

# **THE MINNOWBOARD CHRONICLES**

**A JOURNEY OF DISCOVERY IN X86  
ARCHITECTURE, UEFI, DEBUG &  
TRACE, YOCTO LINUX AND OTHER  
TOPICS**

**BY ALAN SGUIGNA**





*Alan Sguigna – Vice President of Sales & Customer Service*

Alan has more than 20 years of experience in senior-level general management, marketing, engineering, sales, manufacturing, finance and customer service positions. Before joining ASSET, he worked in the telecom industry. He has had profit and loss responsibility for a \$150 million division of Spirent Communications, a supplier of test products and services. Prior to his tenure with Spirent, Mr. Sguigna also served in business development positions with Nortel Networks, overseeing the growth of its voice over Internet protocol (VoIP) products.

## Table of Contents

Foreword .....	5
Episode 1: SourcePoint Debugging the MinnowBoard Turbot .....	6
Episode 2: Updating the UEFI Firmware .....	8
Episode 3: Building the UEFI Image.....	13
Episode 4: UEFI Source Code .....	16
Episode 5: PEIM and DXE .....	19
Episode 6: LBR Trace.....	25
Episode 7: Single-Stepping through Code .....	28
Episode 8: The Reset Vector, and Boot Flow .....	32
Episode 9: SourcePoint Command Language and Macros .....	37
Episode 10: The UEFI shell.....	40
Episode 11: Using Instruction Trace.....	45
Episode 12: Writing UEFI Applications.....	49
Episode 13: UEFI Applications using Standard ‘C’ .....	52
Episode 14: Poking around SecCore in UEFI.....	55
Episode 15: More UEFI Application Development in ‘C’ .....	59
Episode 16: Delving into LBR Trace.....	70
Episode 17: Using LBR Trace without Source Code .....	74
Episode 18: Reverse-Engineering Code Execution .....	77
Episode 19: The Yocto Project .....	82
Episode 20: Building and Installing Linux .....	85
Episode 21: Building and Installing Linux, Part 2.....	92
Episode 22: Project Yocto success! .....	95
Episode 23: Trying Wind River Pulsar Linux, and taking a break .....	99

Episode 24: New MinnowBoard, New PC, and a nod to Netgate .....	101
Episode 25: Yocto builds for the MinnowBoard and the Portwell Neptune Alpha.....	106
Episode 26: Linux image build segmentation faults on AMD?.....	111
Episode 27: Segfault on my AMD Ryzen 7 1700X.....	116
Episode 28: Returning my AMD Ryzen 7 1700X CPU .....	119
Episode 29: My new AMD Ryzen 7 CPU works, kind of.....	120
Episode 30: Using all 16 threads on my Ryzen? .....	126
Episode 31: First attempts to debug the Linux kernel .....	129
Afterword.....	138

© 2017-2018 ASSET InterTech, Inc.

ASSET and ScanWorks are registered trademarks, and SourcePoint and the ScanWorks logo are trademarks of ASSET InterTech, Inc. All other trade and service marks are the properties of their respective owners.



## Foreword

In September 2016, while attending the UEFI Forum Plugfest in Bellevue, WA, I got to thinking about how complex and obscure this successor to the BIOS actually was. It seemed to be the domain of technical gurus with a penchant for obscurity – or so it seemed to my “UEFI newbie” mind. Although I had a nodding familiarity with terms like PEIM, DXE, HOBs, and so on, I really had no direct experience with them. And it seemed difficult, if not impossible, to gain the direct, hands-on learning that I like when exploring a new technical topic. There were few easy-to-read books or YouTube videos that I could find. Maybe this was the way it was supposed to be? Maybe you had to work in the field to actually learn the internals? Was this a “security through obscurity” tactic?

While cogitating on this, I happened across the Intel test room at the Plugfest and struck up a conversation with one of the application engineers there. When I expressed my lament, he asked if I knew about the new MinnowBoard Turbot. “Open source hardware, open source software and firmware – this is exactly what you want!”. After a short discussion, we parted ways, and I continued my investigations.

A few weeks later, I was surfing the web and happened across the [MinnowBoard](#) website. This *was* exactly what I was looking for! And access to the UEFI source, with complete build instructions, was in an easy-to-read [tutorial](#).

So, I got my MinnowBoard Turbot for Christmas, and so began a journey of exploration and learning that has lasted over a year. As I explored all facets of the hardware, firmware and software of the Minnow, I wrote about it in the [ASSET InterTech blog site](#). Typically I wrote something fresh once a week, sometimes once every two weeks when life got in the way.

If you are a newbie to UEFI, or want to learn all about firmware, Linux, platform debug, trace features, and the Yocto Project from a set of fresh eyes, this is the place to be. I hope that you enjoy the story as much as I did writing it.

**Alan Sguigna**

**March 19, 2018**

## Episode 1: SourcePoint Debugging the MinnowBoard Turbot

*January 8, 2017*

It may not be everyone's idea of a good time, but I was delighted to receive a MinnowBoard Turbot for Christmas. I hooked it up to my copy of SourcePoint, and the results were pretty cool.

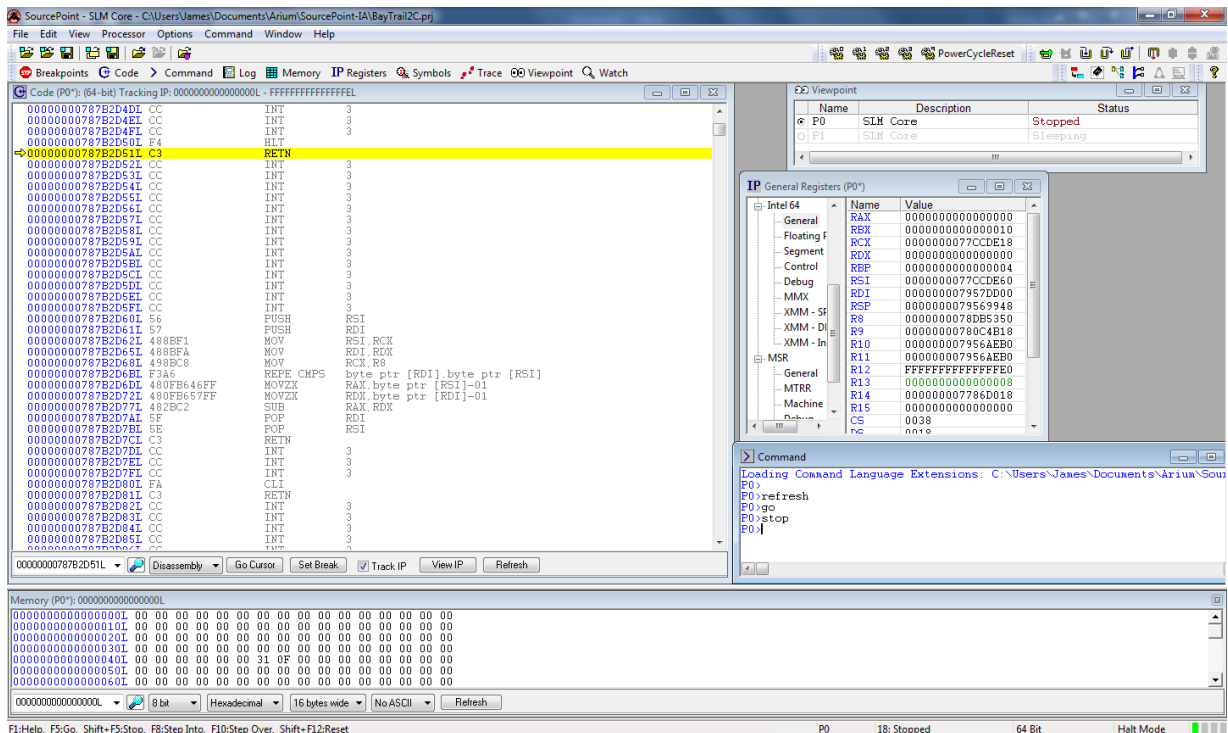
Imagine my surprise when I unwrapped one of my gifts to find a [MinnowBoard Turbot](#) inside. This little PCB is an affordable open-source hardware platform for makers and hackers; think of it as the Intel version of the Raspberry Pi. It sports a dual-core 64-bit Intel® Atom™ E3826 (code name “Bay Trail-I”) system-on-a-chip (SoC) with support for 2GB RAM, USB 2.0, USB 3.0, HDMI, Ethernet, GPIO, SATA2, MicroSD, and a number of other interfaces and built-in capabilities. The entire hardware design is open source and set it up with a USB keyboard and HDMI monitor and it boots right up into the EFI shell so you can play with it right away. It's easy enough to add support for a higher-level OS, such as Debian Linux, Windows IoT, Yocto, or others.

But the *really* cool thing about the MinnowBoard is that it supports a high-speed expansion (HSE) 60-pin connector on the bottom side of the board. This interface is used to connect to a variety of breakout board “Lures” for fast prototyping and tinkering. One of these Lures is the “Debugger Lure” from [Tin Can Tools](#). The Debugger Lure is an expansion board that adds a JTAG debugging interface for the Intel XDP. It is designed to work with Intel's In-Target Probe (ITP) XDP JTAG debugger, but of course it just hooks up to the standard JTAG/XDP interface, so it works with ASSET's SourcePoint debugger too. I was really excited about hooking it up to my ECM-XDP3 hardware emulator and debugging UEFI on the MinnowBoard. Hooked together, it looks like this:



The emulator plugs into the standard 60-pin XDP header on the Debugger Lure, which in turn plugs into the HSE header on the bottom of the MinnowBoard. It was very easy to set up.

Once you have the hardware set up, getting SourcePoint configured is simple. I just opened a New Project, imported the target configuration file for 2-core Bay Trail, powered on the target, and then powered on the emulator. Everything came up the very first time, and after a couple of minutes I had the target halted and in debug mode, with full display of the processor status, x86 registers, memory dump, and disassembled code window.



So, what's next? Well, this is just the beginning. I'm planning on getting full source code and symbols display down on my PC so I can single-step through code and use some of the trace features on the Bay Trail to see how UEFI behaves. Maybe I'll make some changes to the UEFI code underneath and break stuff and see what happens. I also plan on adding Debian Linux to the platform so I'll be able to tinker with the Linux kernel and use SourcePoint to do some OS-aware debugging. I'll write about my adventures (or misadventures) in upcoming blogs.

If you want to know more about SourcePoint, please feel free to visit our website [here](#). There's an excellent video of the GUI which shows the ease-of-use and power of the tool on that page.

You can also request a live demo [here](#).

## Episode 2: Updating the UEFI Firmware

*January 15, 2017*

Last week, I wrote about my out-of-the-box experience with the MinnowBoard Turbot, and how easy it was to start JTAG-based debugging on it with our SourcePoint tool. This week, I explored the UEFI shell and updated the board firmware.

When I first powered up the MinnowBoard Turbot [last week](#), it went directly into the UEFI shell. It's a natural reaction to type "help" at the Shell> prompt (Tip: type "help -b" so the information doesn't scroll off the page), and below is the first screen of what you'll see:

```
alias      - Displays, creates, or deletes UEFI Shell aliases.
attrib    - Displays or changes the attributes of files or directories.
bcfg      - Manages the boot and driver options that are stored in NVRAM.
cd        - Displays or changes the current directory.
cls       - Clears standard output and optionally changes background color.
comp      - Compares the contents of two files on a byte for byte basis.
connect   - Binds a driver to a specific device and starts the driver.
cp        - Copies one or more files or directories to another location.
date      - Displays and sets the current date for the system.
dblk      - Displays one or more blocks from a block device.
devices   - Displays the list of devices managed by UEFI drivers.
devtree   - Displays the UEFI Driver Model compliant device tree.
dh        - Displays the device handles in the UEFI environment.
disconnect - Disconnects one or more drivers from the specified devices.
dmem      - Displays the contents of system or device memory.
dmpstore  - Manages all UEFI variables.
drivers   - Displays the UEFI driver list.
drvcfg    - Invokes the driver configuration.
drvdiag   - Invokes the Driver Diagnostics Protocol.
echo      - Controls script file command echoing or displays a message.
edit      - Full screen editor for ASCII or UCS-2 files.
eficompress - Compress a file using UEFI Compression Algorithm.
efidecompress - Decompress a file using UEFI Decompression Algorithm.
else      - Identifies the code executed when 'if' is FALSE.
endfor    - Ends a 'for' loop.
endif     - Ends the block of a script controlled by an 'if' statement.
exit      - Exits the UEFI Shell or the current script.
for       - Starts a loop based on 'for' syntax.
getwtc    - Gets the WTC from BootServices and displays it.
goto      - Moves around the point of execution in a script.
Press ENTER to continue or 'Q' break: _
```

It's a powerful shell, with a full suite of commands and scripting operators (note to self: look for some good online documentation (with examples) on the UEFI shell, beyond what is available simply within the help system). Since UEFI is so low-level, you can explore the intrinsics of the



BIOS itself, as well as some architectural aspects of the board. For example, the “pci 00 00 00 – i” command displays the PCI configuration space of Bus 0, Device 0, Function 0:

```

Vendor ID(0) : 8086      Device ID(2) : 0F00
Command(4) : 0007
(00) I/O space access enabled:      1  (01) Memory space access enabled:      1
(02) Behave as bus master:          1  (03) Monitor special cycle enabled:    0
(04) Mem Write & Invalidate enabled: 0  (05) Palette snooping is enabled:      0
(06) Assert PERR# when parity error: 0  (07) Do address/data stepping:         0
(08) SERR# driver enabled:           0  (09) Fast back-to-back transact...:    0

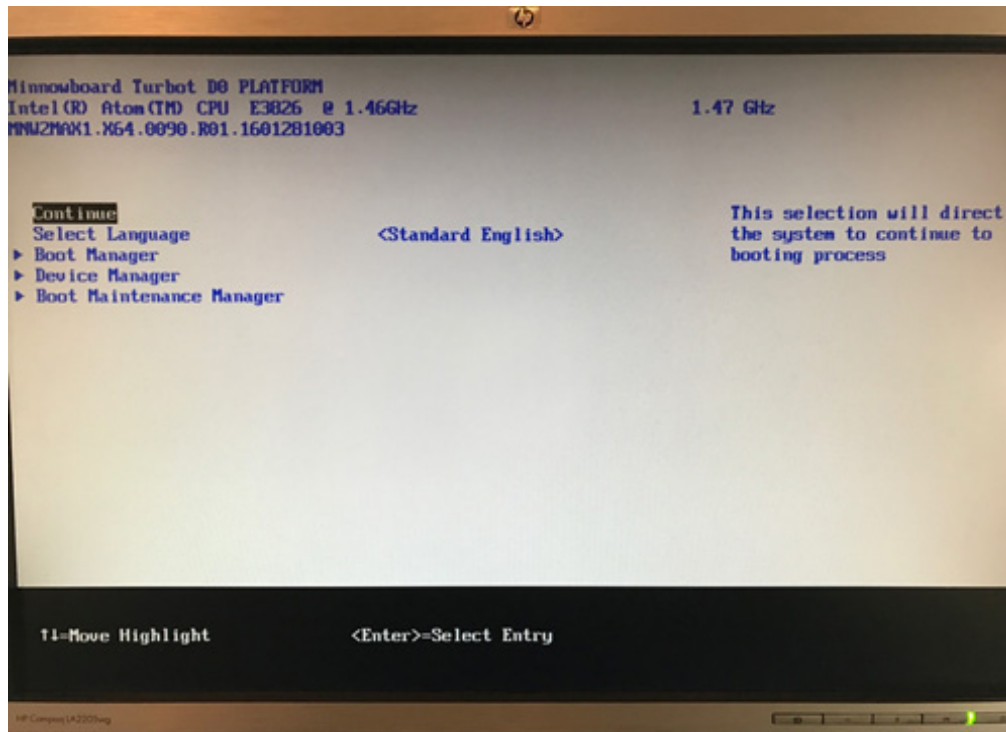
Status(6) : 0000
(04) New Capabilities linked list:    0  (05) 66MHz Capable:                    0
(07) Fast Back-to-Back Capable:       0  (08) Master Data Parity Error:         0
Press ENTER to continue or 'Q' break:
(09) DEUSEL timing:                   Fast  (11) Signaled Target Abort:             0
(12) Received Target Abort:           0  (13) Received Master Abort:            0
(14) Signaled System Error:           0  (15) Detected Parity Error:           0

Revision ID(8) : 11 BIST(0F) : Incapable
Cache Line Size(C) : 00      Latency Timer(D) : 00
Header Type(0E) : 00, Single function, PCI device
Class: Bridge Device - Host/PCI bridge -
Base Address Registers(10) :
  (None)
Expansion ROM Disabled(30)

Cardbus CIS ptr(28) : 00000000
Sub VendorID(2C) :      8086      Subsystem ID(2E) :      7270
Capabilities Ptr(34) :      00
Interrupt Line(3C) :      00      Interrupt Pin(3D) :      00
Min_Gnt(3E) :      00      Max_Lat(3F) :      00
Shell> _

