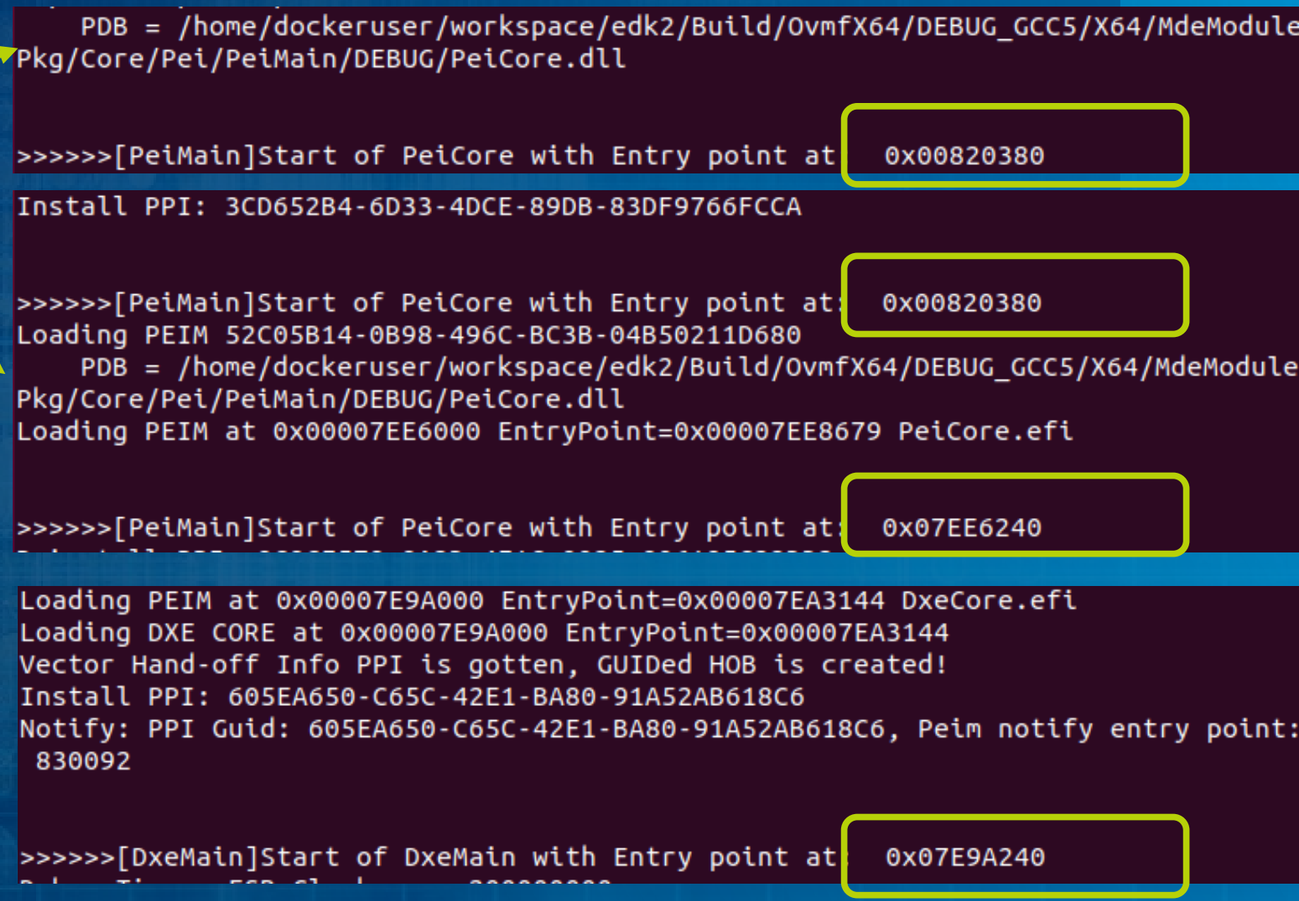
Debugging with DEBUGLIB

Check debug.log for PeiMain and DxeMain

- Notice PeiMain is invoked 3 times
  1. Beginning with Temp Memory
  2. After memory Initiation
  3. Address in System memory

- Notice DxeCore and entry point for DxeMain
- Most of the Initialization is done in DxeCore
- Make note of addresses to be later used for breakpoints in gdb /udk

EXIT QEMU



INVOKE DEBUGGER

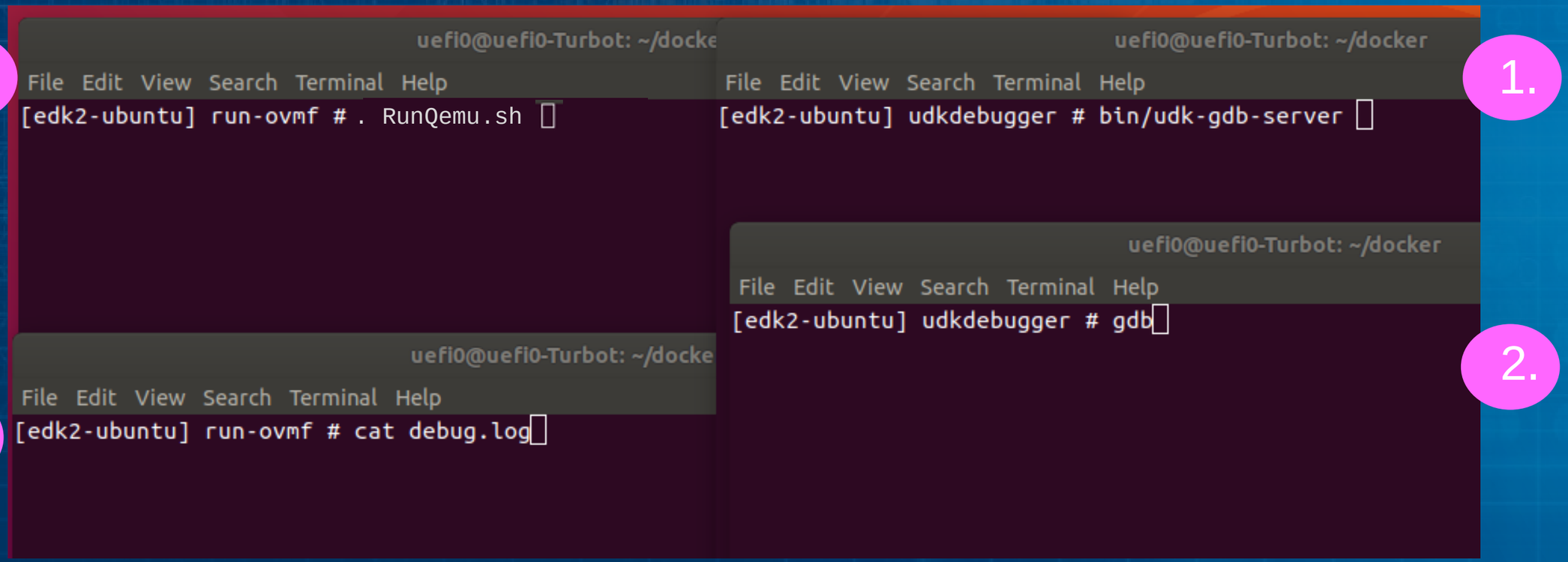
Invoking Intel® UDK, GDB and QEMU

Open 4 Terminal Console Windows

Open successive terminal windows with:

```
$> docker exec –it edk2 bash
```

1. Run Intel® UDK Debugger
2. Run Gdb
3. Run Qemu
4. Check debug.log with cat



1st Terminal window, UDK Debugger

1.

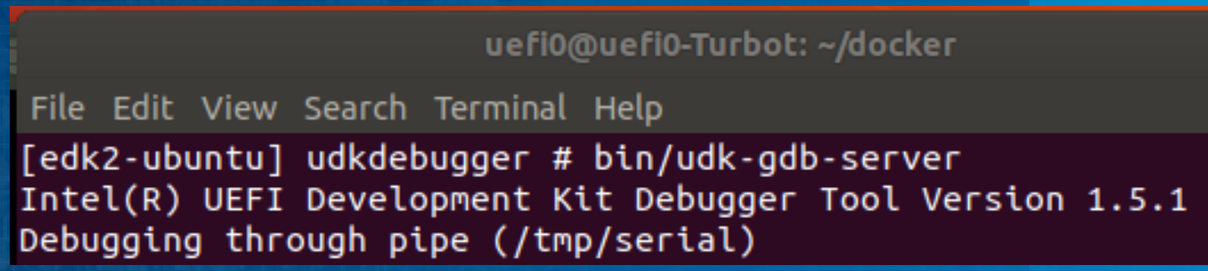
1st Terminal window

Invoke Intel® UDK Debugger

Open a terminal window in `opt/intel/udkdebugger`

```
bash$ cd /opt/intel/udkdebugger
bash$ ./bin/udk-gdb-server
```

Terminal (1)



2nd Terminal window, GDB

2.

