Debugging Firmware Based on The Unified Extensible Firmware Interface

Every software developer knows that debugging is more difficult than coding. Debugging is one of the most common tasks in product development and maintenance. In each phase of software engineering, engineers need debug tools. For UEFI firmware, this is an even larger problem. An environment with rich debug capability will speed up development for UEFI firmware, but specialized tools are often required.

This article provides an overview of common debug solutions including hardware based debugging, system checkpoints, and source-level debugging. Firmware specific concepts such as status codes, DEBUG/ASSERT macros, and the UEFI debug protocol are introduced. This article also demonstrates source-level debugging support using AMI and Intel solutions, comparing them to hardware-based alternatives in various scenarios.

Common Debug Scenarios in UEFI
This section describes the various aspects of debugging within a UEFI environment as well as the resources available to the developer during debug.

Hardware-Based Debugging

JTAG-style connectors such as the Intel XDP debug port allow complete software execution control and direct hardware inspection. JTAG is widely used for early silicon debugging on Intel platforms, especially when the system is not stable. We can use it to configure the system, fuse registers before boot, add workarounds, view the system state even when the system is stuck and the CPU is out of control at that time.

However, JTAG-style connectors are not commonly available on production hardware. This can limit debugging on systems in production, especially if physical intrusion is not possible. There are also scenarios when only a subset of JTAG capabilities is required to debug a firmware issue, so the cost of a hardware-based debugger is not justified.
**System Checkpoints**

A checkpoint is a hexadecimal value sent to I/O port 0x80. This checkpoint system was developed in legacy BIOS implementations and is still used in many UEFI implementations. The BIOS outputs checkpoints through PEI, DXE, and BDS to indicate the task the system is currently executing. Checkpoints are very useful in aiding software developers or technicians in debugging problems that occur during the pre-boot process, since they are available before the local display is initialized.

For BIOS developers, field technicians, and quality assurance technicians, the POST checkpoints are like the diagnostic codes used in today’s automotive computers. When the “check engine light” comes on, customers expect a mechanic to read the engine code and diagnose the problem. Checkpoints expose the same feature in the BIOS, providing debug information before the OS boots.

The use of checkpoint codes for firmware debugging is limited by a number of factors:

- It requires access inside the system enclosure, which is not possible on some server, embedded, and mobile applications.

- It requires a PCI slot, which is no longer standard on newer computer platforms. Today’s systems use PCI Express (PCIe) for internal expansion slots and Universal Serial Bus (USB) for external devices. PCIe and USB are preferable to system vendors due to the reduced size of the expansion slot.

- If all of the system PCI slots are populated, then the hardware configuration has to be changed to use the PCI Checkpoint Card. This may change the problem that the technician or developer is trying to debug.

- Technicians must use additional documentation to translate the hexadecimal checkpoint into useful information, typically via tables in BIOS documentation.

- Checkpoint cards do not store any history of the checkpoints from any boot session, so checkpoint information must be manually recorded during testing.

Opening a consumer desktop system to insert a PCI-based POST
Checkpoint Card is simple, but the same basic diagnostic on a tablet or embedded system is difficult. Computers in point of sale (POS), gaming, digital signage, and rugged computing applications are not designed to be easily opened, which makes system diagnostics difficult if an add-in card has to be installed.

Checkpoints are also limited by the I/O port implementation, restricting checkpoints to 8-bit or 16-bit values that are specific to the firmware vendor’s implementation. This legacy interface could be replaced with a larger bit field value over an interface-agnostic implementation.

**Extended Debug Information**

System checkpoints give us information, but the information is limited, since the use of checkpoints is just a simple method to let us know where we are.

UEFI debug strings are typically disabled on production firmware, but are often enabled on evaluation platforms and test firmware builds. When enabled, UEFI debug strings provide verbose details throughout the boot process such as driver entry notices, GUID for published protocols, and special messages reserved for code compiled in debug mode.