```

After tinkering with the UEFI environment for a while, I realized that the firmware that shipped with the board was a little outdated. This is a picture of the UEFI boot manager screen that came up when I first powered on the unit:

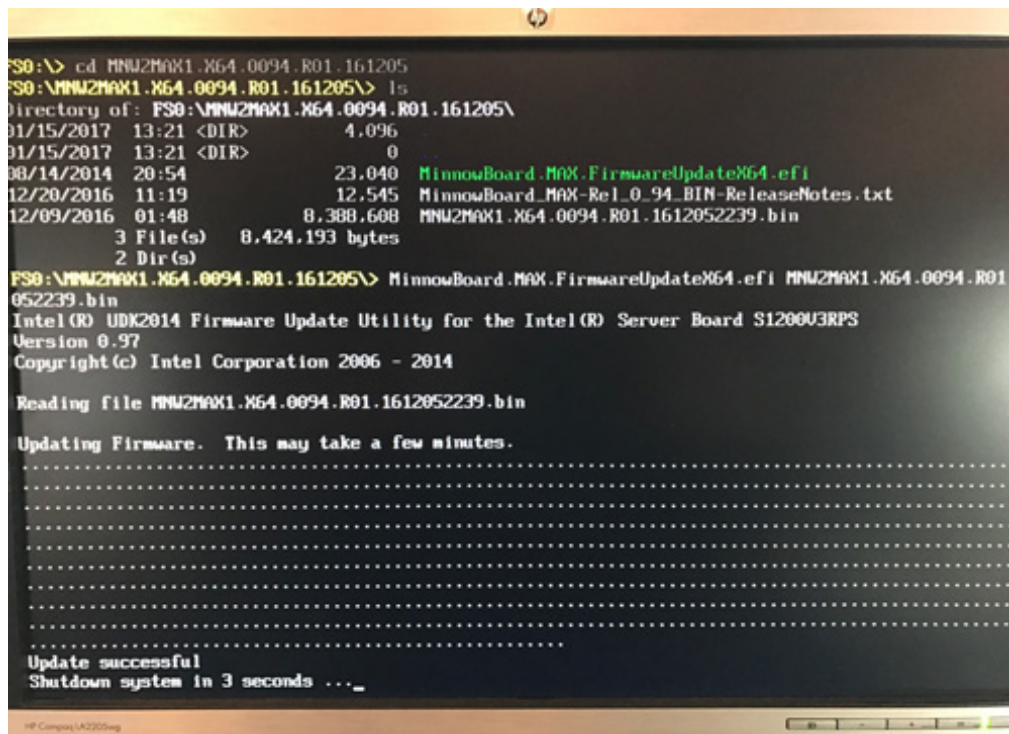


The release of this firmware is MNW2MAX1.X64.0090.R01.1601281003, or in short form 0.90. By going onto the Intel firmware site, <https://firmware.intel.com/projects/MinnowBoard-max>, I saw that the most recent version is 0.94. It was time to update the firmware – something that can be somewhat difficult on Intel designs that you might have at home.

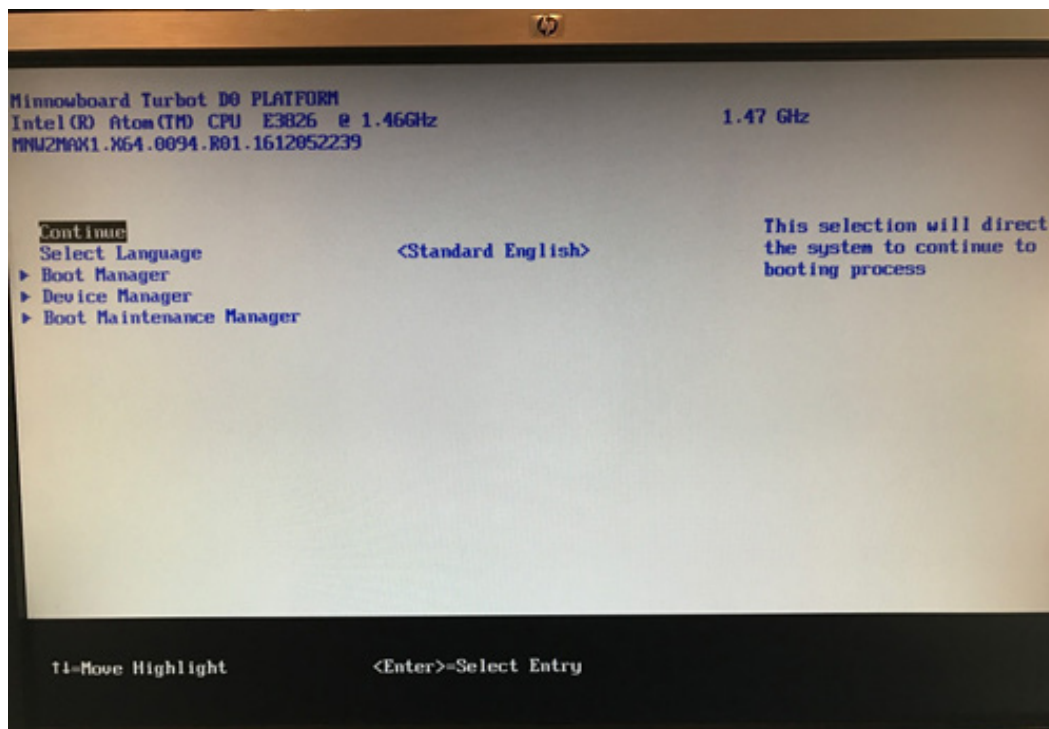
But on the MinnowBoard Turbot, it was easy. Intel’s firmware site provides the EFI shell script as well as the full 64-bit binary image, which I downloaded onto a USB flash stick. Excellent instructions on doing the firmware update are here:

[https://MinnowBoard.org/tutorials/updating\\_your\\_firmware](https://MinnowBoard.org/tutorials/updating_your_firmware) (another note to self: there’s a Debug version of the firmware available too for later exploration). The beauty of it is that you don’t need to flash the board with for example a Dediprog programmer; just run the included UEFI script, and it takes care of the rest.

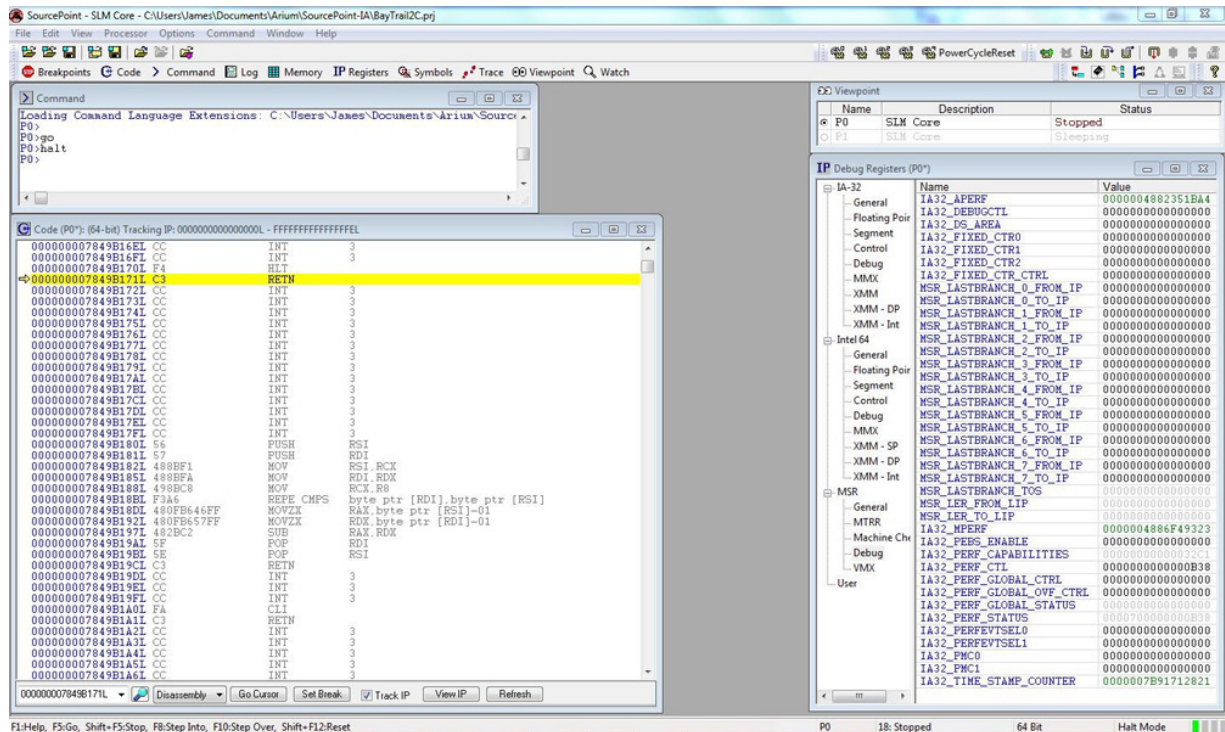
It took some contortions to get the MinnowBoard to recognize the USB stick file system (yet another note to self: start with a small flash drive (8GB), and not a big one (128GB), and make sure that it is FAT32-formatted), but after playing around a little, the update started, and completed successfully:



And, you can see from the UEFI boot manager options screen that the new update (release 0.94) is now installed:



It then occurred to me to launch [SourcePoint](#) and see what the code is doing while waiting for keyboard input from the boot manager screen, as opposed to the UEFI shell from last week. I went through the same procedure as last week, and you can see the disassembled code window [here](#):



What shows up in the Code window is basically identical to what we saw in last week's [blog](#), except for being at a different address. The instruction pointer is sitting at a single RETN instruction. That is worthwhile investigating!

Of course, it would be much clearer too if we had source code – that’s on my agenda for next time.

The commercial is saved for the end: if you want to know more about SourcePoint, please feel free to visit our website [here](#). There's an excellent video of the GUI which shows the ease-of-use and power of the tool on that page. You can also request a live demo [here](#).



## Episode 3: Building the UEFI Image

*January 23, 2017*

As I continue the journey to learn about the internals of UEFI and to debug it with SourcePoint, I encounter some issues doing the firmware build.

[Last week](#), I played around with the UEFI shell, and then updated the firmware on my MinnowBoard to the latest release (v0.94). Then, I used SourcePoint to look at disassembled code when the platform was sitting in the UEFI shell, waiting for keyboard input. From last time, we can see a number of “INT 3” instructions, with opcode CC.

I wanted to explore this a little more but decided to be more aggressive and go ahead and try for a full UEFI EDK II firmware build. My hope was to successfully create the symbol files to be used as input to SourcePoint. With this in place, I can use our tool to do source-level debugging, set breakpoints within modules such as DxeMain, use the Trace capabilities, and so on. Debug is so much more powerful when source code and symbols are available.

Luckily, the [Release Notes for Release 0.94](#) of the MinnowBoard UEFI firmware are very clear when it comes to doing a build. The steps are broken down into the following:

1. Download the complete source from [Tianocore](#) using Git.
2. Get the binaries (the parts that are not available in source code).
3. Set up the build environment (in this case, Visual Studio 2013).
4. Install the needed IASL compiler.
5. Do the build.

I could spend hours describing the detail behind each step, but I'll leave that for follow-up blogs. I'll simply describe the challenges and issues I ran into here.

#1 was easy. All that was required was to download Git from [www.git-scm.com](http://www.git-scm.com), and follow the included instructions. Git is a version control repository and Internet hosting service. It was very easy to use and allowed me to access the needed source files in my Windows PC workspace.

The binary files needed were another story. I was later to find that my anti-virus blocked the downloading of some of these files (silently, of course). Many hours were spent trying to find out why the build was not completing. For example, the anti-virus program did not like the SmmControl.efi macro.

Getting my old dual-core Windows PC to install Visual Studio 2013 took about two hours. Microsoft, of course, initially downloaded the newest Visual Studio 2015 Community – despite my having accessed the Visual Studio 2013 page – but Visual Studio 2015 did not work. I had to download and install Visual Studio 2013 explicitly before I could make any progress.

Another tip for those who might follow in this path: you do want to download the OpenSSL source code. Don't try to skip this step. I did, and it cost me another hour.

The IASL compiler is obtained from the [ACPIA](#) (Advanced Configuration and Power Interface Component Architecture) website. Note to self: I want to dig into this a little later.

After all this, doing the build was easy. From “C:\MyWorkspace\Vlv2TbItDevicePkg” it was a simple matter of typing into the Windows command window “Build\_IFWI.bat MNW2 Release” and waiting about 20 minutes.