2nd Terminal window

Invoke GDB (note `gdb --tui` to get layout src)

```
bash$ cd /opt/intel/udkdebugger
bash$ gdb
```

Terminal (2)

```
uefi0@uefi0-Turbot: ~/docker
File  Edit  View  Search  Terminal  Help
[edk2-ubuntu] udkdebugger # gdb
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.htm
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
(gdb)
```

3rd Terminal window (QEMU)

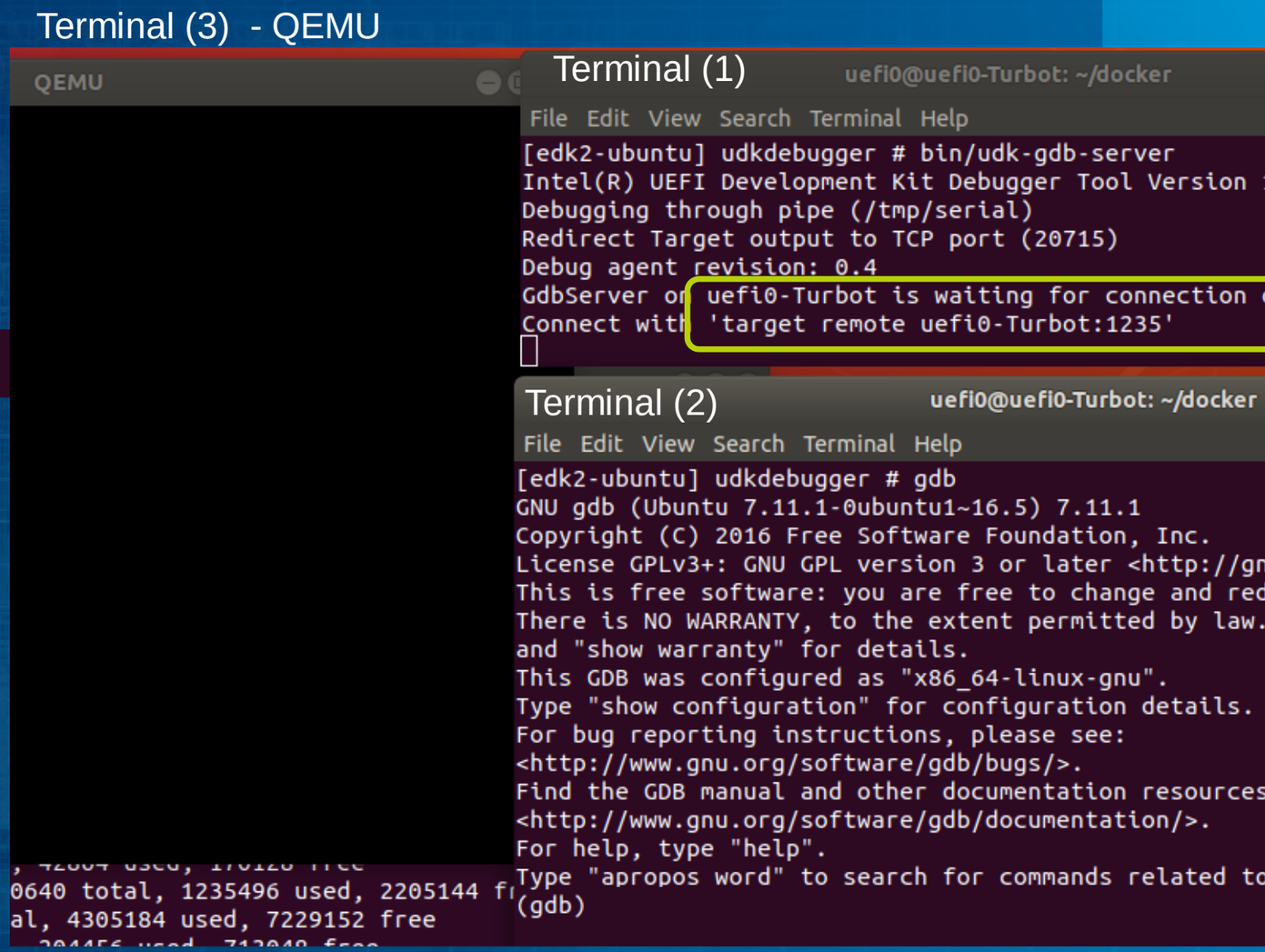
3.

3rd window

Invoke Qemu

```
bash$ . RunQemu.sh
```

Notice that the 2nd Terminal window (UDK) will see the Qemu and will print out the command you need to enter in the 3rd Terminal window (gdb)
Terminal(2) will show "Connection from localhost" message
Attach to the UDK debugger
Connect with ‘target remote <HOST>:1234’



4th Terminal window

4.

4th window

```
bash$ cd ~/workspace/run-ovmf
```

Periodically use cat debug.log to check debug output

```
bash$ cat debug.log
```

OR Edit source and rebuild in

```
~/workspace/edk2
```

Terminal (4)

```
uefi0@uefi0-Turbot: ~/docker
File  Edit  View  Search  Terminal  Help
Loading PEIM at 0x00007E9A000 EntryPoint=0x00007EA3144 DxeCore.efi
Loading DXE CORE at 0x00007E9A000 EntryPoint=0x00007EA3144
Vector Hand-off Info PPI is gotten, GUIDed HOB is created!
Install PPI: 605EA650-C65C-42E1-BA80-91A52AB618C6
Notify: PPI Guid: 605EA650-C65C-42E1-BA80-91A52AB618C6, Peim notify en
 830092


>>>>>>[DxeMain]Start of DxeMain with Entry point at:  0x07E9A240
Debug Timer: FSB Clock    = 200000000
Debug Timer: Divisor      = 2
Debug Timer: Frequency    = 100000000
Debug Timer: InitialCount = 10000000
CoreInitializeMemoryServices:
  BaseAddress - 0x3F5A000 Length - 0x3CA6000 MinimalMemorySizeNeeded -
InstallProtocolInterface: 5B1B31A1-9562-11D2-8E3F-00A0C969723B 7EC2788
ProtectUefiImageCommon - 0x7EC2788
  - 0x0000000007E9A000 - 0x000000000002F000
InstallProtocolInterface: 09576E91-6D3F-11D2-8E39-00A0C969723B 7EC3840
InstallProtocolInterface: BB25CF6F-F1D4-11D2-9A0C-0090273FC1FD 7EC37E0
    PDB = /home/dockeruser/workspace/edk2/Build/OvmfX64/DEBUG_GCC5/X64/
Pkg/Core/Dxe/DxeMain/DEBUG/DxeCore.dll
HOBLIST address in DXE = 0x7A15018
Memory Allocation 0x0000000A 0x7F78000 - 0x7FFFFFF
Memory Allocation 0x0000000A 0x810000 - 0x81FFFF
```

2nd Terminal window - gdb

2.

In gdb window, type the following to attach to the UDK debugger

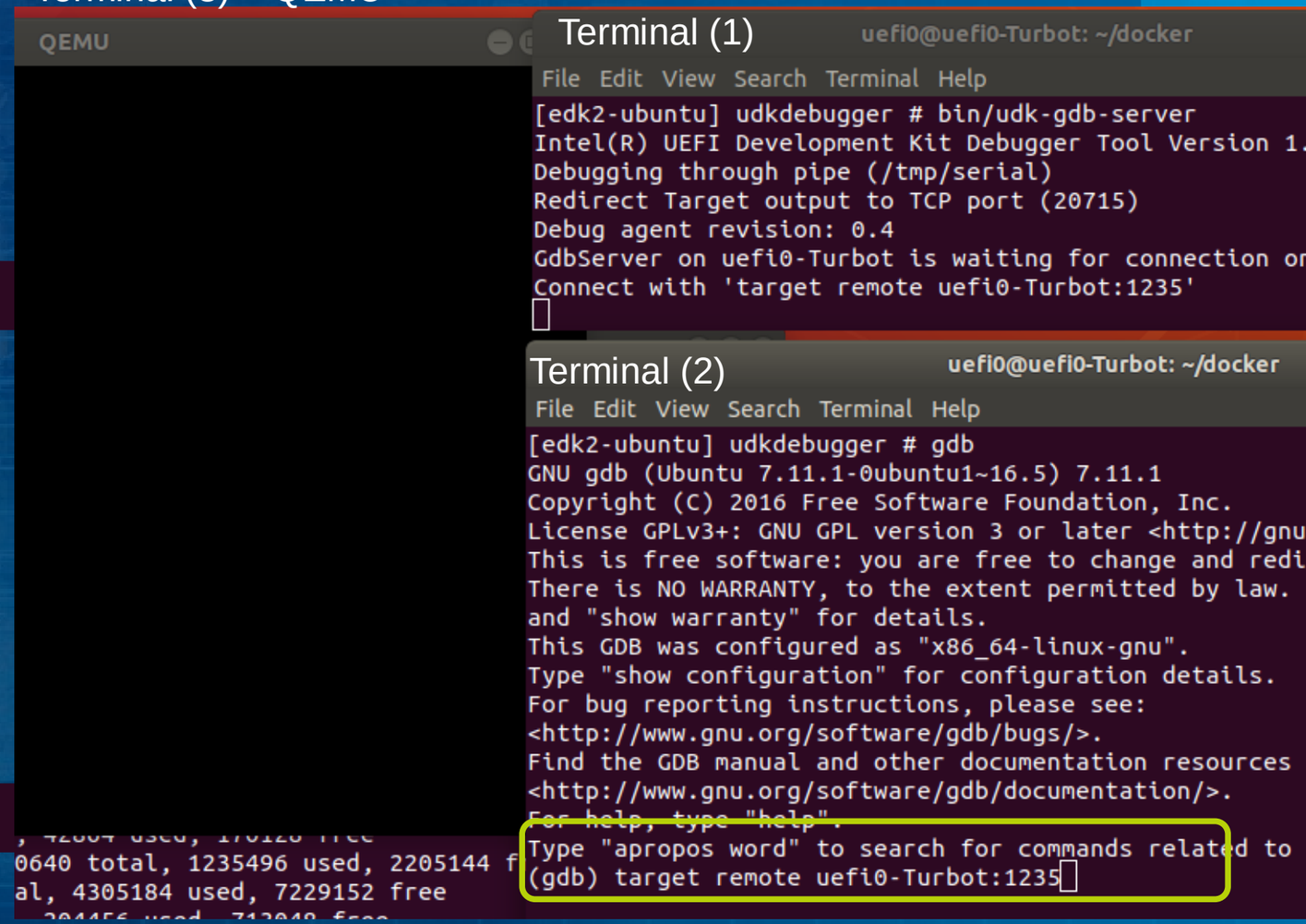
```
(gdb) target remote <HOST>:1234
```

This should be the same as in the 2nd Terminal window (UDK) shows. Note: “<HOST>” should be your Host or VM and 1234 will be the serial pipe port #

NOTE: For OpenSUSE use

```
(gdb) target remote 127.0.0.1:1234
```

Terminal (3) - QEMU



Note: 1235 was hued here

2nd Terminal window - gdb

2.