This example output comes from a memory detection routine in PEI:

```
[AmiDbg]MemDetect.Entry(FFFF6582)
[AmiDbg]Memory Installed: Address51C700000; Length53000000
[AmiDbg]PEI_STACK: Address51C700000; Length5100000
[AmiDbg]HOBLIST address before memory init 5 0xfe00400
[AmiDbg]HOBLIST address after memory init 5 0x1c800000
[AmiDbg]PEI core reallocated to memory
[AmiDbg]CAR stack ever used: 3580 bytes.
[AmiDbg]CAR heap used: 3016 bytes.
[AmiDbg]Notify: PPI Guid: f894643d-c449-42d1-8ea8-85bde8c65bde,
Peim notify entry point: fffbd9a9f
[AmiDbg]Notify: PPI Guid: 36164812-a023-44e5-bd85-05bf3c7700aa,
Peim notify entry point: ffffe7810
[AmiDbg]Capsule.Entry(1F6F0A5E)
[AmiDbg]Capsule Read variable service installed
```

In many cases, UEFI debug strings provide enough advanced
information to resolve firmware issues. The problem for developers is finding a simple way to view and capture the strings. UEFI debug strings were primarily designed for systems with RS-232 serial ports, which are not commonplace on modern computer systems.

**Source-Level Debugging**

JTAG is very powerful, and most JTAG software provides the capability for source-level debugging, if the image has a debug section, for instance PE/COFF PDB information and ELF Dwarf information. That helps firmware developers use tools commonly employed in software development.

Figure 1 is a source code window of the ITP/JTAG debugger. C code of the firmware can be displayed in source code window. The execution stops at line 374, and there is a breakpoint at line 384.

![Figure 1: Example of a source level debug screen (source: Intel corporation, 2011)](image)

Figure 2 shows a processor registers window. It displays processor registers. Register values can be changed directly in this window.
UEFI Solutions for Pre-Boot Debugging

This section describes some of the data elements one would deal with when debugging software within UEFI.

UEFI Status Codes

Appendix D of the UEFI Specification defines a series of Status Codes (EFI_STATUS) used by UEFI interfaces to indicate successes, errors, and warnings. These codes expand on the concept of checkpoints defined by legacy BIOS, with standardized values and use of larger bit fields. They are summarized in Table 1.
<table>
<thead>
<tr>
<th>Supported 32-bit Range</th>
<th>Supported 64-bit Architecture Ranges</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000-0x1fffffff</td>
<td>0x0000000000000000-0xfffffffffffff</td>
<td>Success and warning codes reserved for use by UEFI main specification.</td>
</tr>
<tr>
<td>0x20000000-0x3fffffff</td>
<td>0x2000000000000000-0xfffffffffffff</td>
<td>Success and warning codes reserved for use by the Platform Initialization Architecture Specification.</td>
</tr>
<tr>
<td>0x80000000-0x9fffffff</td>
<td>0x8000000000000000-0xfffffffffffff</td>
<td>Error codes reserved for use by UEFI main specification.</td>
</tr>
<tr>
<td>0xa0000000-0xbfffffff</td>
<td>0xa000000000000000-0xbffffffffffff</td>
<td>Error codes reserved for use by the Platform Initialization Architecture Specification.</td>
</tr>
</tbody>
</table>

| Table 1: UEFI status codes (source: UEFI specification 2.3.1) |

Status codes are not restricted to a specific hardware interface, such as ISA or PCI as used by legacy checkpoints. The UEFI Platform Interface (PI) specification provides Status Code Services and code definitions for the PPI and protocols used in a Report Status Code Router.

**DEBUG() and ASSERT() Macros**

DEBUG and ASSERT are very useful for developers who want to debug without using source level tools. These macros produce human readable debug output that can directed to a standard serial port or USB debug port. These macros can be enabled or disabled during firmware build. Figure 4 is an example of DEBUG() output from a UEFI system.
Chapter 17 of the UEFI 2.3.1 specification describes the UEFI debug architecture, a minimal set of protocols and data structures used to enable sourcelevel debugging. UEFI debug support is presented as a pair of protocol interfaces:

- **Debug Support** – this protocol abstracts the processor’s debugging facilities, namely a mechanism to manage the processor’s context via caller-installable exception handlers.
- **Debug Port** – this protocol abstracts the device that is used for communication between the host and target. Typically this will be a serial stream style interface.

The best example of UEFI Debug Protocol implementation uses the USB Debug Port. USB-based debug solutions function thanks to the USB 2.0 Debug Port. The debug port is a function of an EHCl USB 2.0 host controller, but is implemented by most Intel peripheral controller hubs. The debug port uses a simplified USB protocol that does not require a full memory stack, unlike the standard USB protocol. This allows the debug port to be initialized in the SEC or early PEI phase, almost immediately after the firmware gets control.