The final 8MB firmware binary image MNW2MAX1.X64.0094.R01.1701221828.bin appears in the directory “C:\MyWorkspace\Ulv2TbItDevicePkg\Stitch”.

```

Administrator: Developer Command Prompt for VS2013
The Guid Tool Definition comes from the build-in default configuration.
Intel(R) Firmware Configuration Editor. (Intel(R) FCE) Version 0.29. .
BfmLib Version: 0.30
Decoding
Start the Update Mode:

-- Update List --

[Results]: 0 question has been updated successfully in total.
Congratulations. The output Fd file '..\Build\Ulv2TbItDevicePkg\RELEASE_US2012x86\FU\UlvX64.fd' has been completed successfully.
Build location:      Build\Ulv2TbItDevicePkg\RELEASE_US2012x86
BIOS ROM Created:    MNW2MAX_X64_R_0094_01.ROM

----- The EDKII BIOS build has successfully completed. -----

Sun 01/22/2017  18:44:34.23
Finished Building BIOS.
BIOS_ID=MNW2MAX1.X64.0094.R01.1701221828
=====
Build_IFWI: Calling IFWI Stitching Script...
Sun 01/22/2017  18:44:35.13

Creating backup of original BIOS rom.
Stitching IFWI for ..\..\MNW2MAX_X64_R_0094_01.ROM ...

IFWI Header: IFWIHeader\IFWI_HEADER.bin, SEC version: 1.0.3.1164,
BIOS Version: 0094_01
Platform Type: MNW2, IFWI Prefix: MNW2MAX1.X64.0094.R01.1701221828
=====

Generating IFWI... MNW2MAX1.X64.0094.R01.1701221828.bin
IFWIHeader\IFWI_HEADER.bin
..\..\Ulv2MiscBinariesPkg\SEC\1.0.3.1164\ULU_SEC_REGION.bin
..\..\Ulv2MiscBinariesPkg\SEC\1.0.3.1164\Uvacant.bin
..\..\MNW2MAX_X64_R_0094_01.ROM
1 file(s) copied.

=====

-- All specified ROM files Stitched. --
-- See Stitching.log for more info. --

Sun 01/22/2017  18:44:35.37

Build_IFWI is finished.
The final IFWI file is located in C:\MyWorkspace\Ulv2TbItDevicePkg\Stitch\
=====
C:\MyWorkspace\Ulv2TbItDevicePkg>

```

This coming week, I'll be pulling the symbols into [SourcePoint](#), and doing some hardware-assisted source-level debug!

## Episode 4: UEFI Source Code

*January 29<sup>th</sup>, 2017*

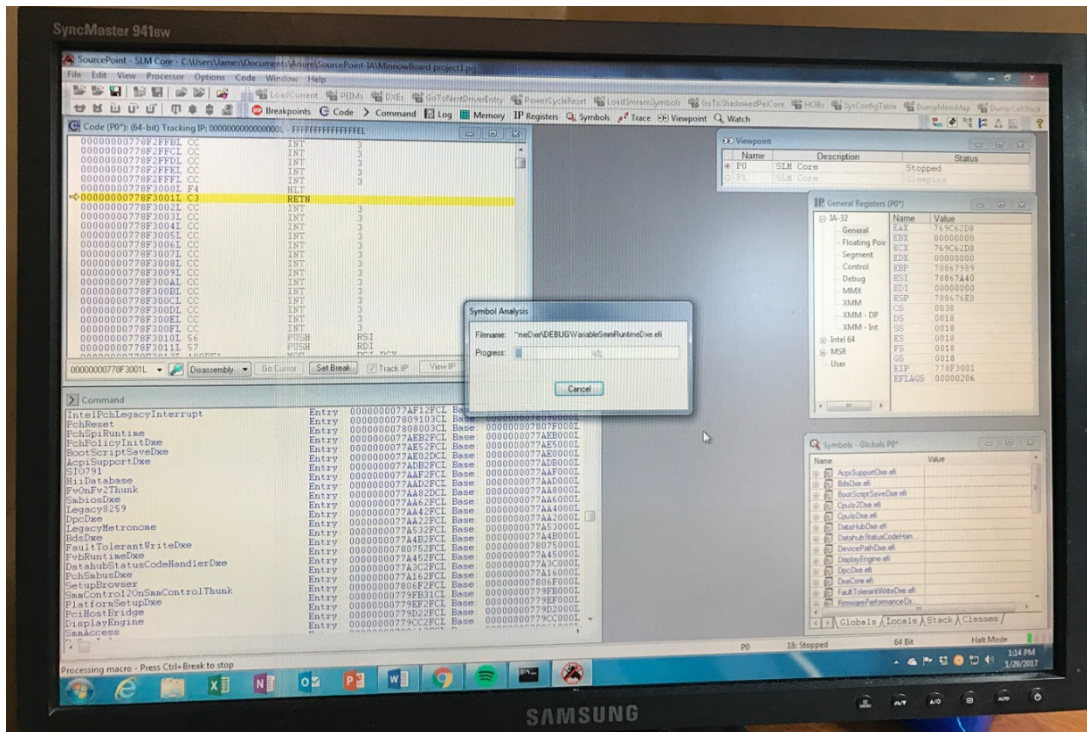
Success! This week, I managed to compile a debug version of the UEFI, load it into the MinnowBoard, and see UEFI source code in SourcePoint for the first time.

Last week, in the [MinnowBoard Chronicles Episode 3](#), I did a source build for the release BIOS of the MinnowBoard Turbot (also known somewhat interchangeably as the MinnowBoard MAX). With ASSET's JTAG-based hardware-assisted x86 debugger, [SourcePoint](#), I was able to see disassembled code for the UEFI shell. This was interesting, but it only whet my appetite for more investigation. By downloading the UEFI source tree, I had access to the source code, and wanted to see it within our debugger in a meaningful code execution context. Making the source code visible to the SourcePoint debugger took some extra steps.

The linker .map files contain information on the absolute (or relative) addresses for the code that is part of the object build. When loaded into the target, the UEFI firmware on the target contains strings that hold the paths to the program symbol files on your hard drive. SourcePoint macros can be executed to read target memory, find these strings, then load the symbol files specified in these paths. The symbol files must be located in the same path specified in the UEFI firmware. So, it was a relatively straightforward matter to rebuild a Debug image with the symbol information, and then run the EFI.mac macro file located in the SourcePoint Macro\UEFI directory. This creates six custom toolbar buttons and associates each with a corresponding UEFI procedure:

- The StartPEI icon resets the target, then runs to PeiMain and loads the PEI symbols.
- The PEIMs (Pre-UEFI Initialization Modules) icon loads the symbol files for the PEI modules found in target memory.
- The DXEs (Driver Execution Environments) icon loads the symbol files for the DXE modules found in target memory.
- The HOBs (Hand-Off Blocks) icon displays a list of UEFI HOBs found in target memory.
- The SysConfigTable icon displays the contents of the UEFI system configuration table.
- The DumpMemMap icon displays the UEFI Memory Map.

After clicking on the DXE icon within SourcePoint, I loaded much of the DXE source code and symbols. It took a few minutes, with clear progress bar indicators along the way:



Ultimately, I got all the symbols and source code loaded and displayed, and could easily click on Globals, Locals, a view of the Stack, and all Classes:

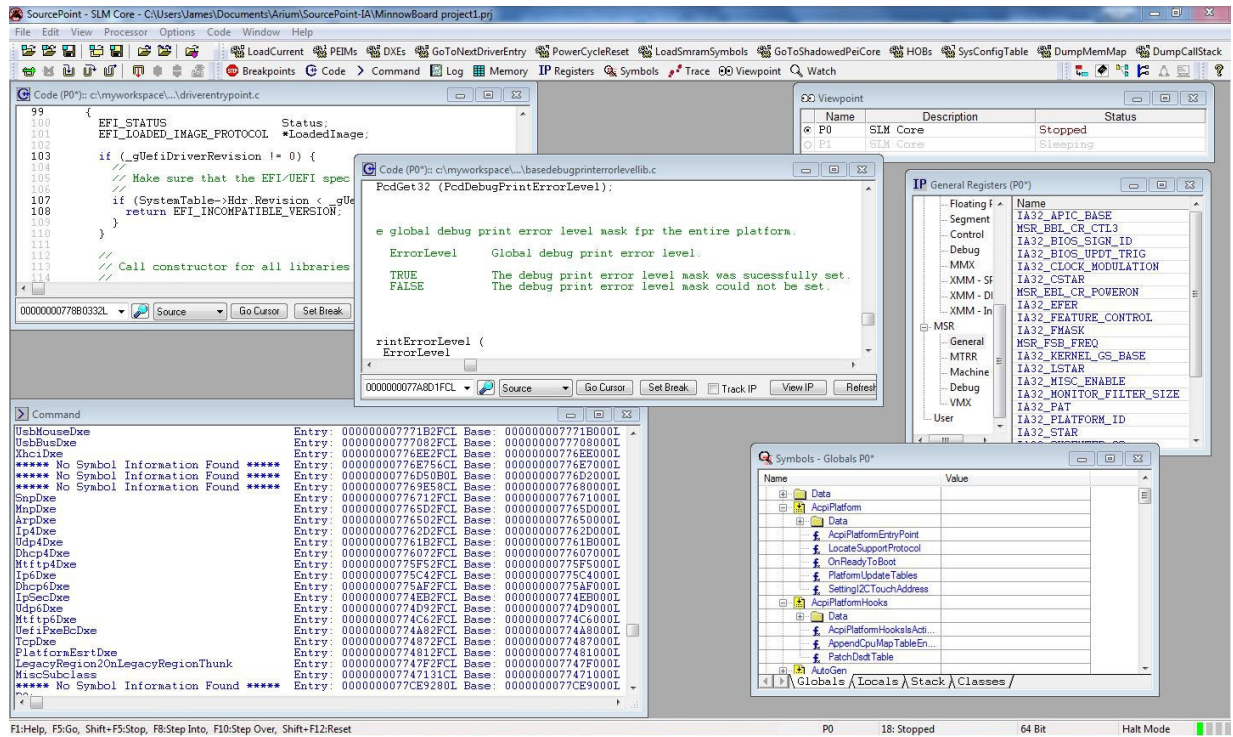
The **Globals** tab displays a hierarchy of loaded programs. Programs can be expanded to show modules, procedures, and symbols.

The **Locals** tab shows the variables accessible in the current stack frame.

The **Stack** tab shows the stack as a list of stack frames.

The **Classes** tab lists structure and class definitions in a hierarchy similar to that under the **Globals** tab.





Very impressive!

In my next episode of the MinnowBoard Chronicles, I'll be exploring the PEI, looking at HOBs and the UEFI System Configuration table, and loading symbols just before the DXE modules run instead of running to the UEFI shell.

## Episode 5: PEIM and DXE

*February 5, 2017*

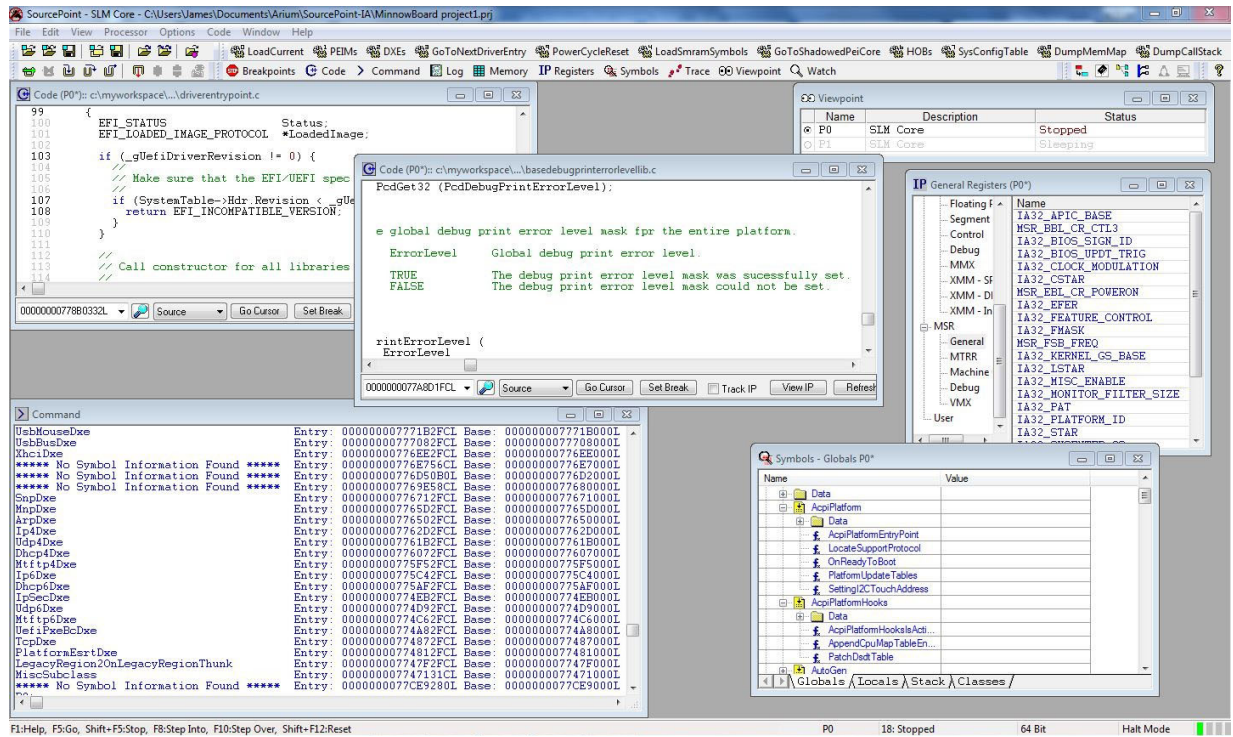
This week, I explore DXE, and do a deeper dive into the PEI.

Last week in the [MinnowBoard Chronicles, Episode 4](#), I achieved a significant milestone: compiling a debug version of the UEFI, loading it into the MinnowBoard, and seeing UEFI source code within the SourcePoint JTAG debugger for the first time.

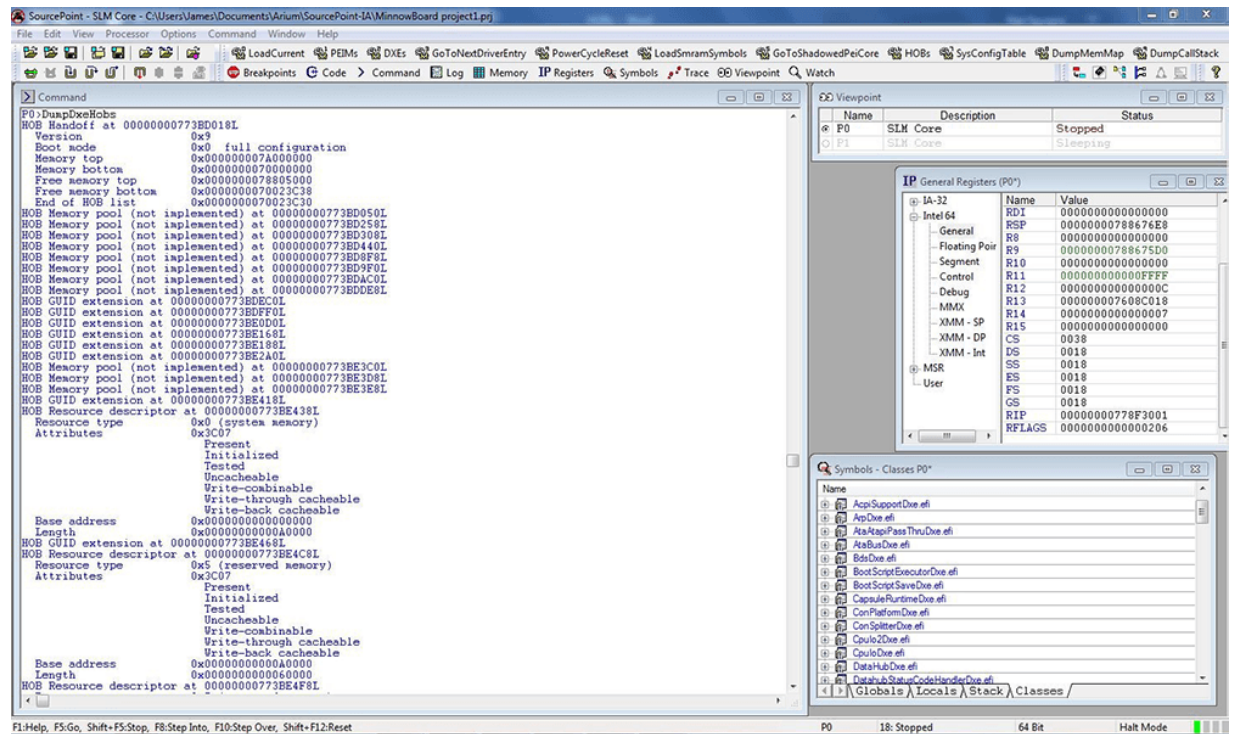
But this, of course, has only whet my appetite for further exploration. I'm fortunate enough to have the MinnowBoard available as an open source hardware and software platform that boots into the UEFI shell. So, the sky's the limit in terms of what I can learn about this x86-based platform and the UEFI software which is at the heart of almost all Intel-based PCs, workstations and servers. The fact that I also have available SourcePoint, which allows me to see all architecturally visible aspects of the Intel hardware and firmware, creates a very powerful learning environment.

