EDK II Remote Debug Support

Laurie Jarlstrom

Intel Corporation

Disclaimer

THIS INFORMATION COTNAINED IN THIS DOCUMENT, INCLUDING ANY TEST RESULTS ARE PROVIDED "AS IS" WITH NO WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY, NONINFRINGEMENT FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE.INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT OR BY THE SALE OF INTEL PRODUCTS. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. Intel products are not intended for use in medical, life saving, or life sustaining applications.

Intel retains the right to make changes to its specifications at any time, without notice.

Recipients of this information remain solely responsible for the design, sale and functionality of their products, including any liability arising from product infringement or product warranty.

Intel may make changes to specifications, product roadmaps and product descriptions at any time, without notice.

Intel and the Intel logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

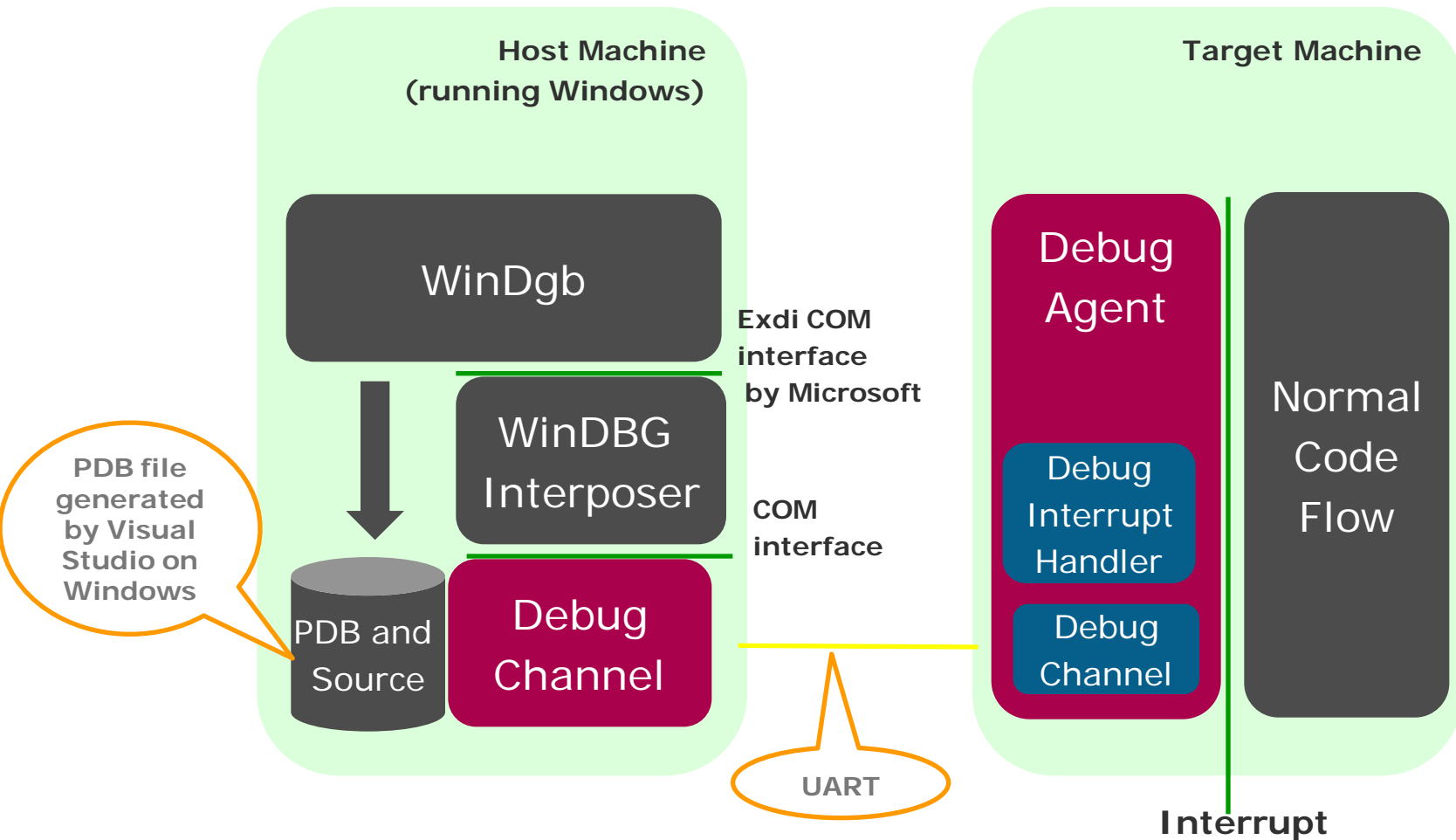*Other names and brands may be claimed as the property of others.