USB-based debug solutions offer the most flexibility for the platform developer and field technician:

- **Externally accessible** – USB ports are designed for external expansion, so technicians don’t have to open the case to connect the device
- **USB 2.0 enables early debugging** – accessible via the USB EHCl debug port
- **No additional hardware cost** – use the same USB port for debugging devices or with standard USB 1.1 and USB 2.0 devices
- **USB is ubiquitous** – users expect USB to be enabled on today’s systems

Since debug firmware contains extra strings and information for synchronizing the host and target systems, the UEFI debug protocol is not enabled by default.
UEFI firmware and drivers must be compiled in “debug mode” to enable support for the UEFI debug protocol. This allows debug capabilities to be quickly disabled between builds, allowing non-debug firmware to be properly optimized before release.

The UEFI specification only describes low-level interfaces, not the entire debug architecture. Different solutions take advantage of the UEFI debug protocols, either for simple debug output or full source-level debugging at the firmware level. This allows the developer to choose from a variety of tools.

**Practical Examples of UEFI Debug Solutions**

This section covers several examples of debugger interfaces and tools that a user can experience both from open source repositories as well as AMI.

**Intel® UEFI Development Kit (Intel UDK) Debugger Tool**

The Intel® UDK Debugger Tool provides the ability to debug UDK-based firmware running on an IA-32 family processor through a simple debug cable (serial or USB).

In conjunction with the Microsoft Windows* Debug Tool (WinDbg) and Linux GDB, the Intel UDK Debugger Tool provides the ability to debug UDK-based firmware on UEFI IA-32 and UEFI x64 platforms. The target side “debug agent” (SourceLevelDebugPkg) is part of the EDK II project used to create UDK2010.

Figure 5 shows the WinDbg window. It can show the C source code, and the execution is stopped at the highlighted line.

![Figure 5: snapshot of the winDbg interface with which the open source software debug package interacts (source: Intel corporation, 2011)](image-url)
Figure 6 shows the Linux GDB execution; it demonstrates the calling of PeiServicesAllocatePages.

Figure 6: snapshot of the gdb interface with which the open source software debug package interacts (source: Intel corporation, 2011)

**AMIDebug Rx**

AMIDebug Rx*, shown in Figure 7, is the first of its kind: a low-cost debug tool built around the debug port feature common to today’s USB 2.0 EHCI controllers. AMI Debug Rx is designed as replacement for the PCI POST Checkpoint Card, and also serves as a host-to-target interface for AMIDebug for UEFI and the Intel UDK 2010 firmware debug solutions.
Figure 7: snapshot of the AMI DebugRx device (source: American Megatrends Inc., 2011)

Key features:

- USB-based replacement for the PCI port 80h “POST Checkpoint” card
- Checkpoints can be captured and stored to one of four “sessions” for later review
- Measures elapsed time between checkpoints to analyze boot performance timing
- Display descriptive text for each checkpoint, based on built-in string table or custom table
- USB Virtual COM (VCOM) port for data transfer and configuration
- UEFI debug messages redirected over USB VCOM or saved in device memory

Diagnosing small form factor platforms with AMI Debug Rx is nonintrusive, allowing technicians to see checkpoints without opening the case. AMI Debug Rx replaces the POST checkpoint card’s LED display with an easy-to-read LCD screen. This debug method produces more descriptive debugging messages than the checkpoint card, along with extended features such as boot speed timing and UEFI debug
message redirection.

**AMIDebug for UEFI**

AMIDebug for UEFI is a powerful debug solution for Aptio, AMI’s product solution for UEFI firmware. AMIDebug for UEFI uses the UEFI debug protocols to provide an alternative to ITP/JTAG debugging for IA-32 and x64 firmware. Developers have access to source-level debugging and control the debug target hardware through the Visual eBIOS (VeB) development interface, shown in Figure 8.

AMIDebug for UEFI provides functionality similar to hardware-based development tools:

- Source-level symbolic debugging
- Access to hardware resources (CPU registers, PCI configuration space, memory, and I/O locations)
- Debug Aptio firmware, UEFI/DXE drivers, PEIMs, and pre-boot applications in the UEFI Shell

AMIDebug for UEFI and the UDK 2010 debugger have many common features, but AMIDebug is an example of a commercial software debugger with additional features.
The AMIDebug interface is integrated with Visual eBIOS (VeB), a development environment specifically built for developing BIOS and UEFI firmware. This allows debugging and development in the same interface, which speeds up issue resolution. AMIDebug extensions for VeB also incorporate extended debug information, such as checkpoints or debug messages, which normally appear in secondary redirection consoles. AMIDebug for UEFI also adds specialized firmware debug interfaces and can be extended to debug System Management Mode (SMM) routines on IA32 and x64 processors.