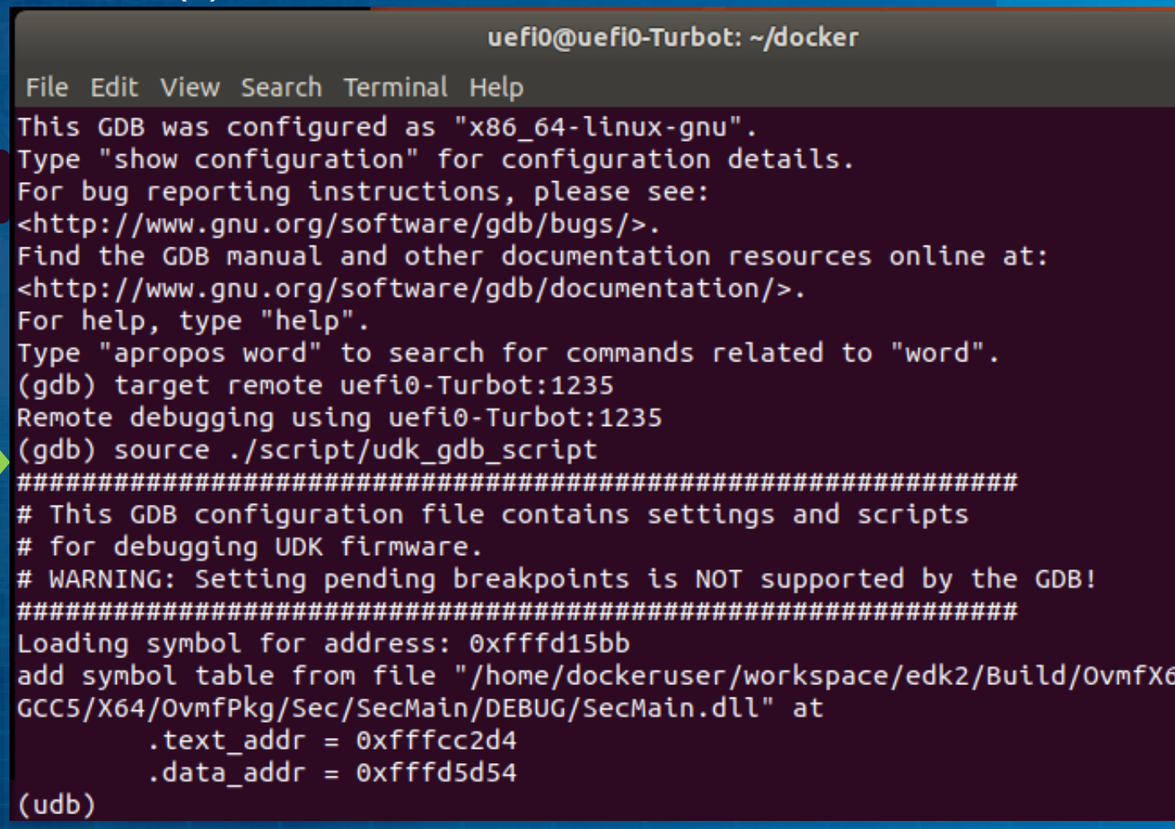
Now the gdb Terminal window will be in control of QEMU

Open the udk scripts in GDB – Terminal(2)

```
(gdb) source ./script/udk_gdb_script
```

Symbols will show for PeiCore, also notice the prompt changes from "(gdb)" to "(udb)"

Terminal (2)



2nd Terminal window -gdb

2.

At this point the Qemu is waiting because the udk has halted.

In gdb :

Set breakpoints (i.e. Address of DxeMain from debug.log)

```
(udk) b *0x07e9a240
```

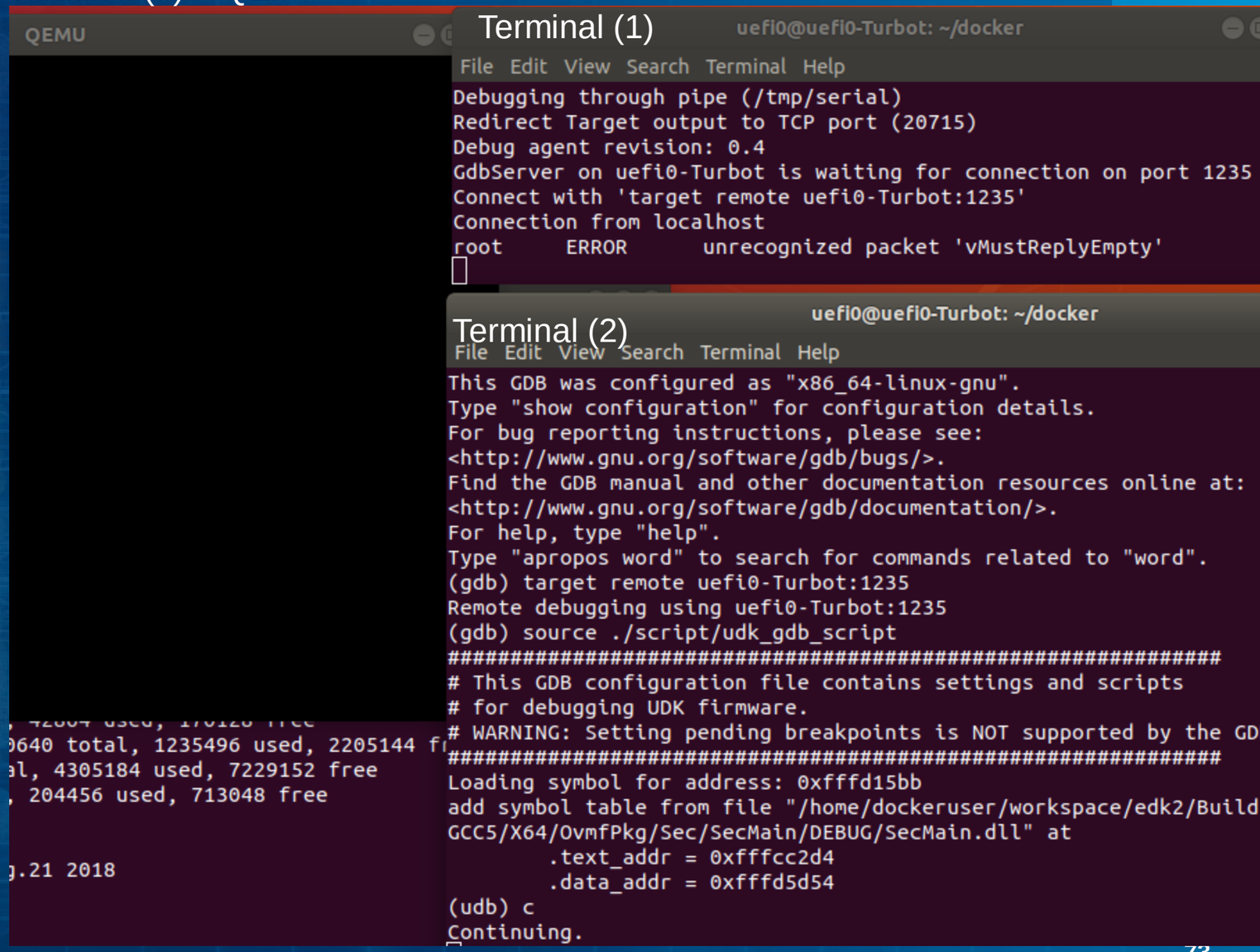
Continue

```
(udk) c
```

Warning unless a breakpoint is hit there is no way to Halt gdb once “continue” is invoked

DO NOT Cntl-Z out of the gdb while Qemu is running or you may have to reload the docker

Terminal (3) - QEMU



Example: Break at DXE Core

If a breakpoint was set for DxeMain, Debug could continue to debug DXE / UEFI drivers