Agenda

- Overview
- General architecture
- Changes to the target firmware
- Debug Features
- Distribution
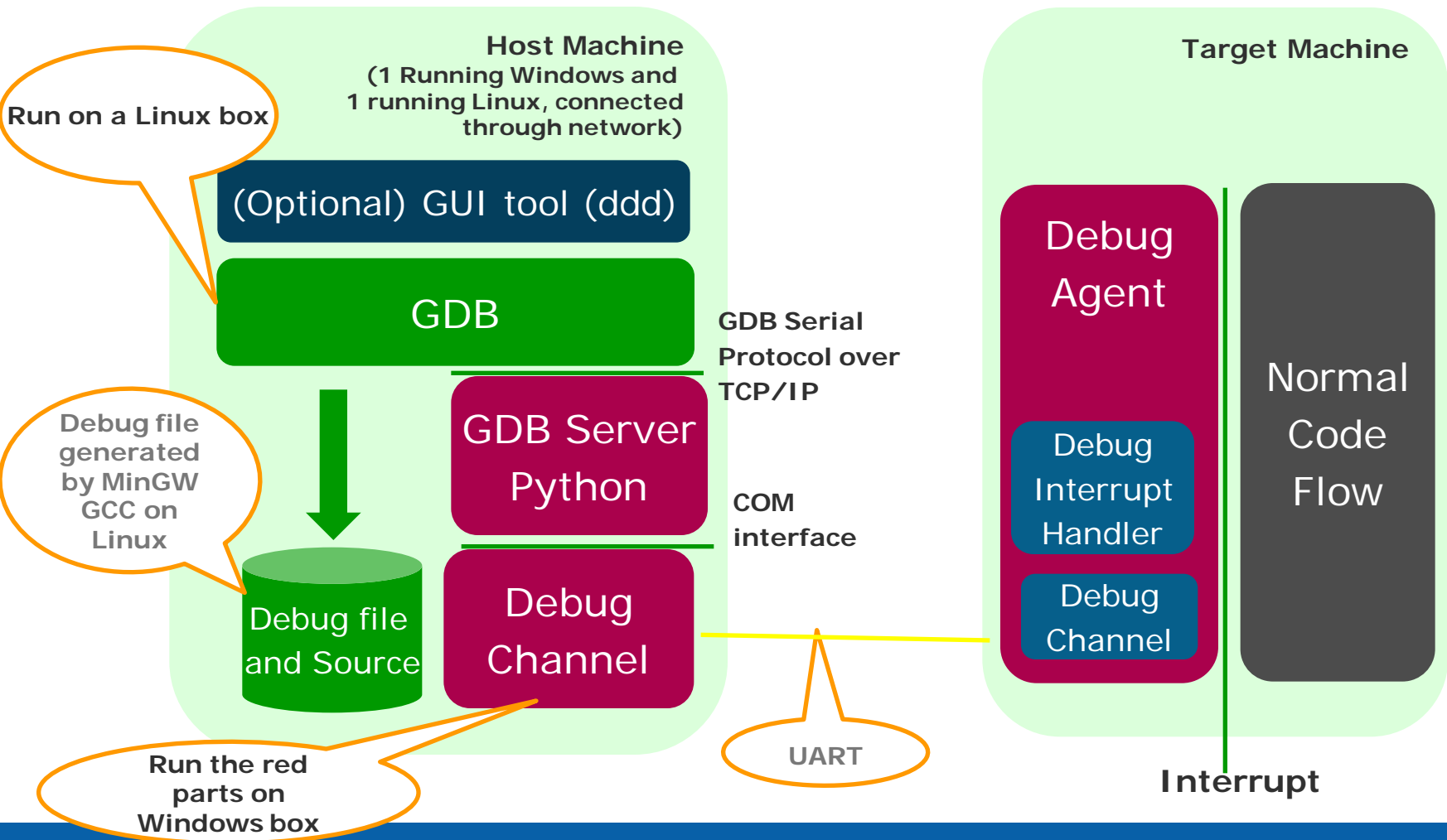- Known Limitation
- Usage Scenario

Overview

- Provide ability to support both GDB* and WinDbg* with key debug features to trace the EDK II code flow and check status (variable, registers, etc).