**Firmware Debugging in UDK 2010**

This section covers the various aspects of debugging code within the UDK codebase along with the pertinent architectural phases within which such debugging might take place.

**General Architecture**

Debugging UDK-based firmware requires two machines: a target and a host. The target contains the UDK firmware to be debugged and the
host executes the debug interface software (Figure 9).

![Diagram](image)

**Figure 9:** cable connection between the target and host machines (source: Intel corporation, 2011)

The architecture of WinDbg debugging support is depicted in Figure 10.

![Diagram](image)

**Figure 10:** general architecture (windows tool chain) (source: Intel corporation, 2011)
The host’s responsibility:

- Communicate with target for debug
- Implement SoftDebugger APIs
- Communicate with debug front end (WinDbg, Gdb, and so on)
- Read a configuration file to accept user configuration

The target’s responsibility:

- Respond to host’s request to set hardware breakpoint in debug registers in processor
- Handle interrupt (INT1, INT3, and so on) and give response to host, and wait for next step from host

Figure 11 shows the architecture of using GDB to debug UDK-based firmware compiled with GCC in Linux.

**Figure 11:** general architecture (linux/gcc tool chain) (source: Intel corporation, 2011)
**WinDbg Debug**

A UEFI firmware developer can use the Microsoft Windows Debug Tool (WinDbg) and the Intel UEFI Development Kit Debugger Tool (Intel UDK Debugger Tool) to debug UDK-based firmware on UEFI IA-32 and UEFI x64 platforms. The target side component (debug agent) is “SourceLevelDebugPkg” in UDK 2010.

The host is a Microsoft Windows XP (SP3) platform executing WinDbg and the Intel UDK Debugger Tool. The target and host interconnect via a serial null modem or USB 2.0 debug cable (as shown in Figure 9).

The Intel UDK Debugger Tool supports the following:

- Source-level debugging using Microsoft Windows Debug Tool (WinDbg) – host running Microsoft Windows XP (SP3)
- Debugging as early as late SEC (after temporary RAM setup) for the normal boot path
- Starting debugging SMM code by requesting target to stop at next SMI
- Cable interconnect: serial null modem cable or USB host to host cable (USB 2.0 debug device cable)
- Setting unresolved breakpoints

The basic debugging flow includes three major steps: compiling, programming, and launching.

1. Compile the firmware that includes the target side debug agent, as depicted in Figure 12.

![Figure 12: Compiling firmware image with debug agent (Source: Intel Corporation, 2011)](image-url)
2. Program the firmware image into flash on the target system.

3. Launch a debugger on the host to debug the firmware on the target system.

Figure 13 shows how the components interact during a debug session in Windbg debugging.

Figure 13: Debug session active components (Source: Intel Corporation, 2011)

**Debugging SEC and PEI Code**
Most of the code for SEC and PEI phase executes from read-only memory. The Intel UDK Debugging Tool automatically uses a hardware breakpoint if it detects the address is within the read-only memory flash range.

**Debugging DXE Code**
Some PI firmware implementations execute SEC/PEI in 32-bit mode and DXE/SMM in 64-bit mode. When the UEFI Debugger Tool detects a mode switch from either 32-bit mode to 64-bit mode (or from 64-bit
mode to 32-bit mode), the Microsoft Windows Debug Tool (WinDbg) is automatically re-launched.

**Debugging SMM Code**

The `smmentrybreak` command must be used to set a flag so the next entry into SMM will force the target to break into the debugger. The `smmentrybreak` switch must be set to inform the target whether it should stop the next time it enters SMM mode. Breakpoints can be set after the target enters SMM mode and debugging can continue. When the target stops at the SMM entry, the source for SMM handlers may be opened, and software breakpoints may be set.

**Comparing Hardware and Software Debug Capabilities**

Debuggers are critical tools for the development of software. Intel offers hardware debug solutions based on the ITP/JTAG model. Other solutions exist in the marketplace for hardware-level debugging, such as the WindRiver* ICE 2 JTAG Debugger. Software debugger solutions also exist, such as the Intel UEFI Development Kit Debugger Tool or the AMIDebug for UEFI solution that integrates into the Aptio Visual eBIOS (VeB) development environment.