Last week, I booted the MinnowBoard into the UEFI Shell, where it simply sits and waits for keyboard input. Then I Halted the target to put it into debug mode (also known as probe mode); after which the SourcePoint DXE macro loads the source symbols. After that, it's a very simple matter of opening the Symbols window to see the Globals, Locals, the Stack, and Classes.

If you want to poke around, it helps that Symbol Search on SourcePoint is screamingly fast. The UEFI build is prodigiously large, and for example doing a "Find Symbol" wildcard search on everything prefaced with "DXE" could in principle take a significant amount of time, given that the database is so large. With SourcePoint, the search is displayed in real time!



There are also pre-built EFI macros available within SourcePoint to view the HOBs, System Configuration Table, the System Memory Map, and other interesting structures. The debugger is “UEFI-aware”.





SourcePoint - SLM Core - C:\Users\James\Documents\Ariun\SourcePoint-IA\MinnowBoard project1.prj

File Edit View Processor Options Command Window Help

LoadCurrent PEIMs DXEs GoToNextDriverEntry PowerCycleReset LoadSmmramSymbols GoToShadowedPeiCore HOBs SysConfigTable DumpMemMap DumpCallStack

Breakpoints Code Command Log Memory IP Registers Symbols Trace Viewpoint Watch

Command

```

P0>DumpDxeHob
HOB Handoff at 00000000773BD018L
Version 0x9 full configuration
Memory top 0x000000007A000000
Memory bottom 0x0000000070000000
Free memory top 0x0000000078905000
Free memory bottom 0x0000000070023C38
End of HOB list 0x0000000070023C30
HOB Memory pool (not implemented) at 00000000773BD050L
HOB Memory pool (not implemented) at 00000000773BD258L
HOB Memory pool (not implemented) at 00000000773BD308L
HOB Memory pool (not implemented) at 00000000773BD440L
HOB Memory pool (not implemented) at 00000000773BD0F0L
HOB Memory pool (not implemented) at 00000000773BD9F0L
HOB Memory pool (not implemented) at 00000000773BDAC0L
HOB Memory pool (not implemented) at 00000000773BDDE8L
HOB GUID extension at 00000000773BDEC0L
HOB GUID extension at 00000000773BDEF0L
HOB GUID extension at 00000000773BD0D0L
HOB GUID extension at 00000000773BE168L
HOB GUID extension at 00000000773BE188L
HOB GUID extension at 00000000773BE2A0L
HOB Memory pool (not implemented) at 00000000773BE3C0L
HOB Memory pool (not implemented) at 00000000773BE3D8L
HOB Memory pool (not implemented) at 00000000773BE3E8L
HOB GUID extension at 00000000773BE418L
HOB Resource descriptor at 00000000773BE438L
Resource type 0x1 (system memory)
Attributes 0x3C07
Present
Initialized
Tested
Uncacheable
Write-combining
Write-through cacheable
Write-back cacheable
Base address 0x0000000000000000
Length 0x000000000000A0000
HOB GUID extension at 00000000773BE468L
HOB Resource descriptor at 00000000773BE4C8L
Resource type 0x5 (reserved memory)
Attributes 0x3C07
Present
Initialized
Tested
Uncacheable
Write-combining
Write-through cacheable
Write-back cacheable
Base address 0x000000000000A0000
Length 0x00000000000060000
HOB Resource descriptor at 00000000773BE4F8L

```

Viewpoint

Name	Description	Status
P0	SLM Core	Stopped
P1	SLM Core	Sleeping

IP General Registers (P0\*)

Name	Value
IA-32	
RD1	0000000000000000
General	
RSP	00000000788676E8
R8	0000000000000000
Floating Pair	
R9	00000000788675D0
Segment	
R10	0000000000000000
Control	
R11	00000000000000FF
Debug	
R12	0000000000000000
MMX	
R13	000000007608C018
XMM - SP	
R14	0000000000000007
XMM - DP	
R15	0000000000000000
CS	0038
XMM - Int	
DS	0018
SS	0018
ES	0018
FS	0018
GS	0018
RIP	00000000778F3001
RFLAGS	0000000000000206

Symbols - Classes P0\*

AcpiSupportDxe.efi  
 AppDxe.efi  
 AkaAppPassThruDxe.efi  
 AkaBusDxe.efi  
 BdsDxe.efi  
 BootScriptExecutorDxe.efi  
 BootScriptSaveDxe.efi  
 CapsuleRuntimeDxe.efi  
 ConPlatformDxe.efi  
 ConSplitterDxe.efi  
 CpuIo2Dxe.efi  
 CpuIoDxe.efi  
 DataHubDxe.efi  
 DatabaseStatusCodeHandlerDxe.efi  
 Globals\Locals\Stack\Classes

Fl:Help, F5:Go, Shift+F5:Stop, F8:Step Into, F10:Step Over, Shift+F12:Reset

P0 18: Stopped 64 Bit Halt Mode

SourcePoint - SLM Core - C:\Users\James\Documents\Ariun\SourcePoint-IA\MinnowBoard project1.prj

File Edit View Processor Options Window Help

LoadCurrent PEIMs DXEs GoToNextDriverEntry PowerCycleReset LoadSmmramSymbols GoToShadowedPeiCore HOBs SysConfigTable DumpMemMap DumpCallStack

Breakpoints Code Command Log Memory IP Registers Symbols Trace Viewpoint Watch

Code (P0\*) (64-bit) Tracking IP: 0000000000000000 - FFFFFFFF

```

00000000778F2FEF CC INT 3
00000000778F2FF1 CC INT 3
00000000778F3001 F4 INT 3
00000000778F3001 CC RETN
00000000778F3002 CC INT 3
00000000778F3003 CC INT 3
00000000778F3004 CC INT 3
00000000778F3005 CC INT 3
00000000778F3006 CC INT 3
00000000778F3007 CC INT 3
00000000778F3008 CC INT 3
00000000778F3009 CC INT 3
00000000778F300A CC INT 3
00000000778F300B CC INT 3
00000000778F300C CC INT 3
00000000778F300D CC INT 3
00000000778F300E CC INT 3
00000000778F300F CC INT 3
00000000778F3010 54 PUSH RSI
00000000778F3011 57 PUSH RDI
00000000778F3012 48BFF1 MOV RSI, RCX
00000000778F3013 48BFA MOV RDI, RDX

```

Command

```

HOB End of HOB list at 00000000773C0C48L
P0>
P0>stop
Loading User Defined Macro #7: C:\Users\James\Documents\Ariun\SourcePoint-IA\Macros\EFI\button\EfiBtn7.Cl
P0>DumpConfigTable
EFI_SYSTEM_TABLE at 00000000773C018L
EFI Table Version 2.50

```

Viewpoint

Name	Description	Status
P0	SLM Core	Stopped
P1	SLM Core	Sleeping

IP General Registers (P0\*)

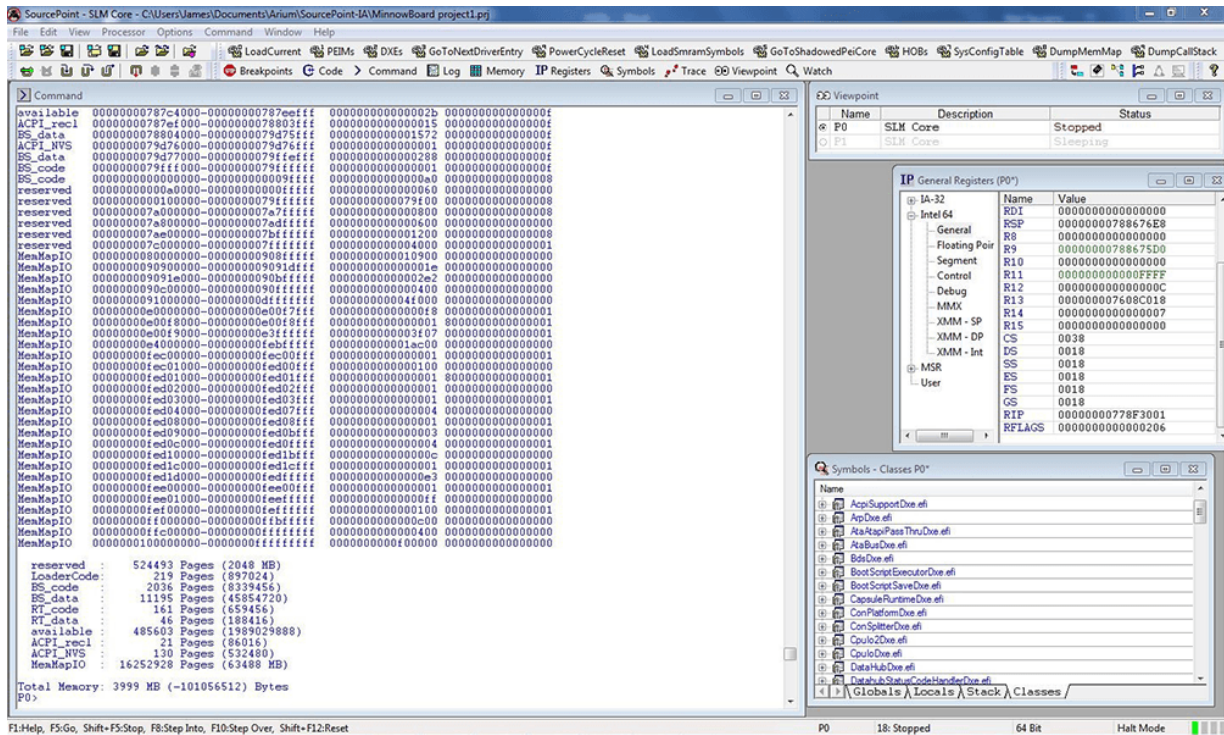
Name	Value
IA-32	
RD1	0000000000000000
General	
RSP	00000000788676E8
R8	0000000000000000
Floating Pair	
R9	00000000788675D0
Segment	
R10	0000000000000000
Control	
R11	00000000000000FF
Debug	
R12	0000000000000000
MMX	
R13	000000007608C018
XMM - SP	
R14	0000000000000007
XMM - DP	
R15	0000000000000000
CS	0038
XMM - Int	
DS	0018
SS	0018
ES	0018
FS	0018
GS	0018
RIP	00000000778F3001
RFLAGS	0000000000000206

Symbols - Classes P0\*

AcpiSupportDxe.efi  
 AppDxe.efi  
 AkaAppPassThruDxe.efi  
 AkaBusDxe.efi  
 BdsDxe.efi  
 BootScriptExecutorDxe.efi  
 BootScriptSaveDxe.efi  
 CapsuleRuntimeDxe.efi  
 ConPlatformDxe.efi  
 ConSplitterDxe.efi  
 CpuIo2Dxe.efi  
 CpuIoDxe.efi  
 DataHubDxe.efi  
 DatabaseStatusCodeHandlerDxe.efi  
 Globals\Locals\Stack\Classes

Fl:Help, F5:Go, Shift+F5:Stop, F8:Step Into, F10:Step Over, Shift+F12:Reset

P0 18: Stopped 64 Bit Halt Mode

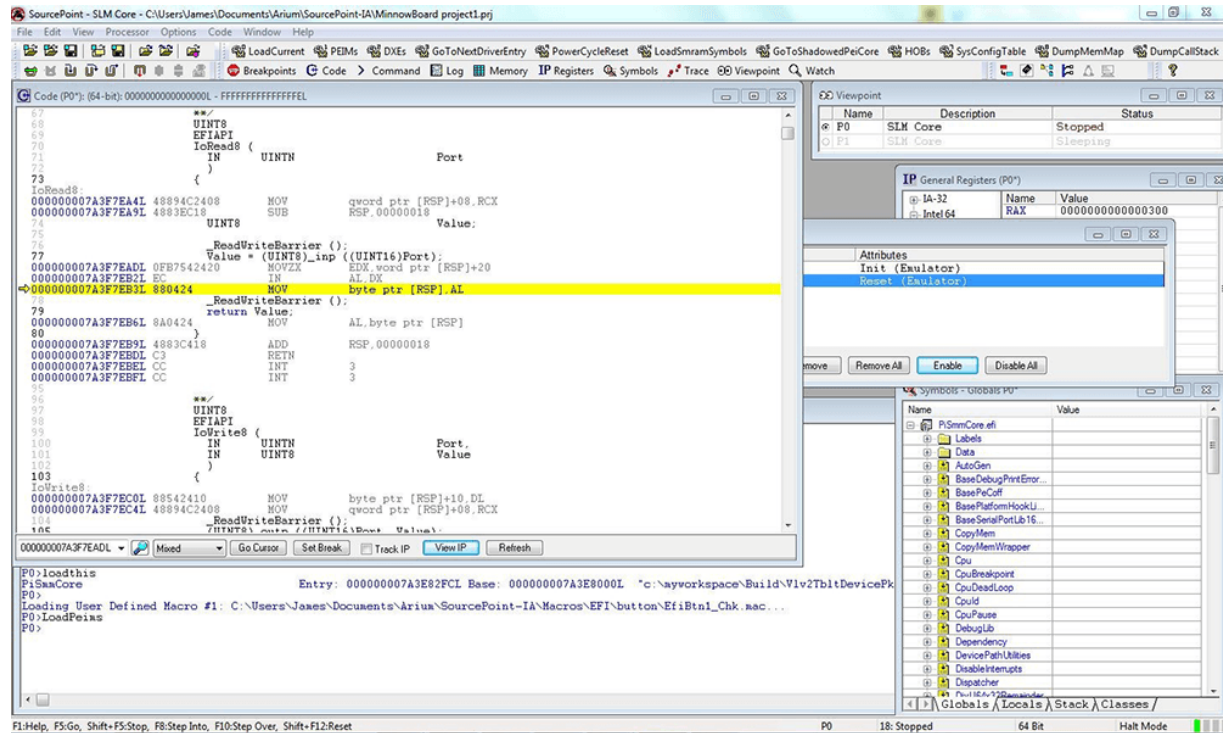


I want to come back to DXE again sometime soon, but for this week, the PEI beckoned. I have a deep interest in exploring the code executing as close to system reset as possible – and maybe one day, before reset. But, for now, exploring early PEI was on the agenda.