- Supported features
  - Use WinDbg to debug target machine which is running EDK II code
  - Use GDB to debug target machine which is running EDK II code
  - Use UART with Null modem cable connected to Host
  - Starting as early as in late SEC phase
  - Basic assembly level debug commands already supported in GDB (tried in early SEC/PEI)
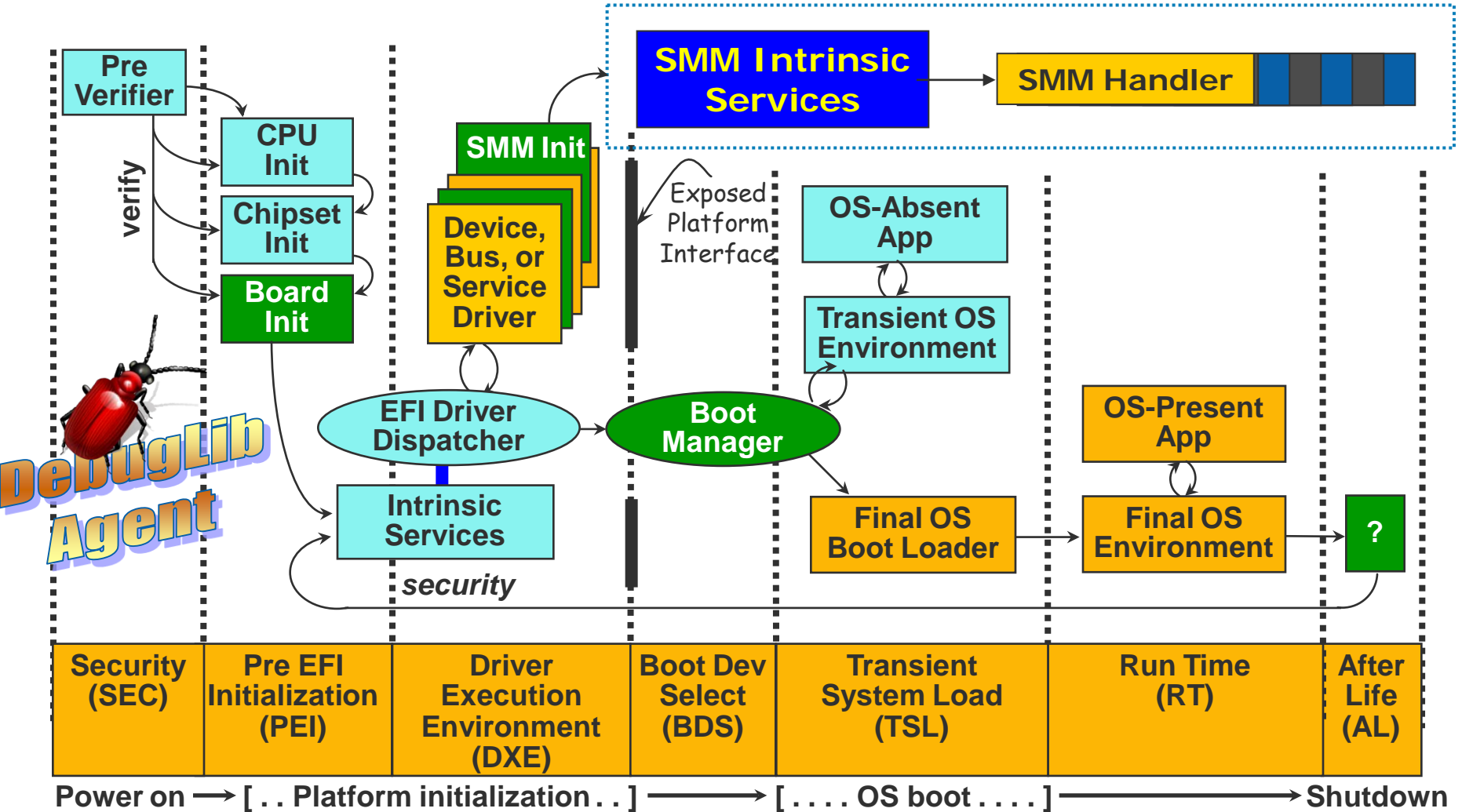
General Architecture (Windows*)

General Architecture (Linux)

Host Machine
(1 Running Windows and 1 running Linux, connected through network)

Target Machine

Run on a Linux box

(Optional) GUI tool (ddd)

GDB

GDB Serial Protocol over TCP/IP

Debug file generated by MinGW GCC on Linux

GDB Server Python

COM interface

Debug file and Source

Debug Channel

Run the red parts on Windows box

UART

Debug Agent

Debug Interrupt Handler

Debug Channel

Normal Code Flow

Interrupt

New Debug Library Agent

- New Debug Library agent installed at different phases

Changes to the Target Firmware

- Goal to minimize changes needed for target firmware

- Add a call to a new library class called the DebugAgentLib at a few key points in the boot flow. One in SEC, one in DXE Main, and another in SMM CPU Module.

- A NULL implementation of the DebugAgentLib will be checked into open source so all modules can build with debug feature disabled

Updates to DSC

Libraries   SourceLevelDebugPkg Lib Instance

[LibraryClasess] *General*

PeCoffExtraActionLib   PeCoffExtraActionLibDebug.inf

[LibraryClasses.IA32] *PEI*

DebugAgentLib   SecPeiDebugAgentLib.inf

[LibraryClasses.X64] *DXE*

DebugAgentLib   DxeDebugAgentLib.inf

[LibraryClasses.X64.DXE_SMM_DRIVER] *SMM*

DebugAgentLib   SmmDebugAgentLib.inf

Updates to FDF

[FV.FVMAIN]

. . .