Layout C Source code

```
(udk) layout src
```

Step Next Instruction

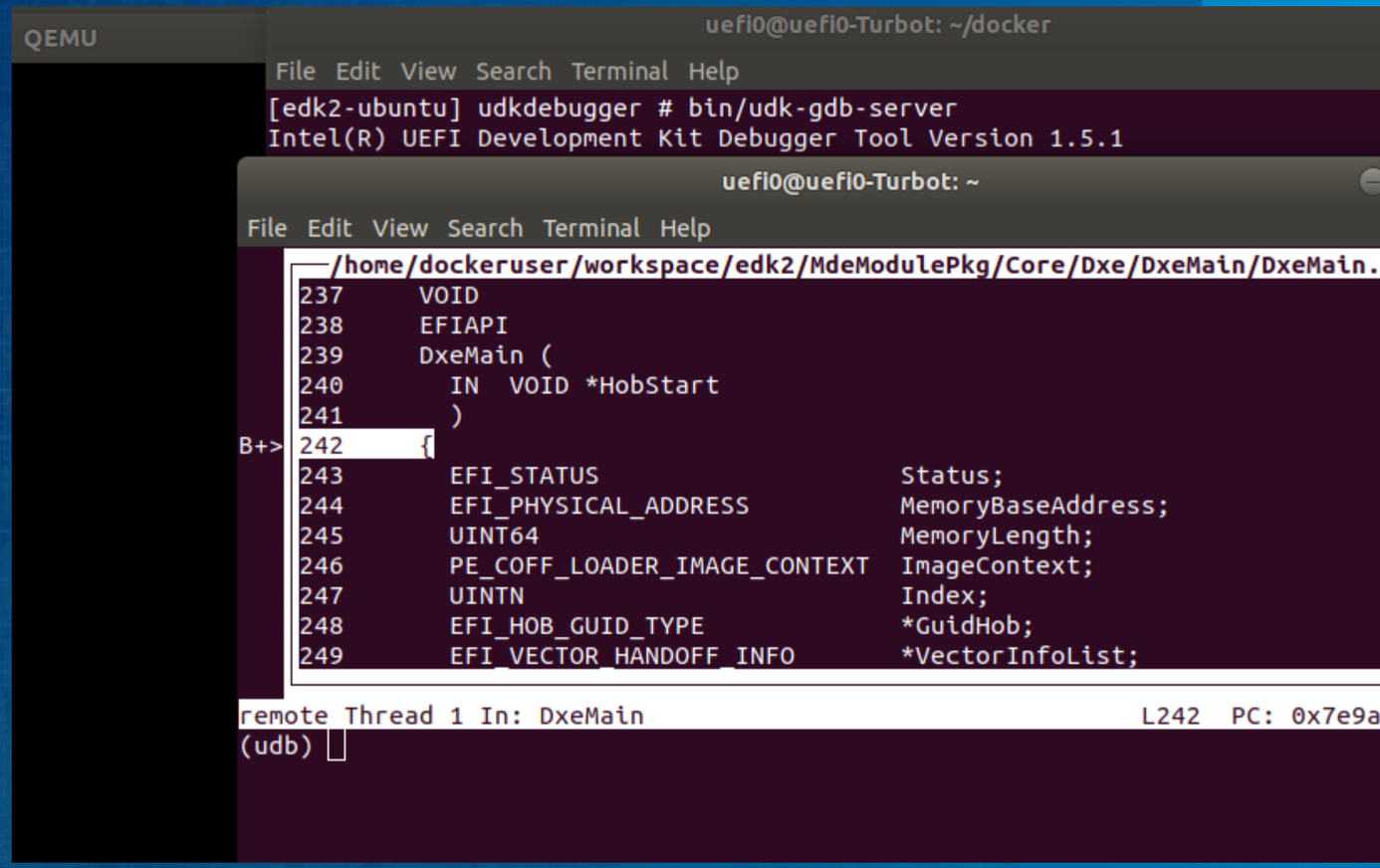
```
(udk) next
```

Show the Modules initialized

```
(udk) info modules
```

See gdb cheat sheet pdfs for more Documentation\gdb_UDK_Debugger

EXIT QEMU if gdb at prompt (udb)



Exit QEMU gdb “Continuing.”

Do not Exit Qemu if the gdb is in “Continuing.”

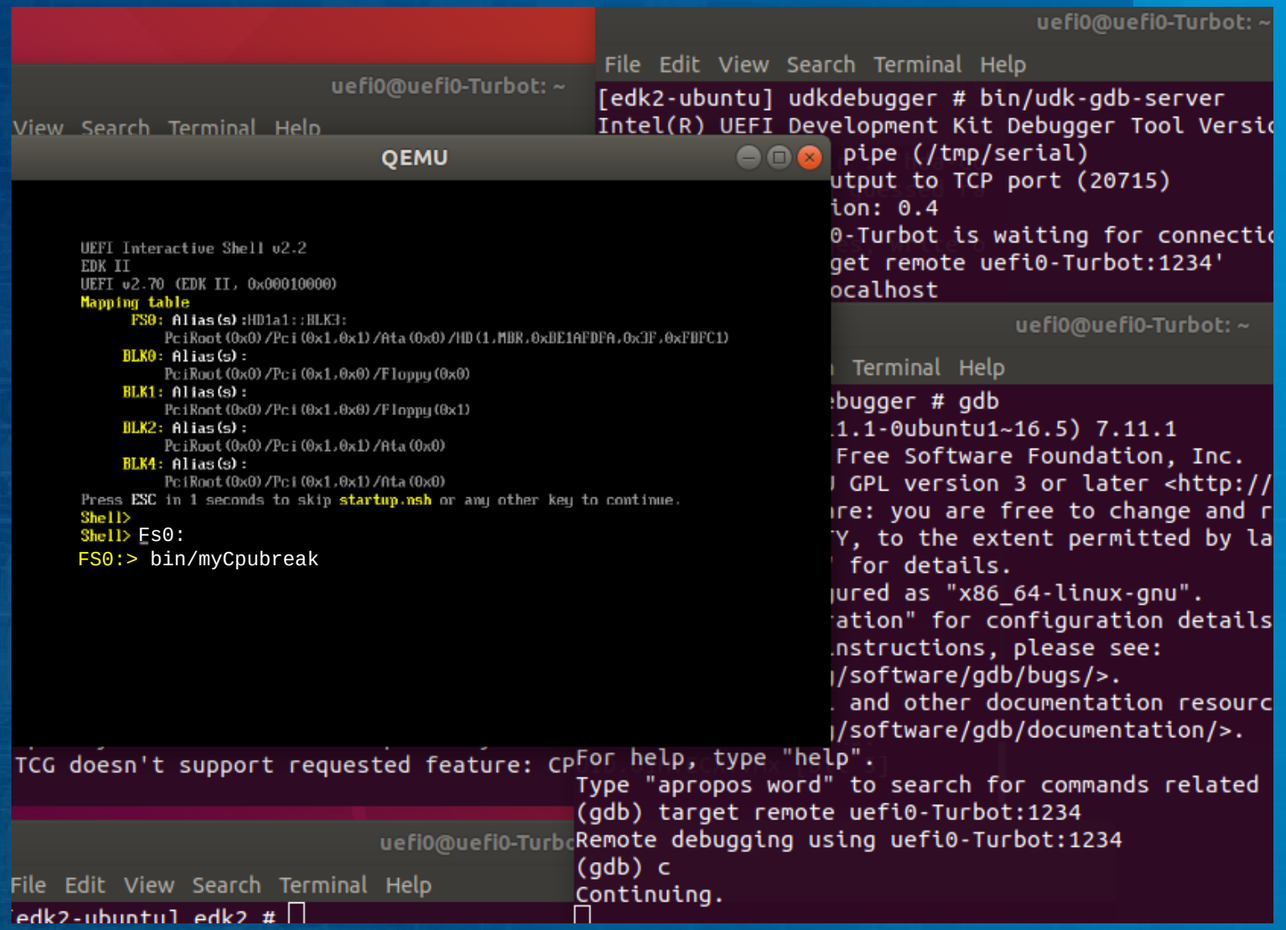
In invoke the app:

```
FS0:> bin\myCpubreak
```

This will force gdb to break at the and stop running Qemu

Then Exit Qemu by closing window for QEMU

DO NOT EXIT gdb



Successfully setup the debugger now lets debug a UEFI Driver

ADD CPU BREAKPOINT

Debug the UEFI Driver by adding function call “CpuBreakpoint()” in the code

Add CpuBreakpoint() Entry and Start

4.

Use 4th Terminal window to edit edk2/MyUefiDriver/MyUefiDriver.c and add CpuBreakpoint(); to the driver's :
Entry point and Start functions

Entry point - Line 246:

```
MyUefiDriverDriverEntryPoint (
  // . . .
)
{
  EFI_STATUS                    Status;
  EFI_HII_PACKAGE_LIST_HEADER *PackageListHeader;
  EFI_HII_DATABASE_PROTOCOL   *HiiDatabase;
  EFI_HII_HANDLE                HiiHandle;
  CpuBreakpoint();
  Status = EFI_SUCCESS;
```

Start function - Line 456:

```
MyUefiDriverDriverBindingStart (
// . . .)
{
MY_DUMMY_DEVICE_PATH     MyDummyDP;
EFI_DEVICE_PATH_PROTOCOL EndDP;
EFI_DEVICE_PATH_PROTOCOL *Dp1;
EFI_STATUS               Status;
CpuBreakpoint();
```

See Workshop-Material/SampleCode//UEFI-Driver01

Add `CpuBreakpoint()` in the UEFI Application

Edit edk2/MyUefiDriver/MyApp.c and add CpuBreakpoint(); to the beginning just after the debug print statement.

Approx.- Line 75:

```
EFI_STATUS
EFIAPI
MyAppUefiMain (
// . . .
  )
{
//. . .
 DEBUG((EFI_D_INFO, "\n\n>>>>>>[MyApp]Start of MyApp with Entry point at:  0x%08x \n", \
      MyAppUefiMain));
 CpuBreakpoint();

 //Initialize local protocol pointer
  EfiShellParametersProtocol = NULL;
```

See Workshop-Material/SampleCode/LabSolutions/UEFI-Driver01

4.

Build UEFI Driver & Test

1. Build with Source Debugger

```
bash$ cd ~/workspace/edk2
bash$ build -D SOURCE_DEBUG_ENABLE
```

2. Change to run-ovmf directory under the home directory

```
bash$ cd ~/workspace/run-ovmf
```

3. Copy the OVMF.fd BIOS image created from the build to the run-ovmf directory naming it bios.bin

```
bash$ cp ~/workspace/edk2/Build/OvmfX64/DEBUG_GCC5/FV/OVMF.fd bios.bin
```

4. Copy MyUefiDriver and MyApp to the Qemu file system had-contents

```
bash$ cp ~/workspace/edk2/Build/OvmfX64/DEBUG_GCC5/X64/MyApp.efi hda-
contents/.
bash$ cp ~/workspace/edk2/Build/OvmfX64/DEBUG_GCC5/X64/MyUefiDriver.efi hda-
contents/.
```

4 Terminal Console Windows

window 1 `bash$ bin/udk-gdb-server`
window 2 `bash$ Gdb`
window 3 `bash$ . RunQemu.sh`
window 4 `bash$ cat debug.log`

SKIP to Slide 83 if gdb and UDK are already running

2nd Terminal - gdb

2.