First, I tried to take control of the target by initiating a PowerCycleReset out of SourcePoint, but that seemed to put the MinnowBoard into a strange state, and the only way to recover was to manually power cycle the board. I'm wondering if there's a rogue watchdog coming from somewhere, or if there's something wrong with the emulator's access to some of the key XDP pins. I know that HOOK0 (PWRGOOD) is hooked up, because SourcePoint detects and displays that when there is No Power on the board. HOOK6 (RESET\_IN#) is hooked up, but I'm not sure about HOOK7 (RESET\_OUT#). This is something I need to check on the MinnowBoard [schematics](#) when I have a little time. There might also be a flaw I guess in the [Debugger Lure](#) design from Tin Can Tools, but unfortunately, the [schematics](#) are not available – I get the dreaded 404 Page Not Found message – hopefully they'll fix that soon, or steer me in the right direction!

In the meantime, it's easy enough to power cycle the target, turn on the emulator, and launch SourcePoint to halt the target before it gets too far into the boot cycle. Then running the

LoadPeims macro (just by clicking on the PEIMs button) loads all the program symbols and points the code view window back to the beginning of the code block where the processor was stopped:



You can see from the Instruction Pointer on the left that the code is stopped at address `0x'000000007A3F7EBE'`, different than where we stopped at the UEFI shell before at `0x'000000007849B171'`, for example, within [Episode 2](#) of the MinnowBoard Chronicles. We're also within `PiSmmCore`.

Scrolling back a little, we see that we are in the function `IoRead8()`. It's also clear on the Stack tab within the Symbols window:



Well, that's it for now. There's a lot to digest. If you have any questions about getting started using [SourcePoint](#) on the MinnowBoard to delve into the BIOS, drop me a note.

## Episode 6: LBR Trace

*February 12, 2017*

This week, I delve into using LBR (Last Branch Record) Trace on the MinnowBoard and continue my exploration into the UEFI source code.

Last week, in the [MinnowBoard Chronicles – Episode 5](#), we saw how our SourcePoint debugger could be used to view the UEFI Hand-off Blocks (HOBs), System Configuration Table, System Memory Map, and other structures. Single-stepping through source code and watching the Call Stack dynamically update in real-time was also fascinating. This is one of the best ways to gain hands-on experience with UEFI, in my humble opinion.

This week, I decided to employ one of the most powerful capabilities within the Intel architecture, the on-die Trace logic. Specifically, of interest are the Last Branch Record (LBR) and Branch Trace Store (BTS) functions.

LBR trace displays a history of executed instructions. It does this by reading Branch Trace Messages (BTMs) from the Last Branch Record MSRs in the processors. The advantage of LBR trace is it is non-intrusive. The processor can run at full speed when using LBR trace. The disadvantage of LBR trace is the limited number of LBRs available (typically 4 - 16). Each LBR stores a single BTM. If you assume an average of 5 instructions between branches, then roughly the last 80 instructions executed are traced.

BTS trace displays a history of executed instructions. It does this by reading Branch Trace Messages (BTMs) from the Branch Trace Store (BTS), an area of system memory set aside for trace. The BTS can be much larger and store many more BTMs than LBRs. The disadvantage of using the BTS is that writing BTMs to memory takes longer than writing BTMs to LBRs. Each branch results in 12-24 bytes of memory being written. For some applications BTS trace may result in too high a speed penalty to use. Another disadvantage of BTS trace is the inability to trace out of reset (if memory is unavailable). Still, getting the much larger buffer can be a good trade-off.

It is worthwhile noting that LBR trace and BTS trace are not mutually exclusive. They can both be enabled at the same time.



You see in the top left in the Code Window that we are viewing the code in “Mixed” mode: that is, the ‘C’ source code is interleaved with its associated assembly language instructions. The instruction pointer points to the beginning of the processor instructions associated with the line of code:

```
while ((SerialPortReadRegister (SerialRegisterBase, R_UART_LSB)
& B_UART_LSR_TEMT) == 0)
```

The instruction pointer is at address ‘000000007889BEF5’L. Note that the ‘L’ signifies a memory linear address, and we are in real (or protected) mode. This is the same as the physical address if paging is not in effect. We are within the SerialPortWrite() function.

In the LBR Trace window, we see the instructions that were executed prior to the location of the instruction pointer. The State field displays the state number (cycle number) of the instruction. The trigger location is marked as 0. Cycles before the trigger are shown as negative numbers. In the first cycle, you can see that we are inside the “while” loop; the five assembly language instructions shown correspond to the last five assembly language instructions shown in the Code window. This is the bottom of the “while” loop, where the `JNE SerialPortWrite+e0` and the `JMP SerialPortWrite+c5` instructions handle the conditional logic of the loop.

But then the information in the Trace window gets interesting. Where State shows -000002, we see a bunch of assembly language instructions ending in an `IRETQ` instruction. `IRETQ` is a 64-bit return from an Interrupt. So, just before the `MOVZX AX, AL` instruction, we have just gotten back from an interrupt service routine, wherein there were a bunch of `POP` statements that restore the values of the general-purpose registers (GPRs) from the stack.

At least that’s my interpretation as of this moment. I’ll need to dig into this a little more to confirm my theory. A good place to start might be to look at the `SerialPortReadRegister` function in some more detail.

I’ll continue this exploration of the MinnowBoard using [SourcePoint](#) next week.

## Episode 7: Single-Stepping through Code

*February 20, 2017*

This week, I single-step through source code to track the execution of programs, and better understand how the MinnowBoard BIOS works.

I'm really curious as to the overall UEFI structure program flow as a platform boots up. A lot is happening as the platform initializes, and the source code is very large, so it's hard to see what the code is doing from just a static viewpoint. I wanted to see what routines call and are called by other routines. Seeing the code execution flow from close-up is definitely a learning experience.

The best way to do this is to step through the code at my own speed, as opposed to the speed that the code normally executes at. [SourcePoint](#) provides stepping operations that, in conjunction with the go/stop and breakpoint capabilities, allow me to effectively track through the execution of programs.

It is worth noting that there are three stepping commands available:

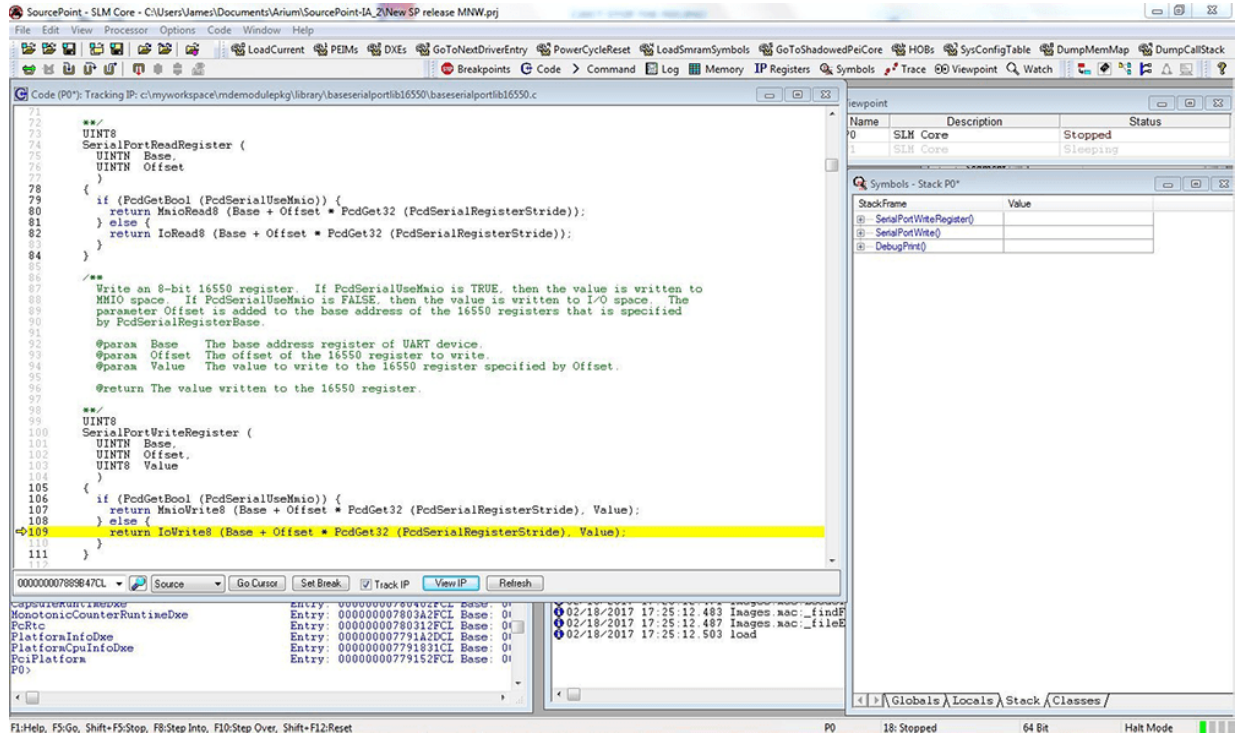
**Step Into.** This single-steps the next instruction in the program and enters each function call that is encountered. This is useful for detailed analysis of all execution paths in a program.

**Step Over.** This single-steps the next instruction in the program and runs through each function call that is encountered without showing the steps in the function. This is useful for analysis of the current routine while skipping the details of called routines.

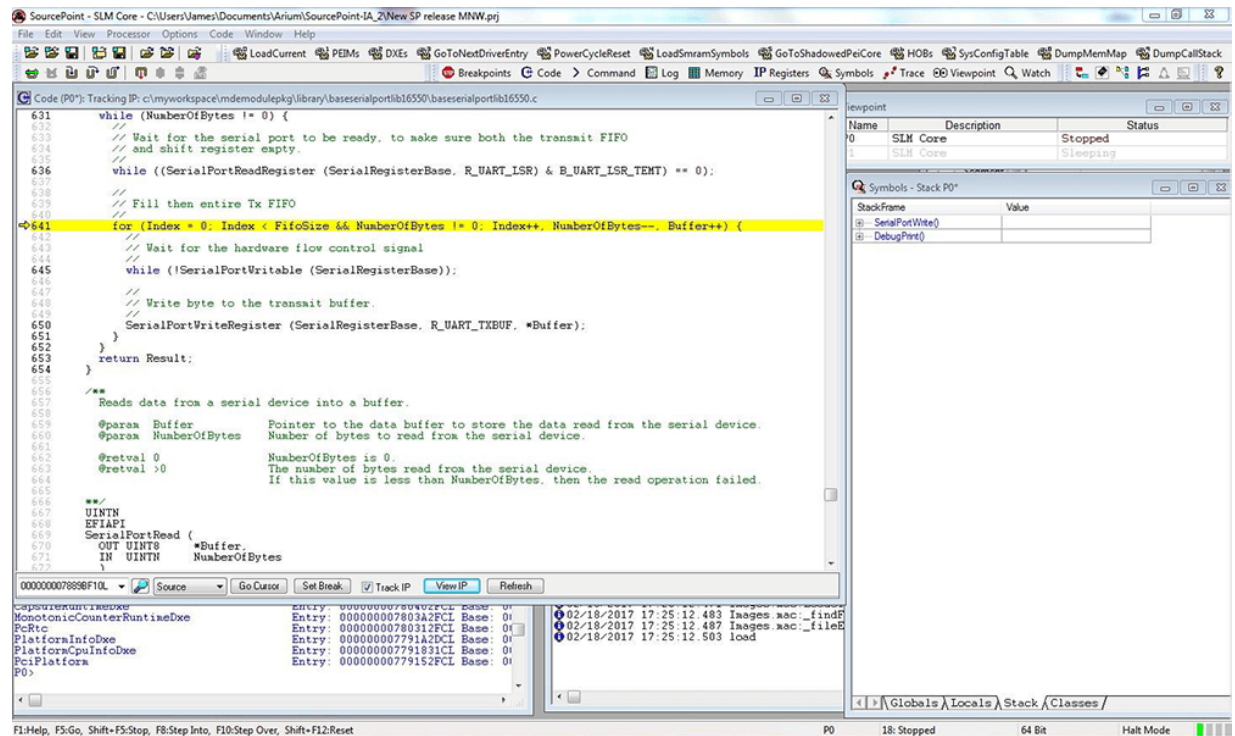
**Step Out Of.** This single-steps the next instruction in the program and runs through the end of an existing function context. This is useful for a quick way to get back to the parent routine.

To put these through their paces, I broke within the `SerialPortWriteRegister()` routine. The source code and instruction pointer is visible on the left. From the Call Stack on the right, you can see that `SerialPortWriteRegister()` is called by `SerialPortWrite()` which in turn is called from `DebugPrint()`:

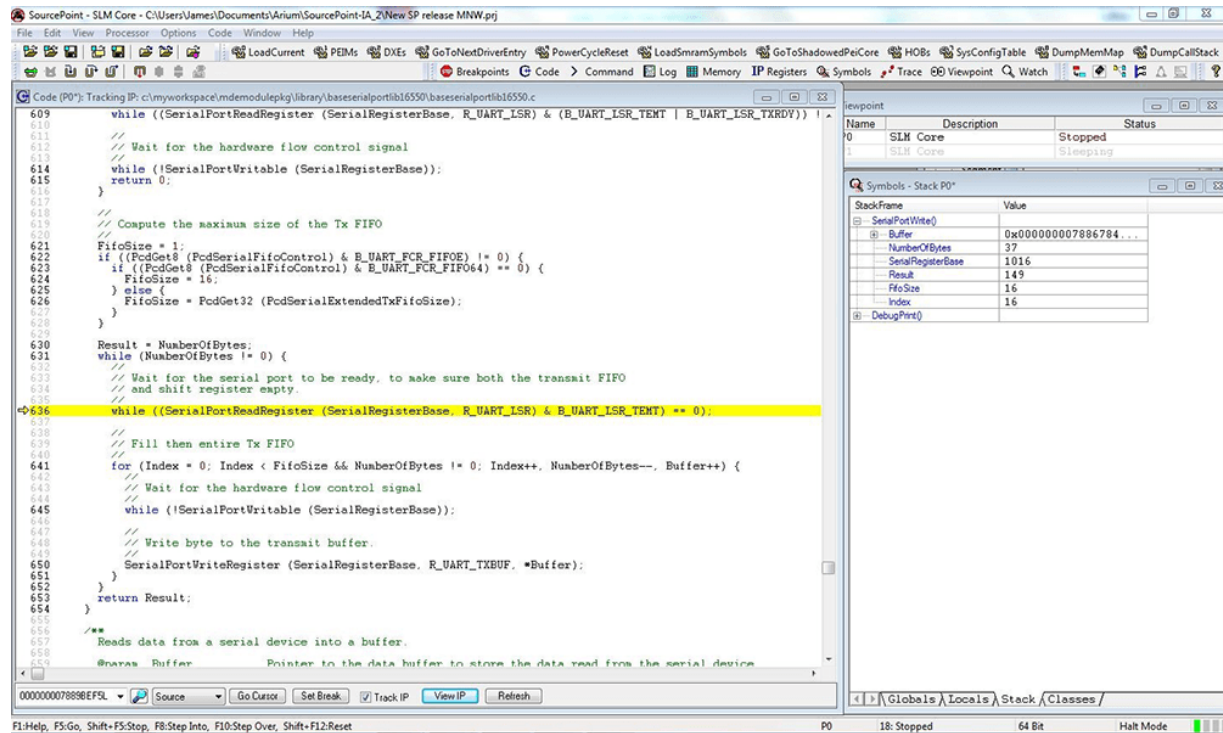




I decided to Step Into some number of execution steps, and as expected the SerialPortWriteRegister() function returns, and I can come into SerialPortWrite() – as can be seen by the StackFrame display updating in real-time:

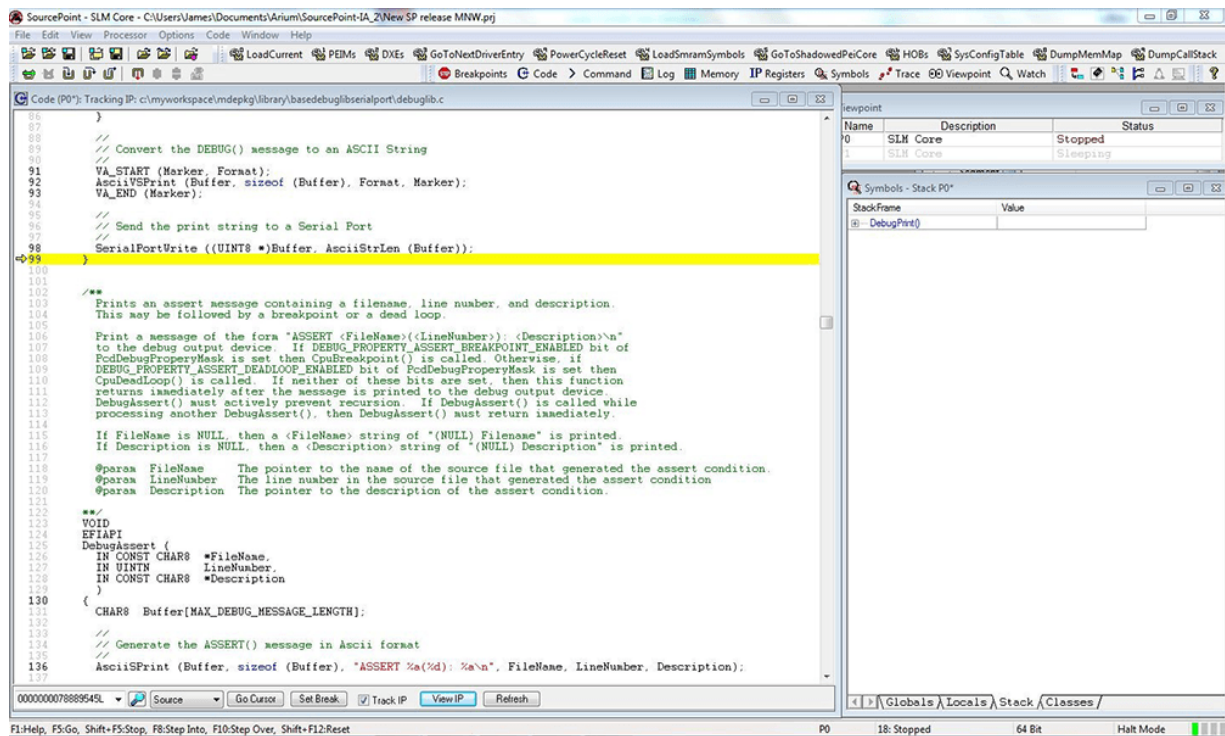


It can be seen that within `SerialPortWrite()` I'm in the middle of a "for" loop that makes multiple calls to `SerialPortWriteRegister()` indexed by the values of `Index` and `NumberOfBytes`. It's easy enough to hover over the values of these variables in SourcePoint and also look at the Locals tab in the Symbols window to see how deep the "for" loop is:



Oh, this is going to take a while. `Index` is 8, and `FifoSize` is 16, but I also have to get `NumberOfBytes` down to 0. So, I dutifully start to click Step Over, expecting this to take a while. But wait! Maybe Step Out Of is a solution to my problem.

And that does turn out to be the case. I get right back into `DebugPrint()`:



Step Into one more time puts me into a whole new section of code. I'm in `PeCoffLoaderExtraActionCommon()` and within the `pecoffextraactionlib.c` (whereas before I was in `debuglib.c`). What is this? More exploration forthcoming!

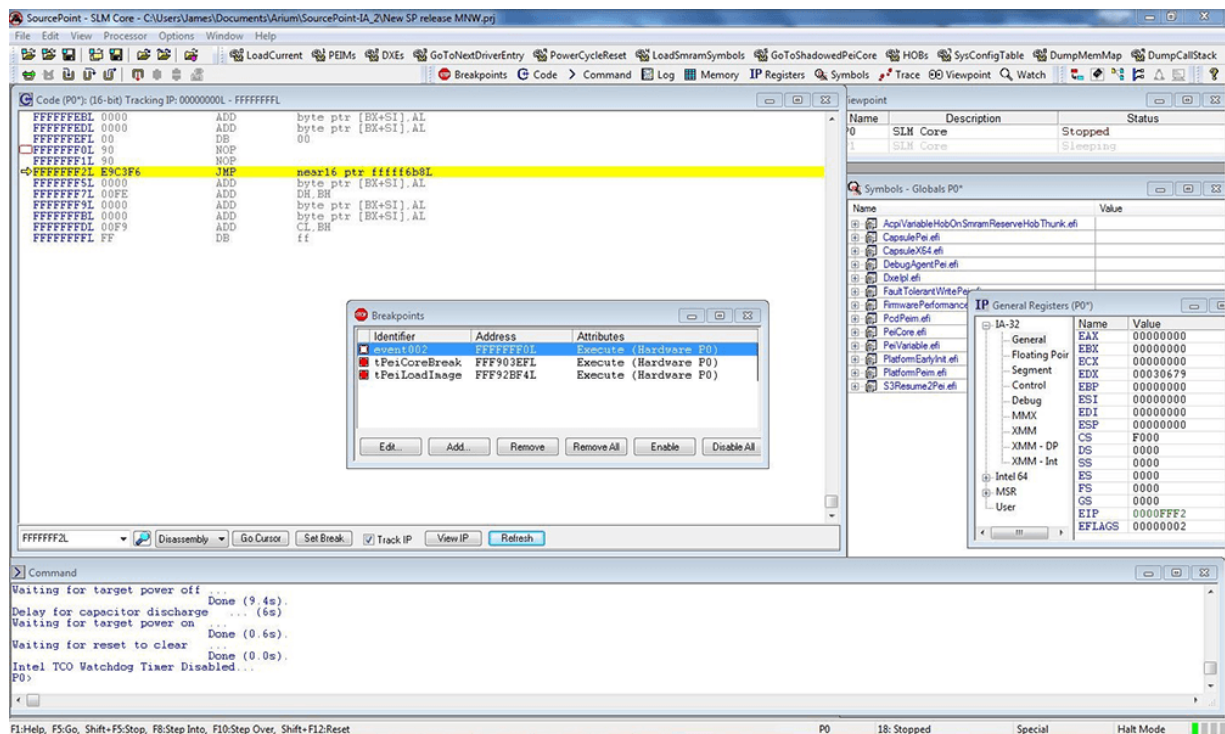
For those who would like to learn more, there's an excellent eBook here: [UEFI Framework Debugging](#) (note: requires registration).

## Episode 8: The Reset Vector, and Boot Flow

February 26, 2017

This is getting pretty intense. It's time to explore the code that executes at the reset vector, and see the printf messages coming out of the BIOS as the MinnowBoard boots.

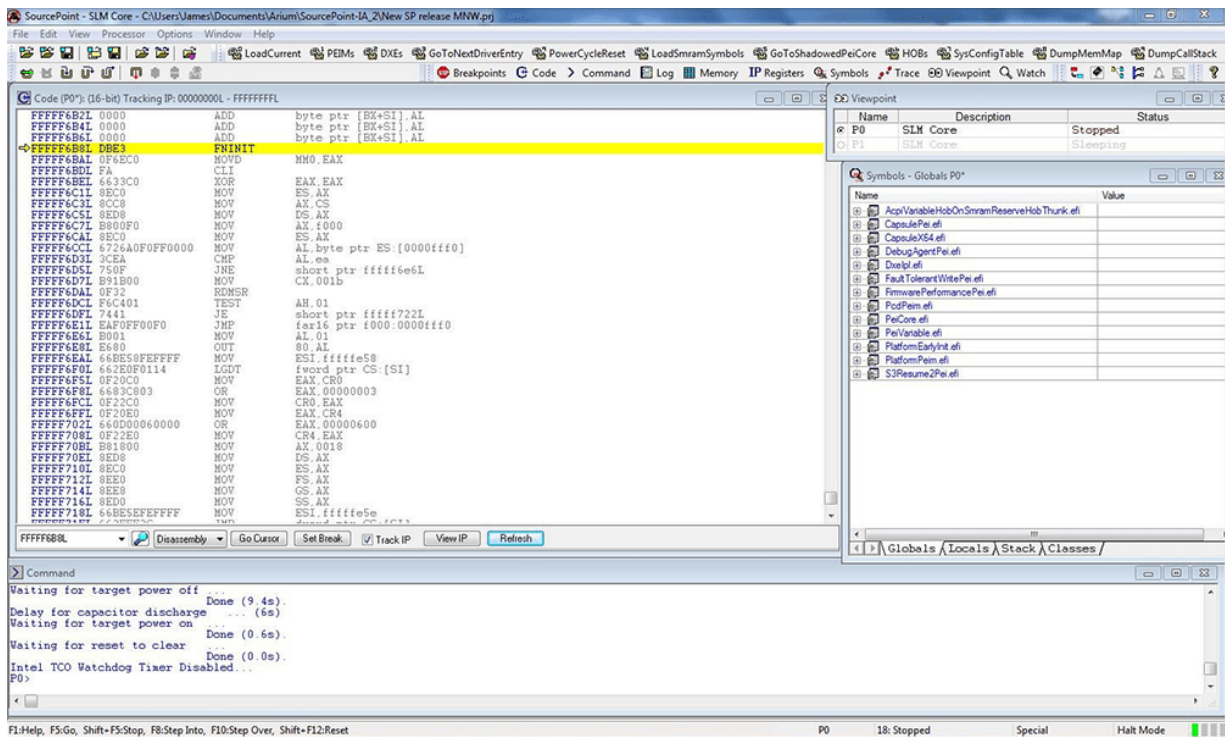
Having gotten past my reset issues on the MinnowBoard, I set a hard breakpoint directly at the reset vector (which is at physical address FFFFFFF0h on the MinnowBoard). The reset vector is the default location (address) where the BayTrail CPU finds the first instruction it will execute after a reset. You can see these first instructions in the [SourcePoint](#) debugger Code window:



There are a couple of NOP instructions, followed by a near JMP to address FFFFF6B8.

Everything is in assembly language because the initial code is being executed out of flash memory. Looking at the code at FFFFF6B8 is informative.





The first instruction there is FPINIT, which initializes the processor floating point unit (FPU). Then there is some more assembly language code, which I will have to determine the meaning of later. It's a good time to load the PEI modules and try to see some source code somewhere, by clicking on the PEIMs button at the top of the screen. An excerpt of the output within the Command window looks like this:

P0>LoadPeims

PlatformEarlyInit Entry: FFF203A0L Base: FFF20140L  
 "c:\myworkspace\Build\Vlv2TbltDevicePkg\DEBUG\_VS2012x86\IA32\Vlv2TbltDevicePkg\PlatformInitPei\PlatformInitPei\DEBUG\PlatformEarlyInit.efi"

PchSmbusArpDisabled Entry: FFF29C20L Base: FFF299C0L FILE NOT FOUND  
 "m:\Build\Vlv2TbltDevicePkg\DEBUG\_VS2008x86\IA32\Vlv2DeviceRefCodePkg\ValleyView2Soc\SouthCluster\Smbus\Pei\PchSmbusArpDisabled\DEBUG\PchSmbusArpDisabled.pdb"

VlvInitPeim Entry: FFF2B820L Base: FFF2B5C0L FILE NOT FOUND  
 "m:\Build\Vlv2TbltDevicePkg\DEBUG\_VS2008x86\IA32\Vlv2DeviceRefCodePkg\ValleyView2Soc\NorthCluster\VlvInit\Pei\VlvInitPeim\DEBUG\VlvInitPeim.pdb"

PchInitPeim Entry: FFF2DA00L Base: FFF2D7A0L FILE NOT FOUND  
 "m:\Build\Vlv2TbltDevicePkg\DEBUG\_VS2008x86\IA32\Vlv2DeviceRefCodePkg\ValleyView2Soc\SouthCluster\PchInit\Pei\PchInitPeim\DEBUG\PchInitPeim.pdb"

PchSpiPeim Entry: FFF33280L Base: FFF33020L FILE NOT FOUND

```

"m:\Build\Vlv2TbltDevicePkg\DEBUG_VS2008x86\IA32\Vlv2DeviceRefCodePkg\ValleyV
iew2Soc\SouthCluster\Spi\Pei\PchSpiPeim\DEBUG\PchSpiPeim.pdb"

PeiSmmAccess                      Entry: FFF35880L Base: FFF35620L  FILE NOT
FOUND

"m:\Build\Vlv2TbltDevicePkg\DEBUG_VS2008x86\IA32\Vlv2DeviceRefCodePkg\ValleyV
iew2Soc\CPU\SmmAccess\Pei\SmmAccess\DEBUG\PeiSmmAccess.pdb"

PeiSmmControl                     Entry: FFF37000L Base: FFF36DA0L  FILE NOT
FOUND

"m:\Build\Vlv2TbltDevicePkg\DEBUG_VS2008x86\IA32\Vlv2DeviceRefCodePkg\ValleyV
iew2Soc\SouthCluster\SmmControl\Pei\SmmControl\DEBUG\PeiSmmControl.pdb"

S3Resume2Pei                     Entry: FFF38280L Base: FFF38020L
"c:\myworkspace\Build\Vlv2TbltDevicePkg\DEBUG_VS2012x86\IA32\UefiCpuPkg\Unive
rsal\Acpi\S3Resume2Pei\S3Resume2Pei\DEBUG\S3Resume2Pei.efi"

```

A number of the modules cannot be found. But for those that are, there are handy pointers to the source build tree and where the .MAP files are. More on this later.