DXE Phase modules

. . .

Comment out module for

TerminalDxe.inf

> #INF MdeModulePkg/Universal/Console/TerminalDxe/TerminalDxe.inf

Debug Features

- Insert CpuBreakpoint() in source code, to start debugging a module
- Source level debug
- Go/Halt/Go till
- Set breakpoint (<=3 for code running on flash)
- Step into, step over
- View and edit local variables and global variables (suggest use Disable Optimization for the compiler option)
- Call-stack (in PEI, PE image should be used to see complete call stack)
- View disassembly, view and edit general purpose register values

(intel)

Distribution

- Plan to provide single package to contain:
  - DebugAgentLib implementations
  - DebugPortLib implementations
  - binaries of the tools that run on the host
  - Documentation
  - License for Intel Tiano Direct Licenses only as non-distributable end point code.
    - >For Direct Licenses to use for Development purposes only

Known Limitations

- Do not debug debugger itself
- MSR read/write access not supported yet
- Do not support Multi Processors
- Do not support pure 32-bit platform
- Not all WinDbg commands validated yet
- Cannot set breakpoint before a module get loaded
- Do not use 2 debuggers at the same time
- Do not support 16-bit debugging
- Do not support IPF
- A small set of code is not debug-able, like early SEC, early SMM
- May have bugs or unsupported features (usually corner cases)

Usage Scenario WinDBG

- Environment
  - WinDbg 6.11.1.404 -Microsoft website http://msdl.microsoft.com/download/symbols/debuggers/dbg_x86_6.11.1.404.msi
  - Windows XP (development environment)
  - or, GDB on Linux
- Configuration
  - Host: Configure the DebugPortUart.ini for COM port used
  - Target: Configure target to use right COM (through PCD), ensure the COM is not used by other module/feature (for example, remove Terminal driver), use non-NULL DebugAgent library instance get used
    - > COM 1 is the Default on Target
    - > Simply print ASCII though that COM is allowed