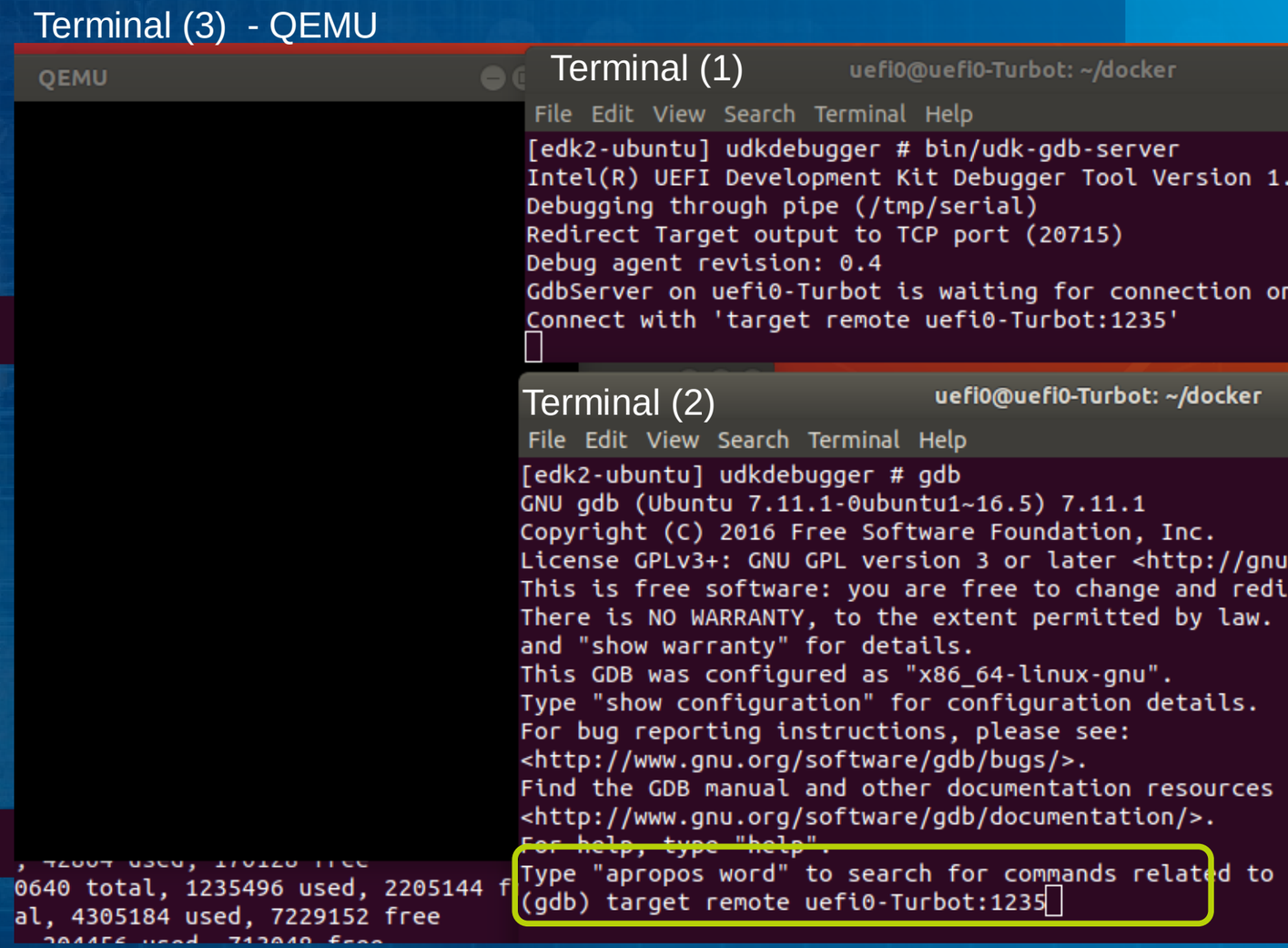
In gdb window, type the following to attach to the UDK debugger

```
(gdb) target remote <HOST>:1234
```

This should be the same as in the 2nd Terminal window (UDK) shows. Note: “<HOST>” should be your Host or VM and 1234 will be the serial pipe port #

NOTE: For OpenSUSE use

```
(gdb) target remote 127.0.0.1:1234
```



Note: 1235 was hued here

2nd Terminal window - gdb

2.

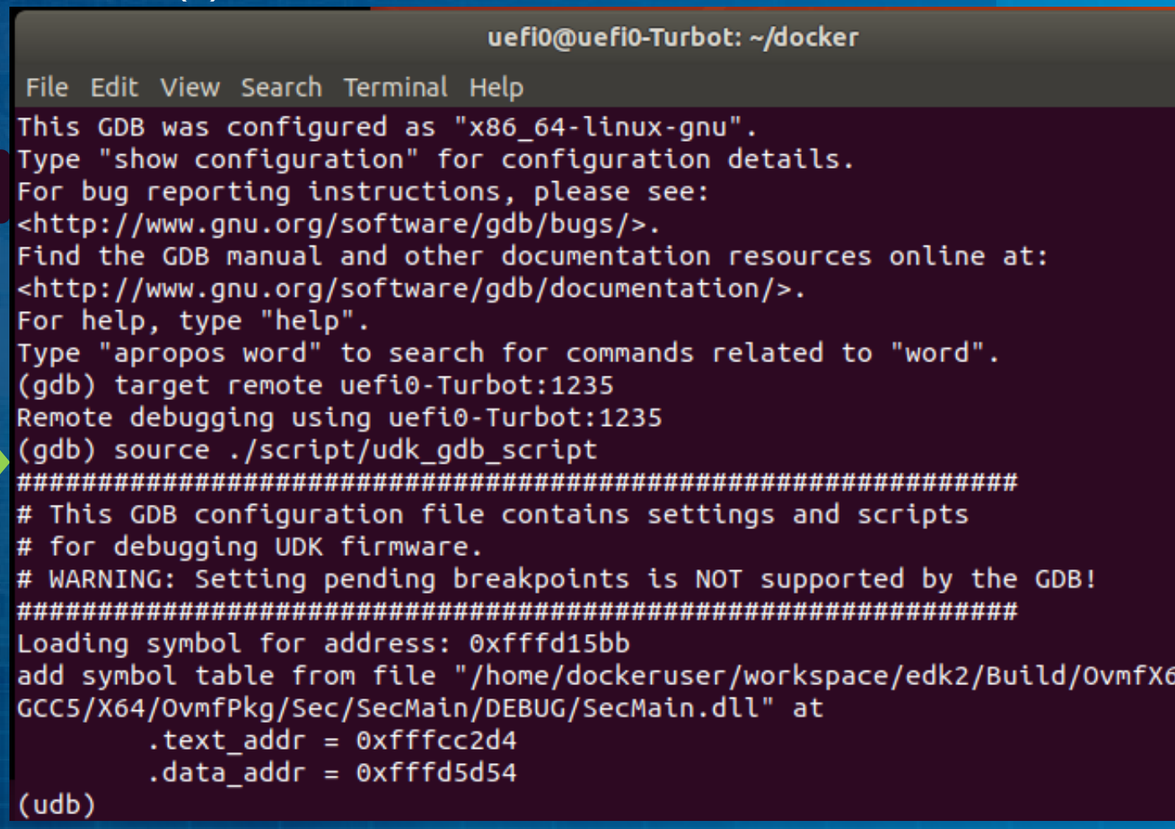
Now the gdb Terminal window will be in control of QEMU

Open the udk scripts in GDB – Terminal(2)

```
(gdb) source ./script/udk_gdb_script
```

Notice the prompt changes from "(gdb)" to "(udb)"

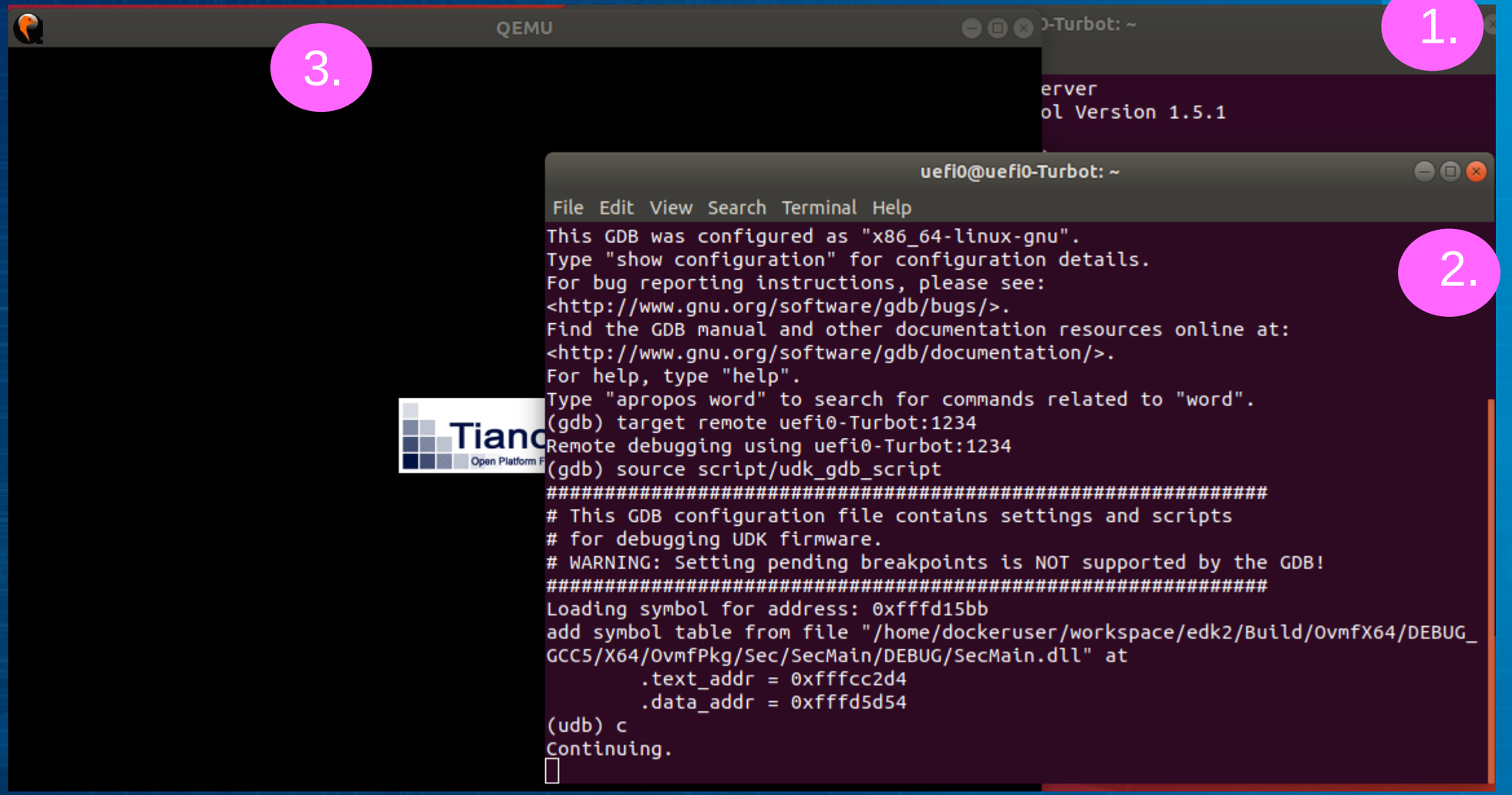
Terminal (2)