As I prepare to delve even further into the source code involved in the early boot process, the other indispensable aid is to take advantage of the debug printf statements that come out of the serial port of the MinnowBoard. I obtained an inexpensive [serial-to-USB cable](#) on Amazon, and hooked it up to my Mac using the [CoolTerm](#) application. That is when things got really exciting: as I booted up the system, I saw some very interesting information come out to the terminal. An excerpt of the first couple of screens are:

```
>>>>SecStartup
```

```

Mono Status Code PEIM Loaded
Install PPI: 1F4C6F90-B06B-48D8-A201-BAE5F1CD7D56
Install PPI: AB294A92-EAF5-4CF3-AB2B-2D4BED4DB63D
Register PPI Notify: F894643D-C449-42D1-8EA8-85BDD8C65BDE
DetermineTurbotBoard() Entry
MmioConf0[0xFED0E200], MmioPadval[0xFED0E208]
Gpio_S5_4 value is 0x3
Gpio_S5_17 value is 0x3
GGC: 0x00000210 GMSSize:0x00000002
CheckCfioPnpSettings: CFIO Pnp Settings Disabled
DetermineTurbotBoard() Entry
MmioConf0[0xFED0E200], MmioPadval[0xFED0E208]
Gpio_S5_4 value is 0x3
Gpio_S5_17 value is 0x3
Setting BootMode to BOOT_WITH_FULL_CONFIGURATION
Setup MMIO size ...

Install PPI: E767BF7F-4DB6-5B34-1011-4FBE4CA7AFD2
PROGRESS CODE: V3020003 I0
PDB =
m:\Build\Vlv2TbltDevicePkg\DEBUG_VS2008x86\IA32\Vlv2DeviceRefCodePkg\Txe\SeCU
ma\SeCUmaPeim\DEBUG\SeCUma.pdb

```

```

Loading PEIM at 0x000FFFA2E20 EntryPoint=0x000FFFA3080 SeCUma.efi
PROGRESS CODE: V3020002 I0
Install PPI: CBD86677-362F-4C04-9459-A741326E05CF
Info: SeC PPI load sucessfully
PROGRESS CODE: V3020003 I0
    PDB =
c:\myworkspace\Build\Vlv2TbltDevicePkg\DEBUG_VS2012x86\IA32\SourceLevelDebugP
kg\DebugAgentPei\DebugAgentPei\DEBUG\DebugAgentPei.pdb
Loading PEIM at 0x000FFFA49A0 EntryPoint=0x000FFFA4C00 DebugAgentPei.efi
PROGRESS CODE: V3020002 I0
Install PPI: 3CD652B4-6D33-4DCE-89DB-83DF9766FCCA
Debug Timer: FSB Clock      = 200000000
Debug Timer: Divisor        = 2
Debug Timer: Frequency      = 100000000
Debug Timer: InitialCount   = 10000000
Register PPI Notify: F894643D-C449-42D1-8EA8-85BDD8C65BDE

```

```

Set MRC paramaters for MinnowBoard Max.
MmioConf0[0xFED0E220], MmioPadval[0xFED0E228]
Gpio_S5_5 value is 0x3
Determine the memory size is [2GB]
DRAM_Speed is 1066MHz!
DRAM_Speed is type 1, EccEnabled = 0
tCL = 7
tRP_tRCD = 7
tWR = 8
tWTR = 5
tRRD = 6
tRTP = 4
tFAW = 28
PROGRESS CODE: V51001 I0
POSTCODE=<0024>
fastboot
MRC getting memory size from SeC ...
SeC Device ID: F18
SeC UMA Size Requested: 16384 KB
MRC SeCUmaSize memory size from SeC ... 10
MRC getting fTPM memory size from SeC ...
MRC SeCfTPMUmaSize memory size from SeC ... 0
PROGRESS CODE: V51002 I0
POSTCODE=<0025>
PROGRESS CODE: V51003 I0
POSTCODE=<0027>
Configuring Memory...
CheckMicrocodeRevision = 00000906, CpuId = 00030679
####: ConfigureMemory() Entry
Current function is MMRC_Init
    Current function is McEnableHPET
    Current function is ClearSelfRefresh
    Current function is OemTrackInitComplete
    Current function is ProgSFRVolSel
    Current function is ProgDdrTimingControl
    Current function is ProgBunit
    Current function is ProgMpllSetup
    Current function is ProgStaticDdrSetup
    Current function is ProgStaticInitPerf
    Current function is ProgStaticPwrClkGating

```

```

Current function is DUnitBlMode
Current function is ControlDDR3Reset
Current function is EnableVreg
Current function is ProgHmc
Current function is ProgReadWriteFifoPtr
Current function is ProgComp
Current function is SetIOBUFACT
Current function is ProgDdecodeBeforeJedec
Current function is PerformDDR3Reset
Current function is PreJedecInit
Current function is PerformJedecInit
Current function is SetDDRInitializationComplete
Current function is DisableRank2RankSwitching
Current function is MMRC_RcvnRestore
Current function is MMRC_WrLvlRestore
Current function is MMRC_RdTrainRestore
Current function is MMRC_PerformanceSetting
Current function is MMRC_PowerGatingSetting
Current function is MMRC_SearchRmt
START_RMT:
      . RxDqLeft RxDqRight RxVLow RxVHigh TxDqLeft TxDqRight
CmdLeft CmdRight
-----
Channel 0 Rank 0 . -20      18      -23      17      -26      23      0
0
STOP_RMT:
CMD module is per channel only and without Rank differentiation
Current function is ProgDraDrb
  Current function is ProgMemoryMappingRegisters
  Current function is ProgDdrControl
  Current function is SetScrambler
  Current function is ChangeSelfRefreshSetting
  Current function is SetInitDone
  Current function is McDisableHPET
  Current function is FillOutputStructure
MRC INIT DONE

```

There is a treasure trove of information here, relating the stages of the boot process, the PEIMs being loaded, and the functions being performed. I'm at the point now where I can start relating all of this activity back to the original source code within the debug build I created with Visual Studio. Then use the SourcePoint debugger to single-step through the code and see its flow. It's going to get even more exciting from here on in.



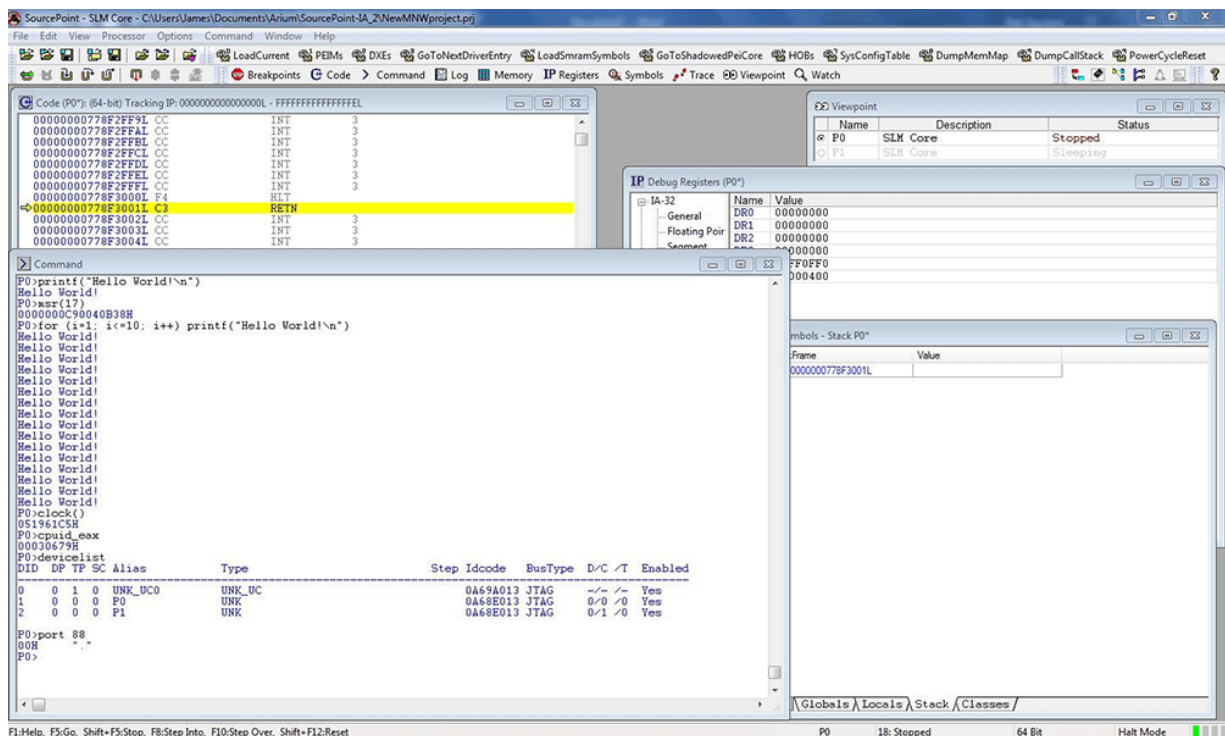
## Episode 9: SourcePoint Command Language and Macros

*March 19, 2017*

Today, I learn more about Intel Architecture and UEFI on the MinnowBoard, by using the built-in commands and macros within SourcePoint.

SourcePoint has a very powerful built-in programming language that is very similar to 'C'. This language gives access to a plethora of commands for JTAG-based run-control, target access, dumping of registers/memory/IO, etc. These allow for command-line control of the debugging environment, as well as creation of scripts for automation of often-repeated tasks. The command language complements source-level debugging, and is particularly useful for hardware qualification and validation. The command language primitives in fact form the foundation of a lot of the SourcePoint functionality, such as the Intel CScripts.

The command language syntax, data type support, use of expressions, control variables, and other aspects are defined within the SourcePoint User Guide. I found the use of the command language to be very easy. You can use the command language directly from the Command window; below are some examples:



The first command issued was invoked by simply typing in:

```
printf("Hello World!\n")
```

Anyone familiar with ‘C’ will recognize this statement. As expected, it prints out “Hello World!” to the console.

The following command, `msr(17)`, prints out the contents of the mode-specific register at address 17H. Those familiar with Intel Architecture may recognize this as the `MSR_PLATFORM_ID`.

You can see from the “for” loop that it is possible to create complex statements in a single line. It’s also possible to chain together multiple statements on a single line, and even wrap them onto several lines.

The `clock()` macro returns the elapsed time (in ms) since SourcePoint started. The value is ‘051961C5’H; which is 85549.509 seconds, or about 24 hours.

The next command, `cpuid_eax`, executes the assembly language `CPUID` instruction and returns the result in `EAX` (and to the screen). You can see that the result is ‘30679’H.

“devicelist” displays the attributes of the devices in the chain known to SourcePoint. The uncore and the two device cores are visible.

Of course, it is possible to create command macro text files which contain multiple commands. Creating command files helps to automate oft-repeated operations. Command files are also referred to as macro files, script files or include files. There are several ways to execute a command file:

- Use the include command in the Command window.
- Drag and drop a command file from Windows Explorer to the Command window.
- Select **File | Macro | Load Macro** from the main menu.
- Select **File | Macro | Configure Macros** to attach a command file to a user-defined toolbar button, and then press the button.
- Select **File | Macro | Configure Macros** to attach a command file to an event. Examples of events include: go, stop, project load, power cycle, etc. When the event occurs, the macro will automatically execute.
- Define a breakpoint and specify a command file to execute when the breakpoint hits.

An example of a macro file which reads the Bay Trail-I MSRs up to '6E0'H consists of only seven lines:

```
define ord8 i=0
define ord8 msrvalue = 0
while (i < 6E0) {
    msrvalue = msr(i)
    printf("%x %x \n", i, msrvalue)
    i += 1
}
```

The results can be compared against the MSR definitions for the Silvermont architecture, which are also contained in the [Intel Software Developers Manuals](#). Pretty cool, huh?

To have a look at the SourcePoint GUI, go [here](#).

More information on SourcePoint Command macros can be found [here](#).

Use of the SourcePoint command environment to invoke the Intel CScripts is found in our eBook [here](#) (note: requires registration).

## Episode 10: The UEFI shell

*March 26, 2017*

Today, I created my first UEFI shell script. And, by a happy coincidence, I noticed a new book in Amazon, [Harnessing the UEFI Shell](#), by Michael Rothman, Vince Zimmer, and Tim Lewis.

In last week's [Episode 9](#) of the MinnowBoard Chronicles, I used the macro command language within [SourcePoint](#) to print "Hello World", read Intel MSRs, and display devices on the MinnowBoard's JTAG chain, among other things. I really liked the power of the command environment: its ability to execute command line functions, chain multiple commands on a single line, wrap multiple commands over multiple lines, and of course, execute scripts. The SourcePoint command language has rich support for conditional logic, looping, branching, and the other capabilities you might expect out of a scripting language. And since it comprehends the 'C' programming language and is identical to the legacy Intel ITP I programming language, my learning curve was extremely short.

Since my interest is in UEFI, this week I decided to explore its shell, and see what capabilities lay therein. It's worthwhile to note that UEFI has support for both scripts and applications. Scripts end with an ".nsh" suffix, whereas applications end with ".efi". Creating UEFI applications is a little more complicated, so I'll leave that for another day. Instead, let's see what goes into creating a simple looping "Hello World" script, like I did last week with SourcePoint in Episode 9.

The convenient thing about the MinnowBoard is that it boots out of the box into the UEFI shell. Typing "Help" at the Shell prompt displays the following list of commands:

alias	-	Displays, creates, or deletes UEFI Shell aliases.
attrib	-	Displays or modifies the attributes of files or directories.
bcfg	-	Manages the boot and driver options that are stored in NVRAM.
cd	-	Displays or changes the current directory.
cls	-	Clears standard output and optionally changes background color.
comp	-	Compares the contents of two files on a byte-for-byte basis.
connect	-	Binds a driver to a specific device and starts the driver.
cp	-	Copies one or more files or directories to another location.
date	-	Displays and sets the current date for the system.

dblk	- Displays one or more blocks from a block device.
devices	- Displays the list of devices managed by UEFI drivers.
devtree	- Displays the UEFI Driver Model compliant device tree.
dh	- Displays the device handles in the UEFI environment.
disconnect	- Disconnects one or more drivers from the specified devices.
dmem	- Displays the contents of system or device memory.
dmpstore	- Manages all UEFI variables.
drivers	- Displays the UEFI driver list.
drvcfg	- Invokes the driver configuration.
drvdiag	- Invokes the Driver Diagnostics Protocol.
echo	- Controls script file command echoing or displays a message.
edit	- Provides a full screen text editor for ASCII or UCS-2 files.
eficompress	- Compresses a file using UEFI Compression Algorithm.
efidecompress	- Decompresses a file using UEFI Decompression Algorithm.
else	- Identifies the code executed when 'if' is FALSE.
endfor	- Ends a 'for' loop.
endif	- Ends the block of a script controlled by an 'if' statement.
exit	- Exits the UEFI Shell or the current script.
for	- Starts a loop based on 'for' syntax.
getmtc	- Gets the MTC from BootServices and displays it.
goto	- Moves around the point of execution in a script.
help	- Displays the UEFI Shell command list or verbose command help.
hexedit	- Provides a full screen hex editor for files, block devices, or memory.
if	- Executes commands in specified conditions.
ifconfig	- Modifies the default IP address of the UEFI IPv4 Network Stack.
load	- Loads a UEFI driver into memory.
loadpcirom	- Loads a PCI Option ROM.
ls	- Lists the contents of a directory or file information.
map	- Displays or defines file system mappings.
memmap	- Displays the memory map maintained by the UEFI environment.
mkdir	- Creates one or more new directories.
mm	- Displays or modifies MEM/MMIO/IO/PCI/PCIE address space.
mode	- Displays or changes the console output device mode.
mv	- Moves one or more files to a destination within or between file systems.
openinfo	- Displays the protocols and agents associated with a handle.
parse	- Retrieves a value from a standard format output file.
pause	- Pauses a script and waits for an operator to press a key.
pci	- Displays a PCI device list or PCI function configuration space of a device.
ping	- Pings the target host with an IPv4 or IPv6 stack.



reconnect	-	Reconnects drivers to the specific device.
reset	-	Resets the system.
rm	-	Deletes one or more files or directories.
sermode	-	Sets serial port attributes.
set	-	Displays or modifies UEFI Shell environment variables.
setsize	-	Adjusts the size of a file.
setvar	-	Displays or modifies a UEFI variable.
shift	-	Shifts in-script parameter positions.
smbiosview	-	Displays SMBIOS information.
stall	-	Stalls the operation for a specified number of microseconds.
time	-	Displays or sets the current time for the system.
timezone	-	Displays or sets time zone information.
touch	-	Updates the filename timestamp with the current system date and time.
type	-	Sends the contents of a file to the standard output device.
unload	-	Unloads a driver image that was already loaded.
ver	-	Displays UEFI Firmware version information.
vol	-	Displays or modifies information about a disk volume.