Starting Debug

- Launch
  - Launch WinDbg Batch file script
  - Then power on the target (within 40s)
  - If target CPU mode switch (like 32->64 bit when PEI -> DXE), close and relaunch WinDbg
- Optional Configure the Symbol path (Alt. "F", then "S") to the Workspace the Build was invoked
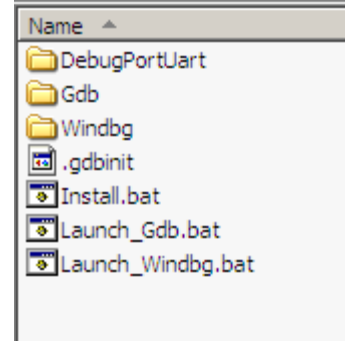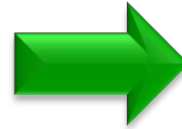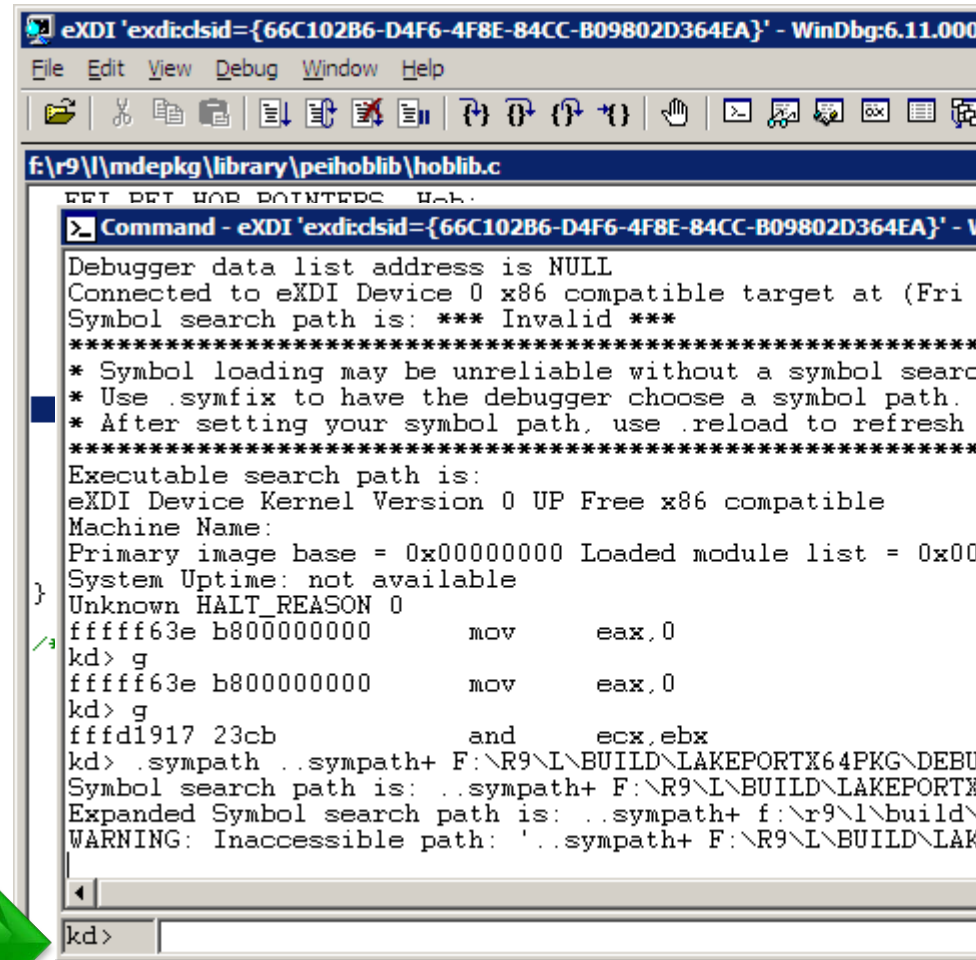
Symbol File Path
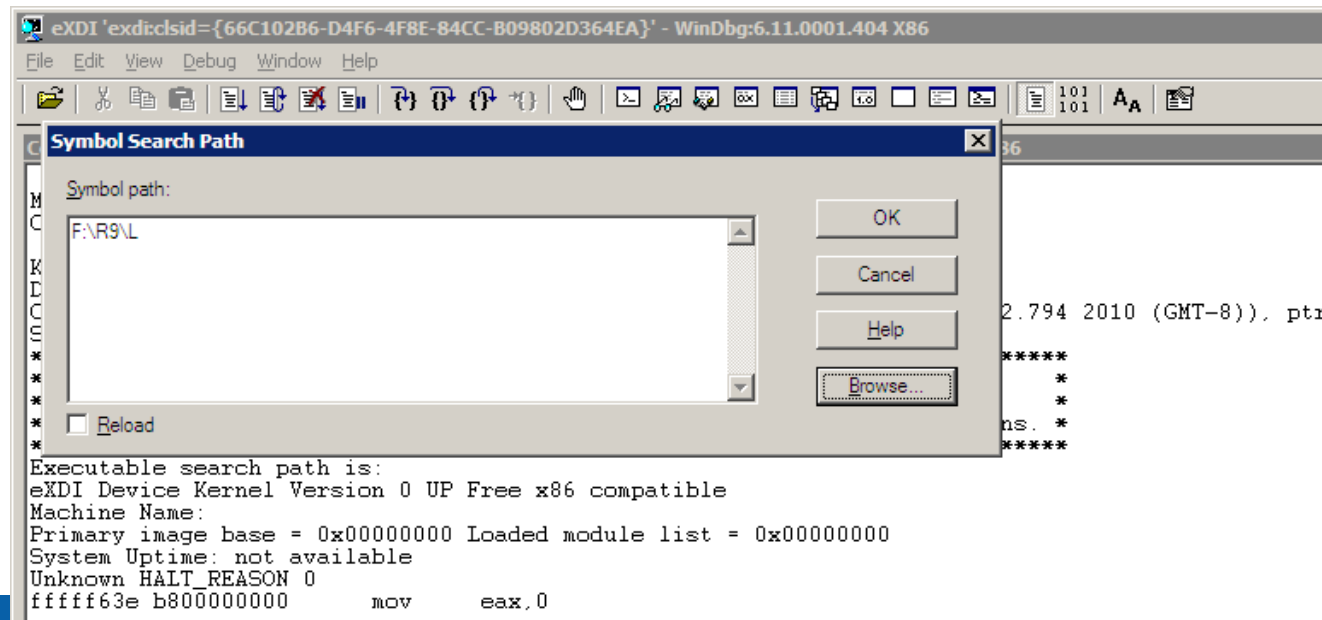
WinDBG Command window

- Command Window must be floating
- After PEI-IPL will need to exit the WinDBG and then Re-invoke. DO NOT exit the "DebugPortUart" window.
- Bottom window allows commands to be entered
  - .reboot
  - Smmentrybreak=1 or 0
  - g - Go
  - B[C|D|E][<bps>] - clear/disable/enable breakpoint(s)
  - Q - quit
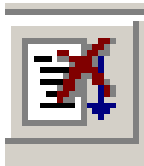 - ? – Command list