Debugging with GDB and UDK

Continue “c” will boot up to UEFI Shell

```
(udb) c
```



Debugging with GDB and UDK

At the UEFI Shell

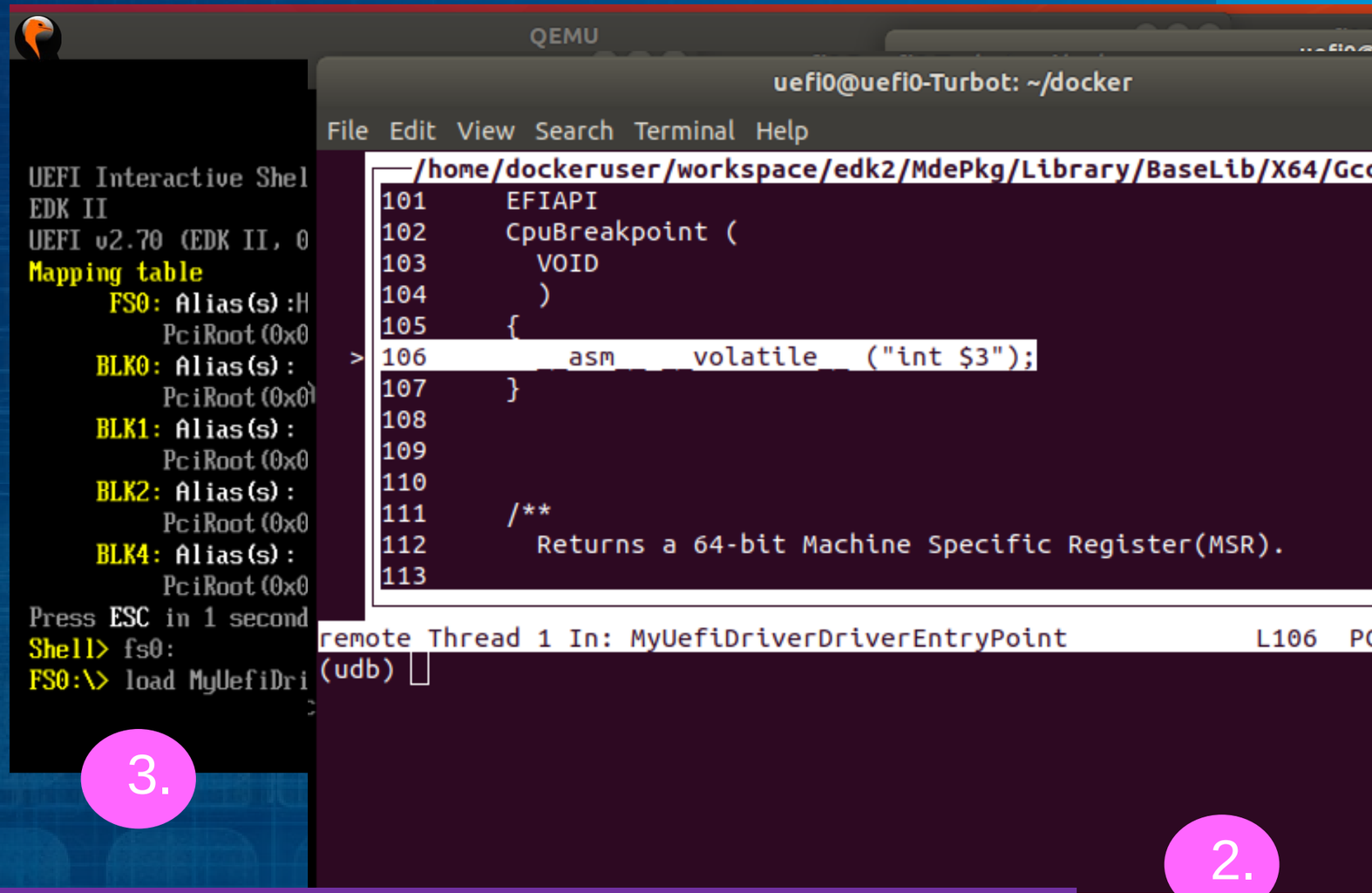
```
Shell > fs0:
FS0: > Load MyUefiDriver.efi
```

Notice that the CpuBreakpoint will stop the QEMU

Now move to GDB window (2)

At the (udb) window

```
(udb) Layout src
(udb) next
```



If the connection between gdb and udk fails – restart 4 new docker sessions : Link

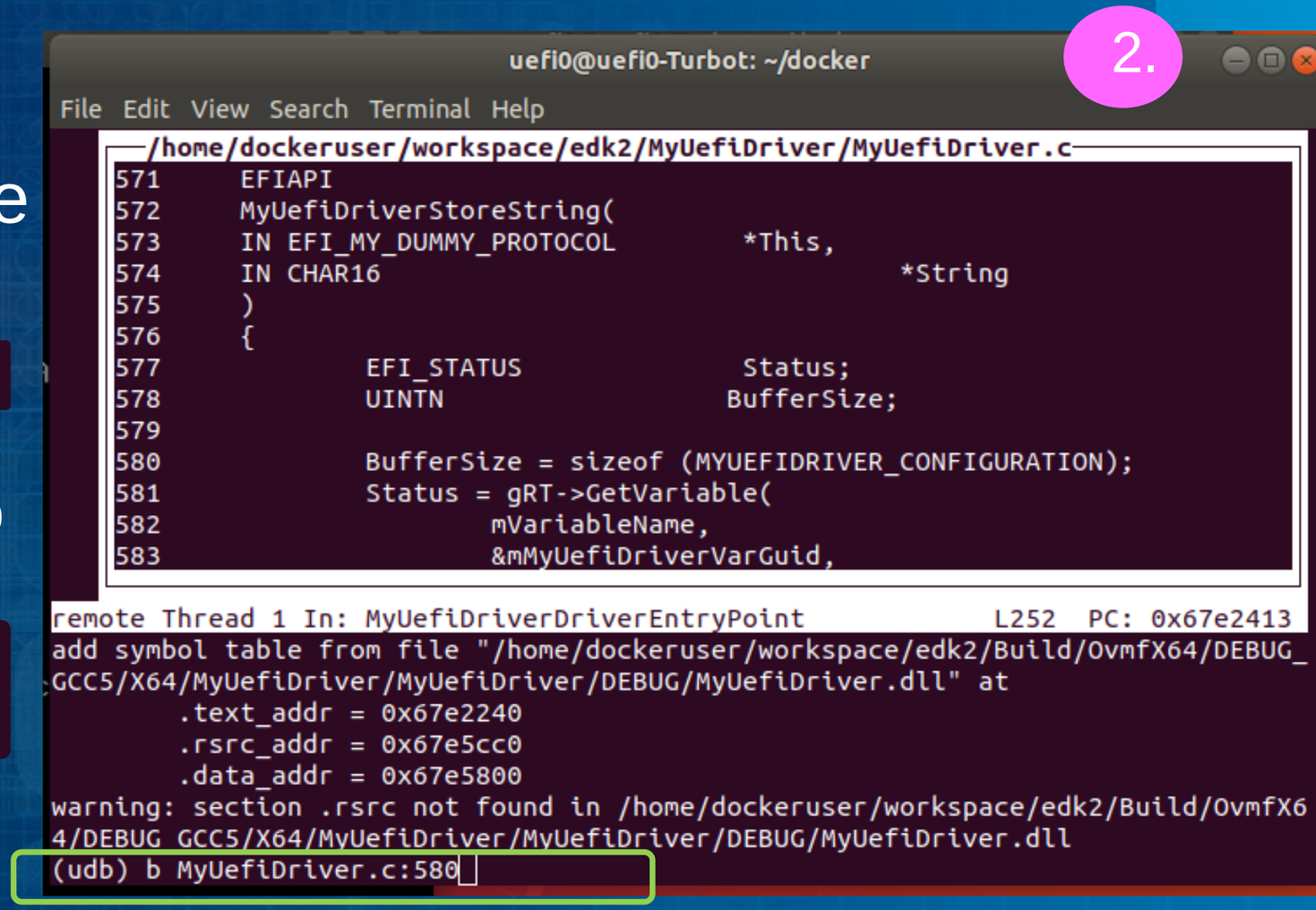
Debugging with GDB and UDK

At the (udb) window set a break point at line 580 in the MyUefiDriver.c

```
(udb) b MyUefiDriver.c:580
```

Swap between gdb source and gdb command window

```
(udb) tui disable
(udb) layout src
```