Table 2 illustrates differences between hardware- and software-based firmware debug solutions.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Hardware Debugger</th>
<th>Software Debugger</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Dependency</strong></td>
<td>Requires platform with JTAG (Joint Test Action Group) or TAP (Test Access Port) interface</td>
<td>Requires support for a low-level debug port (serial, USB debug port, 1394)</td>
</tr>
<tr>
<td><strong>Special Features</strong></td>
<td>Hardware event. (Like init, reset, smmentry/smmexit) SMM/UEFI runtime service debug in OS environment. SEC phase debug. Debug mode switch (like 16-bit legacy code, 32/64 switch code) Debug software debugger agent.</td>
<td>Software logic. Condition breakpoint (break if condition is satisfied). Unresolved breakpoint (Set breakpoint on an unloaded module). View data structures. EBC (EFI byte code) debugger.</td>
</tr>
<tr>
<td>Usage Scenarios</td>
<td>Some special features, like firmware update.</td>
<td>Debug UEFI driver and option ROM (OPROM). View UEFI debug strings, BIOS checkpoints and UEFI status codes.</td>
</tr>
<tr>
<td>----------------</td>
<td>--------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Cost</td>
<td>More costly than software-based solutions</td>
<td>Some tools are open source and free.</td>
</tr>
</tbody>
</table>

**Table 2:** Differences between hardware and software debuggers (source: American Megatrends, Inc. 2011)

For early silicon enabling, the hardware debugger is good for debugging hardware design issues. The direct interface to the processor eliminates communication issues with other platform hardware. The ability to trap hardware events (SMI, INIT, mode switch) is useful in early hardware debugging.

After the initial board power-on, a software debugger can be used to debug complex firmware issues. Full support for hardware and software breakpoints gives developers a great deal of flexibility without using an ITP/JTAG solution. The reduced cost of a software solution enables a larger number of developers access to power debugging tools.

One area of concern is debugging UEFI drivers based on EFI Byte Code (EBC). EBC is executed in a processor-independent virtual machine that interprets a predefined instruction set, similar to Java. EBC code interpretation on the target is hard to debug using a hardware debugger solution. This is an issue that may be addressed in future software-based debug solutions.

**Summary**

The article has described debugging solutions during UEFI firmware development. We can choose any of debugging solutions based on the specific platform hardware design and the firmware problems we meet. Checkpoint solution can be used with PCI POST Checkpoint
Card. A serial port can print out debug messages. A USB debug port can be used as well when a serial port and PCI POST Checkpoint card cannot be used. These solutions can quickly let developers know where firmware execution is. If problems are encountered with silicon problem before production, JTAG software with a JTAG-connector is the most powerful debugging method to do source-level debugging. If the platform has no JTAG-style connectors, AMIDebug and the Intel UDK Debugger Tool can be employed to do source-level debugging as well. These debugging solutions make it possible to develop and debug firmware as general software.

Authors’ Biographies

Stefano Righi is Vice President of Software Utilities at American Megatrends, Inc. (AMI). He has over 25 years of experience in research and development, technologies, product and resource management, with special focus on BIOS and system architecture. During his tenure of more than 12 years at AMI he has been responsible for UEFI BIOS architecture, Firmware Development Environment, Graphical User Interfaces and Pre-Boot solutions including AMIDiag. Stefano is representing AMI in the UEFI Board of Directors.

Brian Richardson started with American Megatrends, Inc. (AMI) in 1996. After spending years in BIOS and test development, Brian moved to technical marketing to support BIOS and UEFI customers. Brian has been a frequent speaker at Intel Developer Forum (IDF) and produced video content for AMI’s YouTube Channel (AMITV).

Jiewen Yao is an EFI/Tiano BIOS engineer in the Software and Solutions Group at Intel and has 8 years’ experience on firmware design, development, and testing. He received an MS degree from Shanghai Jiaotong University and a BS degree from Fudan University. Jiewen holds approximately 10 patents in the firmware area.

Elvin (Wei), Li is a BIOS Engineer in the Software and Solutions Group at Intel and has more than 5 years of UEFI BIOS development experience. He received his MS degree in 2006 and BS degree in 2003 from Beijing Jiaotong University, both in Electrical Engineering. His interests lie predominantly in the areas of embedded system design and system software. His e-mail is elvin.li at intel.com.

This article and more on similar subjects may be found in the Intel Technology Journal, March 2011 Edition, “UEFI Today: Bootstrapping
the Continuum”. More information can be found at http://intel.com/technology/itj.

Copyright

Copyright © 2011 Intel Corporation. All rights reserved. Intel, the Intel logo, and Intel Atom are trademarks of Intel Corporation in the U.S. and other countries. *Other names and brands may be claimed as the property of others.