Debug Commands (WinDBG) GUI
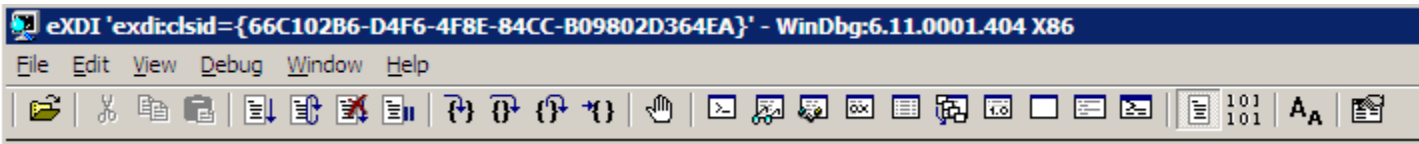
- Go – "G", "F5" or
- Halt – Control Break
- Step Into "F8" or
- Step Over "F10" or
- Step Out "Shift F11" or
- Run to Cursor
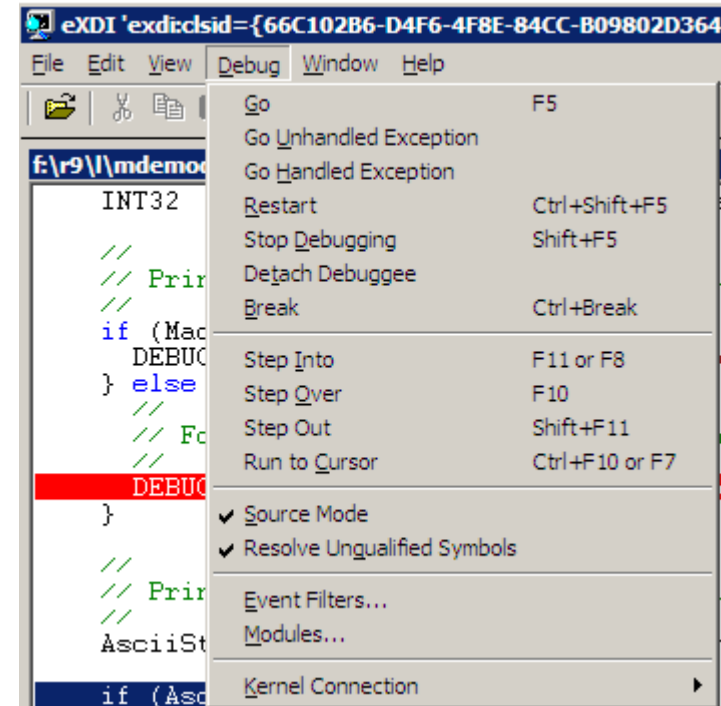
Source Code View

- "C" source code can be viewed after a "Control Break"

Setting a Break point

•Other trademarks and brands are the property of their respective owners

Suggested Breakpoints

- Debugging the Boot Phases
  - Security (SEC) Phase
  - Pre-EFI (PEI) Phase
  - DXE Phase
  - BDS Phase
  - SMM

Security Phase (SEC)

SEC
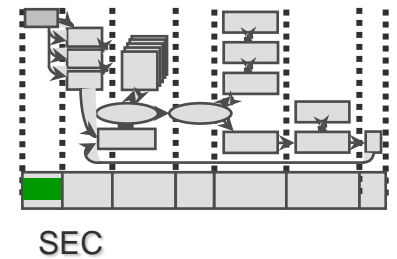
- **Function**
  - Authenticate BIOS
  - Switch to 32-bit flat mode
  - BSP selection
  - Initialize PEI temporary memory
  - Transfer control to PEI Core
- **Platform specific functions**
  - AP waking stub
  - Early microcode update
  - Common ratio programming
  - Collect BIST (Built-in Self Test)
- **Executed in place from flash**
- **Written in assembly (16-bit & 32-bit)**

Debugging Done In The SEC phase

- Checking if reset vector is accessible

- Stepping though the instructions singly

- Make certain the CPU is able to fetch the instructions from the flash and validates that the address is being decoded correctly

- Check for setting up of Cache-As-RAM (CAR)

- Switch to protected mode

- Execution of microcode patch

- Use .reboot command to reset the target

PEI Phase
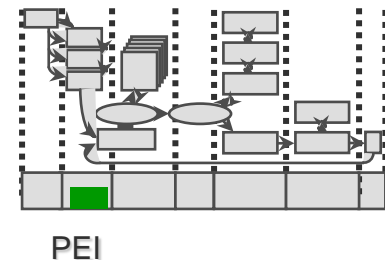
- **Function**
  - Discover and initialize some RAM that won't be reconfigured
  - Describes location of FV(s) containing DXE Core & Architecture Protocols
  - Describes other fixed, platform specific resources that only PEI can know about