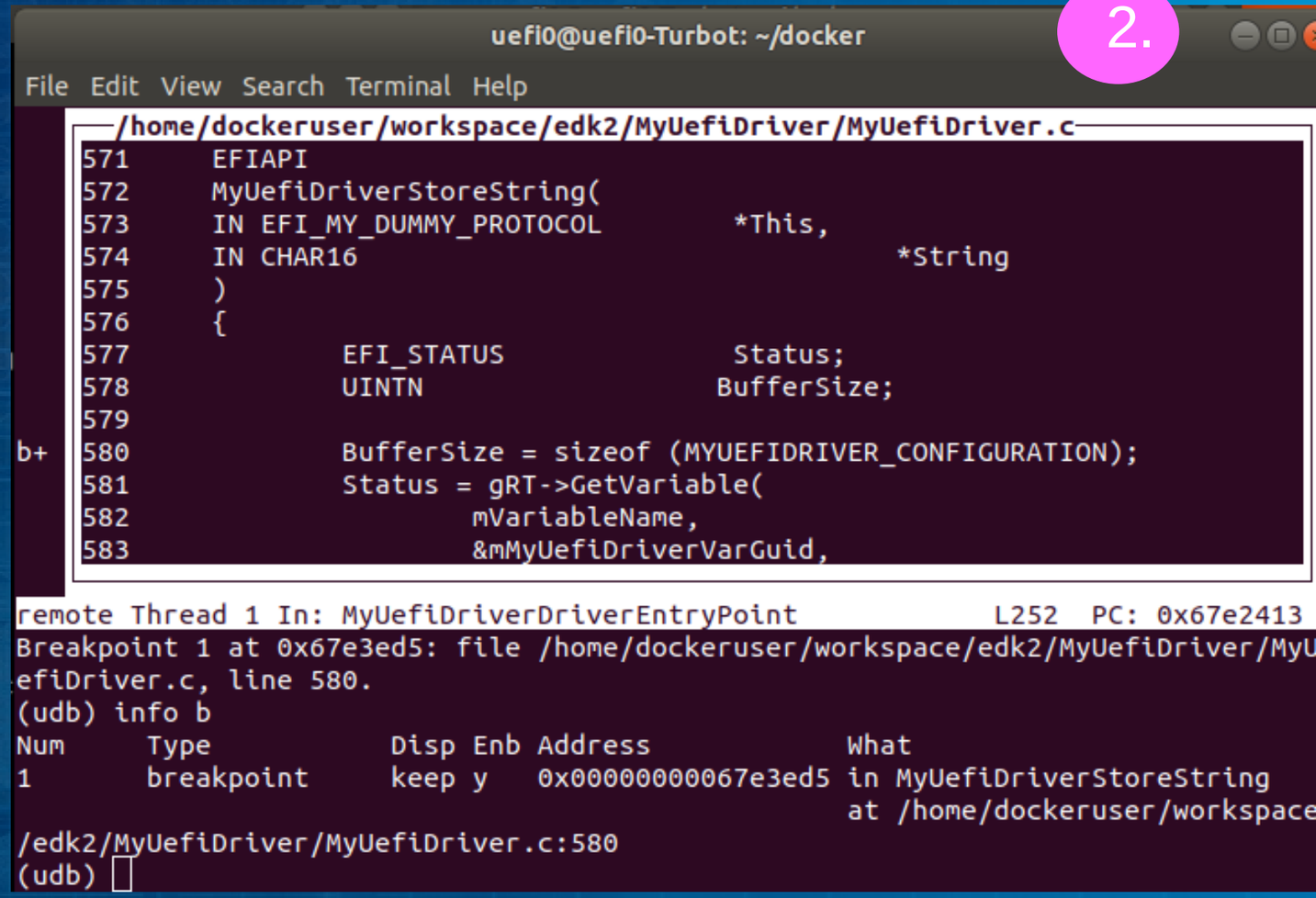
Debugging with GDB and UDK

Notice the “b+” added to the source code

This will set a breakpoint in the MyDummyProtocol for Storing a string

Continue to load the driver

```
(udb) c
```



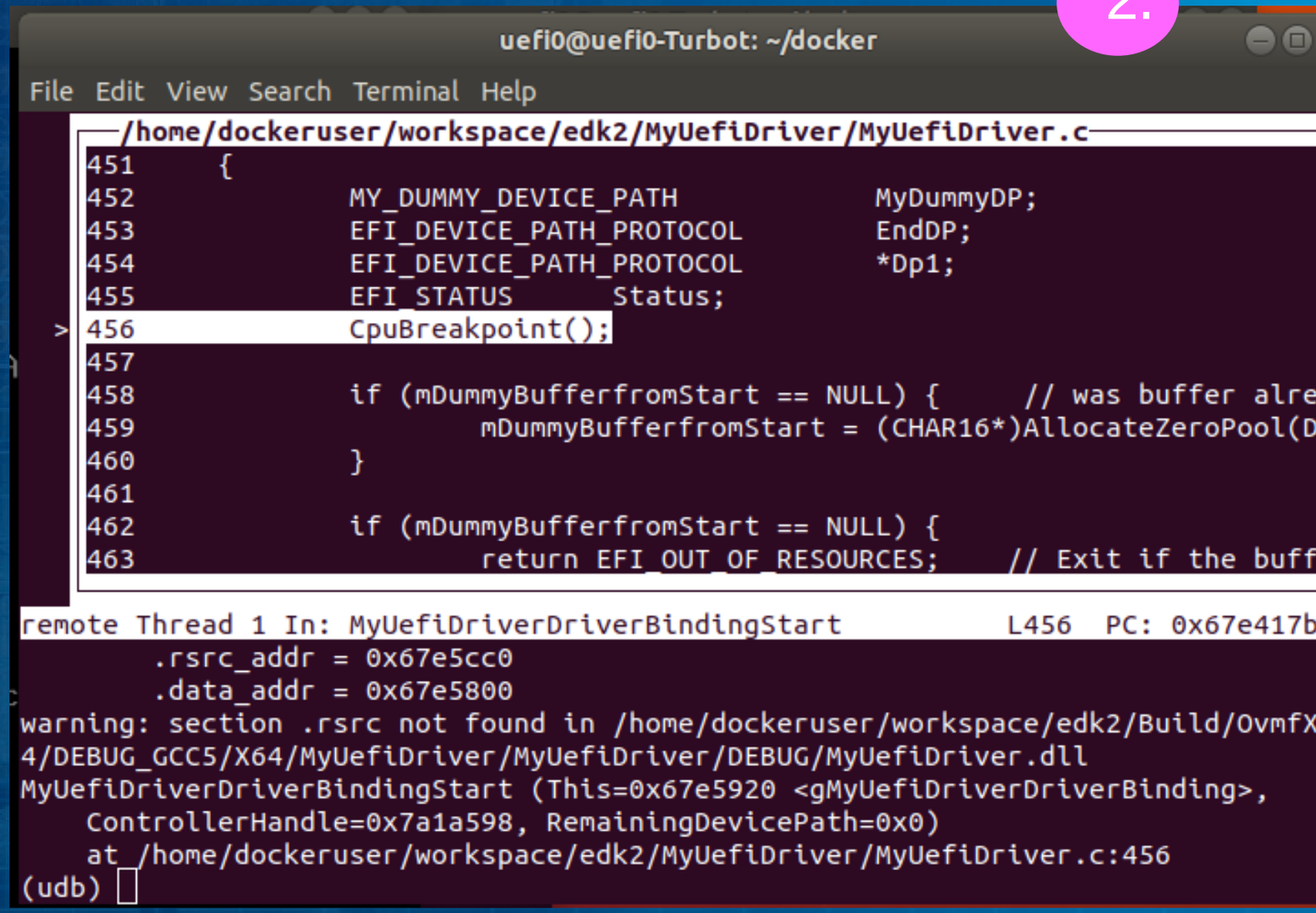
Debugging with GDB and UDK

The next CpuBreakpoint in the Start function

Step through the Start function with “next”

```
(udb) next
```

Note: use cntl-O to repeat the “next” command in (udb)



Debugging with GDB and UDK

Debug in Window 2 (gdb)

Other commands in the Debugger when a breakpoint is hit:

```
(udb) info locals
(udb) info args
(udb) info modules
(udb) backtrace
(udb) b MyApp.c:147
```

Check the PDF docs on gdb and UDK Debugger for other debug commands

Debug in Window 3 (QEMU)

At the UEFI Shell Prompt:

check out nvram variables

```
FS0:> Dmpstore –all –b
```

Mem location from debug.log

```
FS0:> Mem 0x…
```

Try other combinations of the application MyApp

```
FS0:> Myapp J “hello world This is a very long”
FS0:> Myapp  -C
```

Exit QEMU

Do not Exit Qemu if the gdb is in “Continuing.”