- **Components**
  - **Binaries**: PEI Core and PEI Modules (PEIMs)
    - PEIMs are modules scheduled by the PEI core in the early phase of platform initialization. PEIMs are typically executed in place before system memory is available. Only hardware breakpoints can be set on PEIMS because the flash is read only and doesn't allow ITP to patch instructions in the flash.
  - **Interfaces**: Methods of Inter-PEIM communication
    - Core set of services (PeiServices), PEIM to PEIM Interfaces (PPIs), and simple Notifies (no timer in PEI)

Debugging Done In The PEI Phase
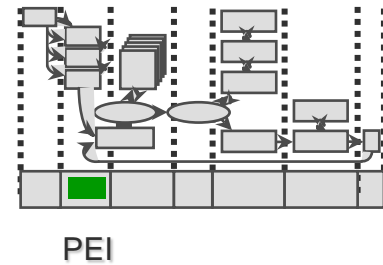
- Check for proper execution and order of all the PEI drivers

- Execution of basic chipset initialization
  - GMCH/Uncore, ICH/PCH, SIO device initialization

- Execution of memory init instruction

- Availability of memory, and complete flash accessibility

- Execution of recovery driver if the recovery jumper is selected, and execution of recovery path if recovery is detected

- Detection of DXE IPL PEIM which in turn detects and launches the DXE core

PEI Phase - Trace each PEIM

- Location
  - **File**: MdeModulePkg\Core\Pei\Dispatcher\Dispatcher.c
  - **Function**: PeiDispatcher()
  - **For Loop**

- Trace all the PEIMs being dispatched load the Dispatcher.c file in PEIMAIM module

- Scroll down to PeiDispatcher() function and set a break point at the main dispatch loop before each PEIM Entry

```
// Call the PEIM entry point

CpuBreakpoint() ;

PeimEntryPoint(PeimFileHandle, (const EFI_PEI_SERVICES **) &Private->Ps);
```

- The next time you hit this breakpoint, you can step into this function to trace each PEIMs being dispatched.

DXE Phase

DXE
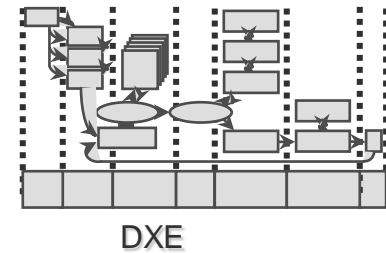
- Works after system memory has been discovered and initialized

- DXE drivers are typically stored in flash in compressed form and must be decompressed into memory before execution

- Both hardware and software breakpoints can be set in DXE drivers

Debugging Done In The DXE Phase
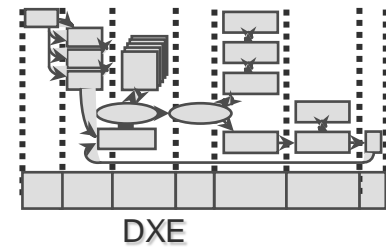
DXE

- Cyclic dependency check

- Tracing any assert that may have been caused during DXE execution

- Debugging of individual DXE driver

- Check for failure to load architectural protocols

- Check to see if BDS entry has been called

Break point at DXE-Phase Entry Point

- Check if PEI-phase reaches DXE-phase
  - Location
    - >**File**: MdeModulePkg\Core\Pei\PeiMain\PeiMain.c
    - >**Function**: PeiCore()
    - >**Call**: DxeIpl->Entry()

```
// Enter DxeIpl to load Dxe core.
//
 CpuBreakpoint() ;
 Status = TempPtr.DxeIpl->Entry (
                           TempPtr.DxeIpl,
                           &PrivateData.Ps,
                           PrivateData.HobList
                           );return EFI_NOT_FOUND;
```

Break point at DXE-Phase Entry Point – part 2

- Verify the address of DXE Core Entry point after IPL from PEI
- Check if we pass behind HandOffToDxeCore call
- Location
  - **File**:MdeModulePkg\Core\DxeIplPeim\DxeLoad.c
  - **Function**: DxeLoadCore (inside the call DxeIpl->Entry())
  - **Call**: HandOffToDxeCore()
    - >**Argument**: DxeCoreEntryPoint

```
// Transfer control to the DXE Core
// The hand off state is simply a pointer to the HOB list
//
  CpuBreakpoint() ;
  HandOffToDxeCore (DxeCoreEntryPoint, HobList);
//
// If we get here, then the DXE Core returned.  This is an error
```

(intel)

DXE – Trace each Driver Load

- Check if control has been transferred to loaded image entry points
  - The system breaks at this point successfully every time a new DXE driver is loaded. Step into this function to trace individual drivers.
  - Location
    - >**File**: MdeModulePkg\Core\Dxe\Image\Image.c
    - >**Function**: CoreStartImage
    - >**Call**: Image->EntryPoint()

```
Image->Started = TRUE;
CpuBreakpoint() ;
Image->Status=Image->EntryPoint (ImageHandle, Image->Info.SystemTable);
```

BDS Phase
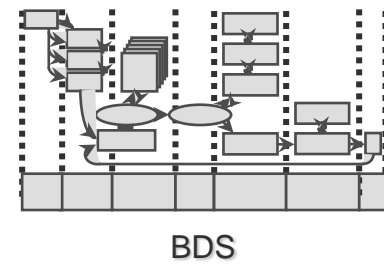
- Centralize Policy and User Interface
  - Lets you customize to different look and feels
- Make a central repository for platform boot policy
- Allow for the ability to boot with minimal driver initialization and user interaction
- Allow for implementation of setup menu
- Allow for ability to store information using NVRAM variables.

Debugging Done In The BDS Phase

BDS
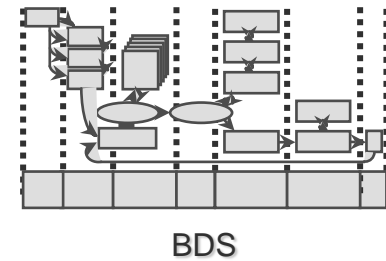
- Ensuring detection of console devices (both input and output)

- Ensuring complete enumeration of all the devices preset (for which the BIOS has the drivers)

- Detection of Boot policy

- Loading of BIOS front page

Debugging BDS-Phase Entry Pont

- Check if you reached and entered the BDS-phase
    - Location:
        - >**File**: MdeModulePkg\Core\Dxe\DxeMain DxeMain.c
        - >**Function**: DxeMain
        - >**Call**: gBds->Entry (gBds);

```
CpuBreakpoint() ;
gBds->Entry (gBds);
// BDS should never return
ASSERT (FALSE);
CpuDeadLoop ();
```

System Management Mode

- Registration vehicle for dispatching drivers in response to System Management Interrupts (SMI)

- Dispatch of drivers in System Management Mode (SMM) will not be able to use core protocol services

- SMM handlers will be logically prevented from accessing conventional memory resources

- SmmLib includes a subset of the DXE core services, such as memory allocation, device I/O protocol, and others

Debugging Done In The System Management Mode

- SMM drivers are a special type of DXE drivers. As with other DXE drivers, SMM drivers are scheduled by the DXE core, but SMM drivers perform the following steps in the entry point:
  - Locate the SmmBase protocol.
  - Invoke SmmBase.InSmm() to see whether the driver is in SMM. If yes, proceed to other initialization relevant to this driver, like what a DXE driver does, and return EFI_SUCCESS. If the driver is not in SMM, proceed with the following steps.
  - Invoke SmmBase.Register() to fork another copy of the SMM driver in SMRAM. At this point, two copies of this driver exist: one in BS memory and the other in SMRAM.
  - The copy of the driver in BS memory returns an error code to make DXE core release the memory occupied by this copy.
- SMM drivers are not as straightforward as DXE drivers, because the processor automatically cleans up debug registers when it enters SMM. Set **smmentrybreak=1**

(intel)

Debugging Done In The System Management Mode

```
SmmBase->InSmm (SmmBase, &InSmm);
if (!InSmm) {
    // Retrieve the Device Path Protocol from the DeviceHandle that this driver was loaded from
    Status = mBS->HandleProtocol (LoadedImage->DeviceHandle,
            &gEfiDevicePathProtocolGuid,
            (VOID*)&ImageDevicePath);
    ASSERT_EFI_ERROR (Status);
    // Build the full device path to the currently executing image
    CompleteFilePath = SmmAppendDevicePath (ImageDevicePath, LoadedImage->FilePath);
    // Load the image in memory to SMRAM; it will automatically generate the SMI.
    Status = SmmBase->Register (SmmBase, CompleteFilePath, NULL, 0, &Handle, FALSE);
    ASSERT_EFI_ERROR (Status);
    return Status;
}
Status = mBS->HandleProtocol ( ImageHandle, &gEfiLoadedImageProtocolGuid, (VOID
    **)&LoadedImage);
ASSERT_EFI_ERROR (Status);
LoadedImage->Unload = _DriverUnloadHandler;
// Skipped…
return Status;
```

Debugging Done In The System Management Mode For Platform Initialization (PI) Spec

SMM Initialization - Load the SMM Core image into SMRAM and execute the SMM Core from SMRAM

- Location:
  - >**File**: MdeModulePkg\Core\PiSmmCore\PiSmmIpl.c
  - >**Function**: SmmIplEntry
  - >**Call**: **ExecuteSmmCoreFromSmram**

```
// Load SMM Core into SMRAM and execute it from SMRAM
  //
  Status = ExecuteSmmCoreFromSmram (mCurrentSmramRange, gSmmCorePrivate);
```

(intel)

Backup