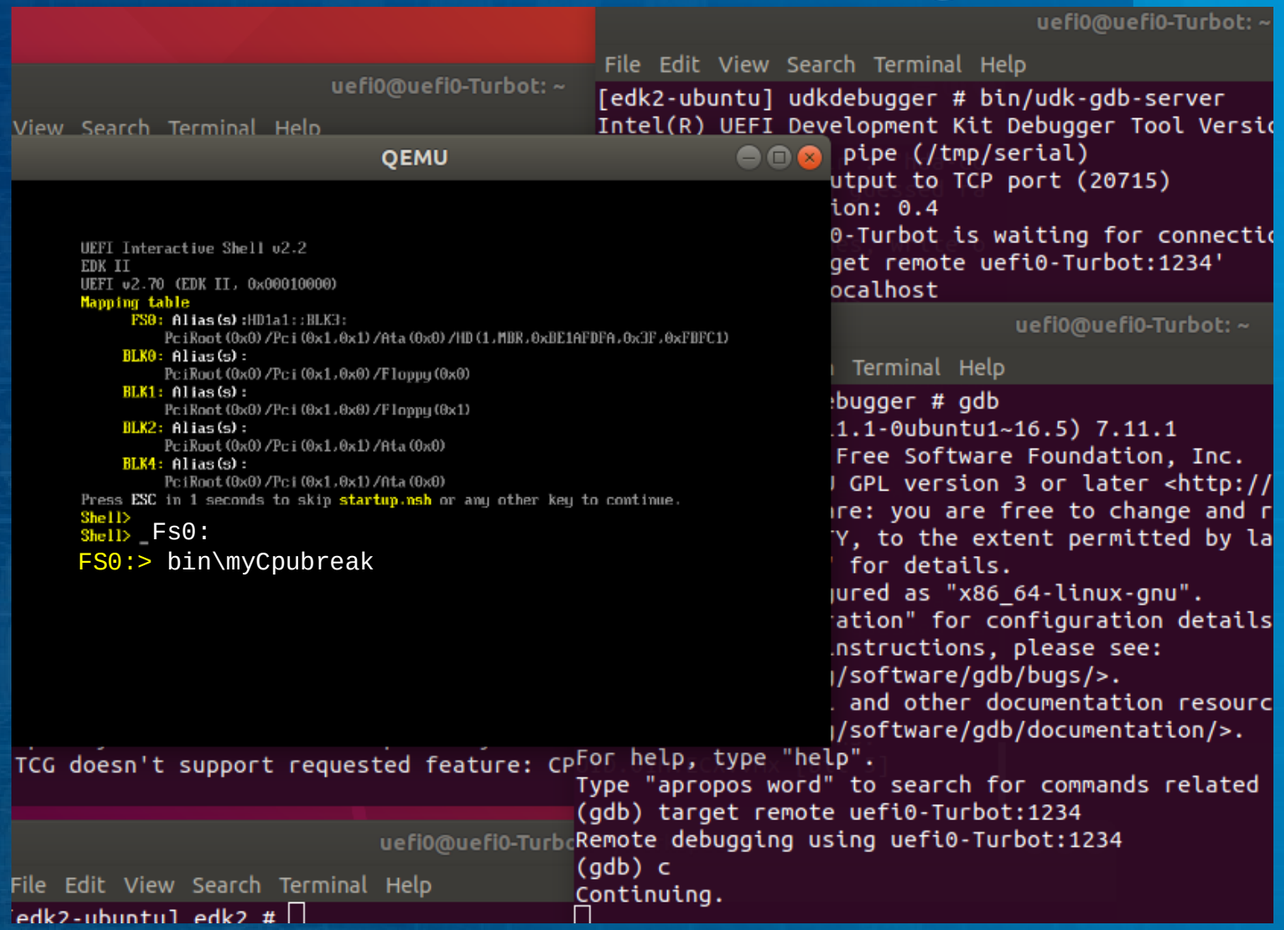
In invoke the app:

```
FS0:> bin\myCpubreak
```

This will force gdb to break at the and stop Qemu

Then Exit Qemu by closing window for QEMU

DO NOT EXIT gdb



DEBUG A REAL BUG

Debug a UEFI Driver with a Real Bug and use the Debugger tools to find the bug

Copy Simple Driver to Your Workspace

Open Directory Workshop-Material/SampleCode/UEFI-Driver

```
bash$ cd ~/workshop/Workshop-Material/SampleCode/UEFI-Driver02
```

Copy the sample driver to the edk2 directories

```
bash$ cp –R MyUefiDriver  ~/workspace/edk2/
```

Be sure that the Driver gets re-built by deleting the intermediate object files

```
bash$ rm -R ~/workshop/edk2/Build/OvmfX64/DEBUG_GCC5/X64/MyUefiDriver
```

Build UEFI Driver & Test

1. Build with Source Debugger

```
bash$ cd ~/workspace/edk2
bash$ build -D SOURCE_DEBUG_ENABLE
```

2. Change to run-ovmf directory under the home directory

```
bash$ cd ~/workspace/run-ovmf
```

3. Copy the OVMF.fd BIOS image created from the build to the run-ovmf directory naming it bios.bin

```
bash$ cp ~/workspace/edk2/Build/OvmfX64/DEBUG_GCC5/FV/OVMF.fd bios.bin
```

4. Copy MyUefiDriver and MyApp to the Qemu file system had-contents

```
bash$ cp ~/workspace/edk2/Build/OvmfX64/DEBUG_GCC5/X64/MyApp.efi hda-
contents/.
bash$ cp ~/workspace/edk2/Build/OvmfX64/DEBUG_GCC5/X64/MyUefiDriver.efi hda-
contents/.
```

Build UEFI Driver & Test

1. Run QEMU

```
bash$ . RunQemu.sh
```

2. Load the UEFI Driver at the shell prompt

```
Shell> fs0:
FS0:> Load MyUefiDriver.efi
```

3. Continue “c” in (gdb) when “CpuBreakpoint()” is hit

4. Test with the MyApp

```
FS0:> MyApp H “hello world”
```

ADD HEAP GUARD

Turn on Heap Guard

Check PCD for Heap Guard Enabling

Open edk2/OvmfPkg/OvmfPkgX64.dsc and scroll to the bottom

```
!if $(LAB_HEAPGUARD)  == TRUE
 MyUefiDriver/MyUefiDriver.inf{
 <PcdsFixedAtBuild>
  gEfiMdeModulePkgTokenSpaceGuid.PcdHeapGuardPageType|0x01e
  gEfiMdeModulePkgTokenSpaceGuid.PcdHeapGuardPoolType|0x01e
  gEfiMdeModulePkgTokenSpaceGuid.PcdHeapGuardPropertyMask|0x03
  gEfiMdeModulePkgTokenSpaceGuid.PcdCpuStackGuard|TRUE
}

 MyUefiDriver/MyApp.inf{
 <PcdsFixedAtBuild>
  gEfiMdeModulePkgTokenSpaceGuid.PcdHeapGuardPageType|0x01e
  gEfiMdeModulePkgTokenSpaceGuid.PcdHeapGuardPoolType|0x01e
  gEfiMdeModulePkgTokenSpaceGuid.PcdHeapGuardPropertyMask|0x03
  gEfiMdeModulePkgTokenSpaceGuid.PcdCpuStackGuard|TRUE
}
```

Build UEFI Driver & Test

1. Build with Source Debugger

```
bash$ cd ~/workspace/edk2
bash$ build -D SOURCE_DEBUG_ENABLE –D LAB_HEAPGUARD
```

2. Change to run-ovmf directory under the home directory

```
bash$ cd ~/workspace/run-ovmf
```

3. Copy the OVMF.fd BIOS image created from the build to the run-ovmf directory naming it bios.bin

```
bash$ cp ~/workspace/edk2/Build/OvmfX64/DEBUG_GCC5/FV/OVMF.fd bios.bin
```

4. Copy MyUefiDriver and MyApp to the Qemu file system had-contents

```
bash$ cp ~/workspace/edk2/Build/OvmfX64/DEBUG_GCC5/X64/MyApp.efi hda-
contents/.
bash$ cp ~/workspace/edk2/Build/OvmfX64/DEBUG_GCC5/X64/MyUefiDriver.efi hda-
contents/.
```

Build UEFI Driver & Test

1. Run QEMU

```
bash$ . RunQemu.sh
```

2. Load the UEFI Driver at the shell prompt

```
Shell> fs0:
FS0:> Load MyUefiDriver.efi
```

3. Continue “c” in (gdb) when “CpuBreakpoint()” is hit

4. Test with the MyApp

```
FS0:> MyApp H “hello world”
```

SUMMARY

- List the ways to debug
- Define EDK II **`DebugLib`** and its attributes
- Introduce the Intel® UEFI Development Kit Debugger (Intel® UDK Debugger)
- Debugging PI's phases
- Debug EDK II using Intel® UDK w/ GDB - LAB

Legal Notice

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

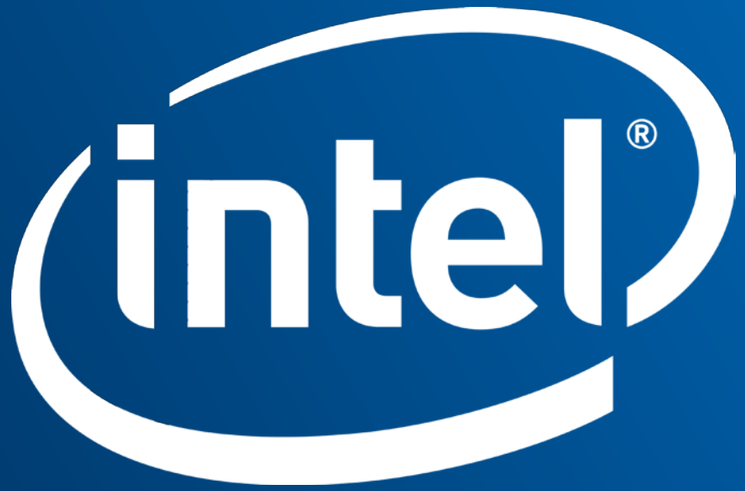
Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice.

The products and services described may contain defects or errors known as errata which may cause deviations from published specifications. Current characterized errata are available on request.

Intel, the Intel logo are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others


intel

Backup

Restarting Docker

Close all docker sessions and open native console/terminal session

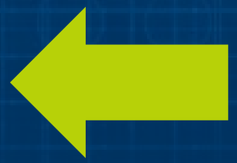
```
$> docker ps –a
$> docker rm –f edk2
$> xhost local:root
```

Re-load Docker image

```
$> docker load –i edk2-ubuntu.docker
$> cd ~/workspace
$> . ~/docker/edk-2-ubuntu.sh
```

Re-do Pipe and UDK Debugger config

```
bash$ mkfifo /tmp/serial.in
bash$ mkfifo /tmp/serial.out
bash$ cp udkdebugger.conf /etc/
```



Return to Open 4 docker terminal console windows