For example, typing “ver” yields the following:

```
Shell> ver
UEFI Interactive Shell v2.1
EDK II
UEFI v2.50 (EDK II, 0x00010000)
```

It is possible to print “Hello World” to the terminal with the “echo” command:

```
Shell> echo Hello World
Hello World
Shell>
```

Interestingly, it doesn’t seem possible to use the looping/branching commands on the command line, as you can do with the shells in SourcePoint, Python, and other platforms. During my first attempt to echo “Hello World” to the screen ten times, I got the below error message:

```
Shell> for (i=0; i<10; i++) echo Hello World!
The command 'for' is incorrect outside of a script
Command Error Status: Aborted
```

At this point, I decided it was time to look at the documentation. The [UEFI Forum Specifications](#) page seemed like a good place to start, where I found the [UEFI Shell Specification Version 2.2](#) (dated January 26, 2016) being the most current. This manual was fairly difficult to plumb

through, but I finally managed to figure out that the equivalent UEFI shell script to print “Hello World” ten times looks like this:

```
echo -off
for %i run (0 10 1)
echo Hello World!
endfor
```

There’s a fairly easy-to-use editor built into the UEFI Shell, appropriately named “edit”. I decided to create a shell script named “junk.nsh” with the above program, and save it to the USB stick plugged into my MinnowBoard; then it’s just a simple matter of typing it at the command line to get my program running:

```
FS0:\> junk.nsh
FS0:\> echo -off
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
FS0:\>
```

And now for the amazing coincidence; after the above experiments, I happened to be browsing on my LinkedIn account, and noticed that one of my connections had “Liked” a new book on the UEFI Shell. Clicking on the link took me to Amazon, where I found [Harnessing the UEFI Shell: Moving the Platform Beyond DOS](#), written by Michael Rothman, Vince Zimmer, and Tim Lewis. It’s recently published (March 6, 2017) and an update to a book that was originally released in 2010, so I purchased it (warning, it’s not cheap!). An excerpt of the book’s preview in Amazon is below:

*Focusing on the use of the UEFI Shell and its recently released formal specification, this book unlocks a wide range of usage models which can help people best utilize the shell solutions. This text also expands on the obvious intended utilization of the shell and explains how it can be used in various areas such as security, networking, configuration, and other anticipated uses such as manufacturing, diagnostics, etc. Among other topics, Harnessing the UEFI Shell demonstrates how to write Shell scripts, how to write a Shell application, how to use provisioning options and*

*more. Since the Shell is also a UEFI component, the book will make clear how the two things interoperate and how both Shell developers as well as UEFI developers can dip into the other's field to further expand the power of their solutions.*

*Harnessing the UEFI Shell is authored by the three chairs of the UEFI working sub-teams, Michael Rothman (Intel, chair of the UEFI Configuration and UEFI Shell sub-teams), Vincent Zimmer (Intel, chair of the UEFI networking sub-team and security sub-team), and Tim Lewis (Insyde Software, chair of the UEFI security sub-team). This book is perfect for any OEMs that ship UEFI-based solutions (which is all of the MNCs such as IBM, Dell, HP, Apple, etc.), software developers who are focused on delivering solutions targeted to manufacturing, diagnostics, hobbyists, or stand-alone kiosk environments.*

I'll review the book in an upcoming episode of The MinnowBoard Chronicles. I'm hoping the book will give me some tips on creating an actual .efi UEFI application, after which I'll debug it using SourcePoint.

For a preview of how I'll be using SourcePoint to debug my app, have a look at our eBook, [UEFI Framework Debugging](#) (note: requires registration).

## Episode 11: Using Instruction Trace

*April 2, 2017*

I discover an incredible Trace capability that's built into the Intel Atom Bay Trail chip!

My MinnowBoard Turbot has a dual core 64-bit Intel® Atom™ “Bay Trail-I” E3826 System on a Chip (SoC) on-board. Like most Intel CPUs, it supports the standard Last Branch Record (LBR) and Branch Trace Store (BTS) trace capabilities.

LBR stores a very limited amount of trace information (typically 4 – 16 branch locations) inside model-specific registers (MSRs). It has virtually no overhead.

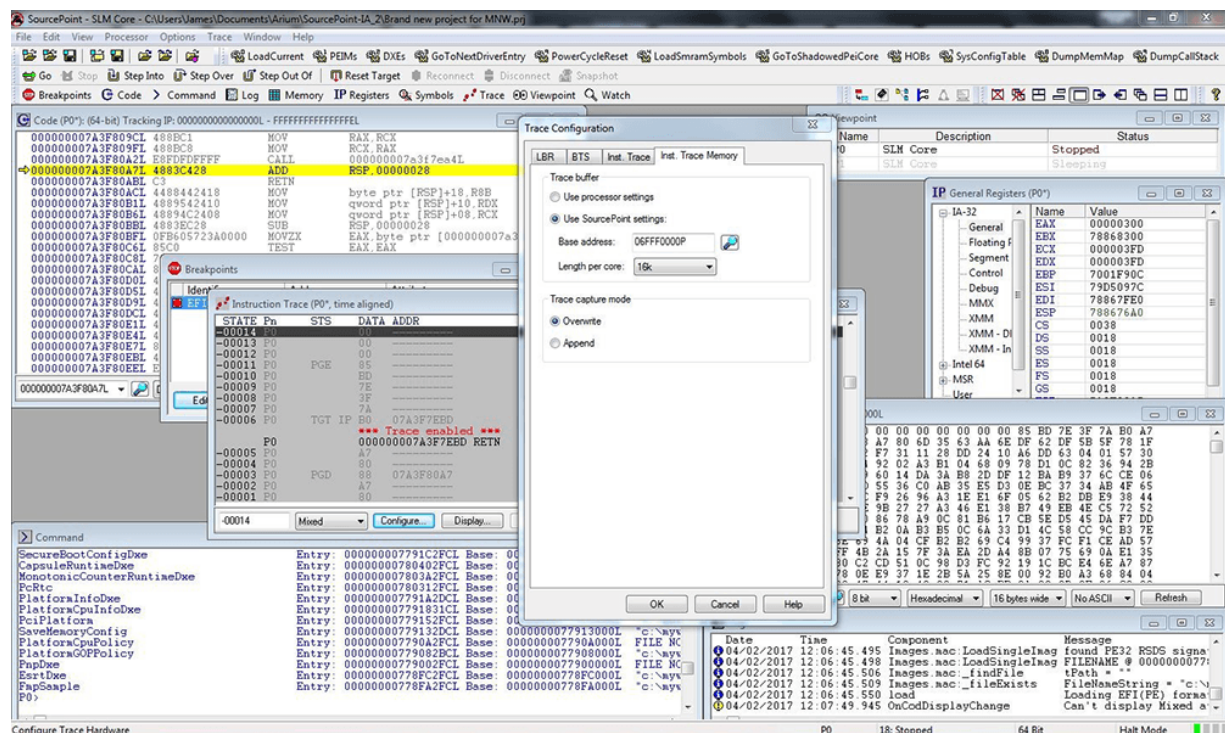
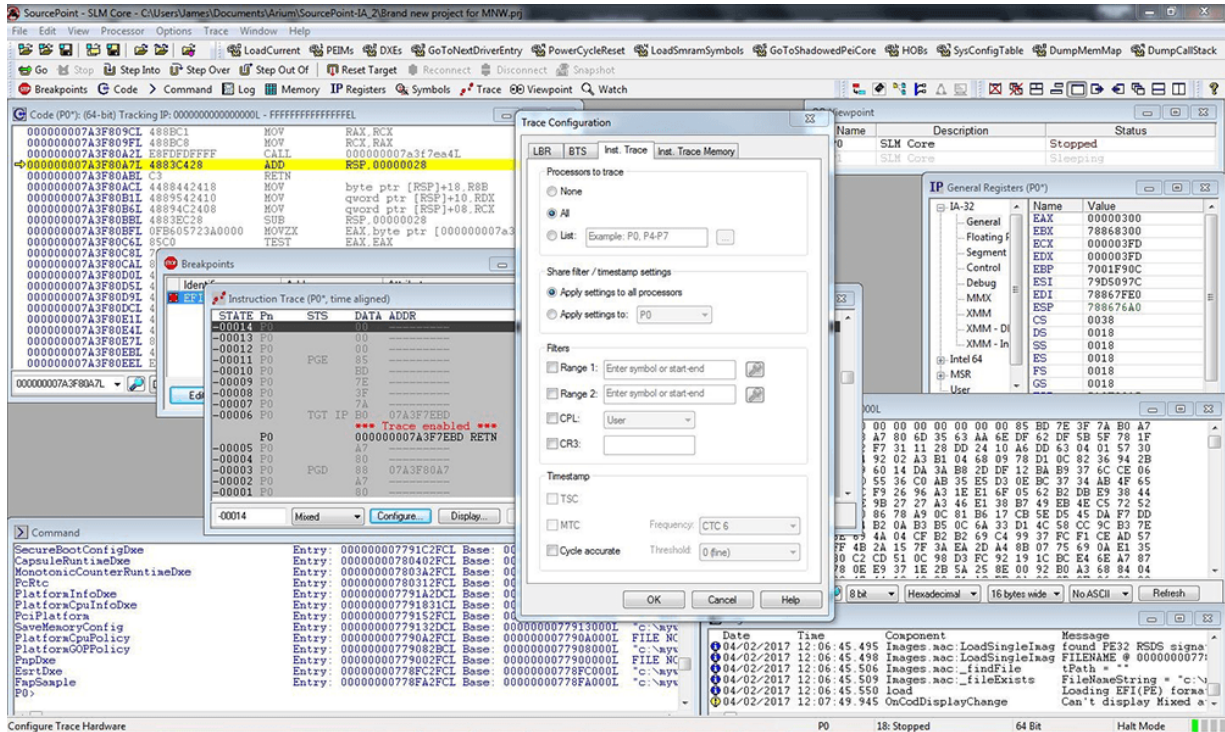
BTS uses cache-as-RAM (CAR) or system DRAM to store many more instructions and events, limited only by the amount of memory on the target system. Unlike LBR, BTS overhead impact is anywhere from 20% to 100%.

So, LBR and BTS provide limited trace capabilities, limited by trace depth and performance overhead, respectively. These constraints are one of the things that can make debugging Intel platforms very challenging. Many different approaches to debug have emerged to work around these constraints.

In my early days of tinkering with the MinnowBoard, I used LBR to demonstrate Trace on ASSET's JTAG-based debugger, [SourcePoint](#). In [Episode 6](#), there's a good screenshot of the instruction trace within a call to DebugPrint(). But the trace depth was very shallow; I wanted to go back much further in time.

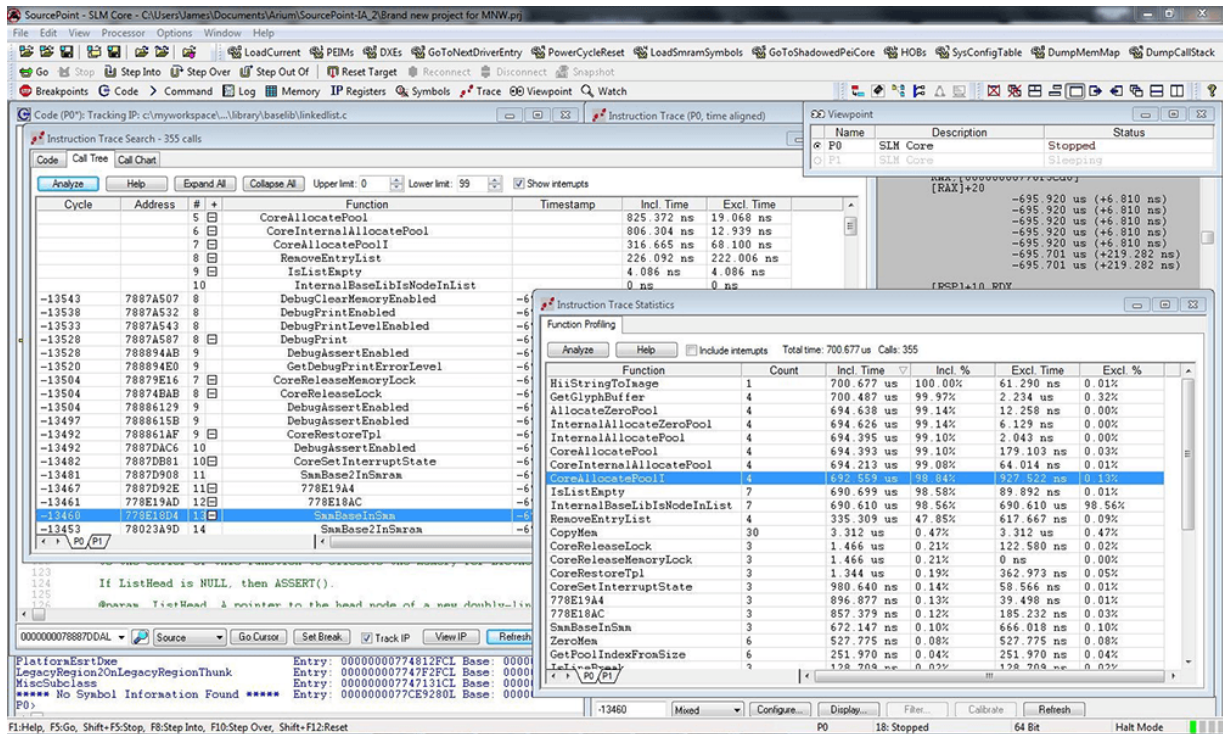
As it turns out, the Bay Trail platform supports a much more powerful trace capability, known as Instruction Trace. One of the most important features of Instruction Trace in Intel's newer ICs is that it is nearly full speed: it has no significant impact on the execution speed of the program being executed. In contrast, when using Branch Trace Messages (BTMs) with BTS (storage to memory), there is a minimum of a 60% slow down. For some code this could be much greater. Instruction Trace uses highly compressed packets and has no measurable impact on code execution. This change in execution speed can often impact whether a bug does or does not occur.

Instruction Trace is easily configured within [SourcePoint](#) by going into the Trace Configuration dialog boxes for this capability. It's a few mouse clicks to enable the feature and designate the memory location and buffer size for the trace data to be stored:

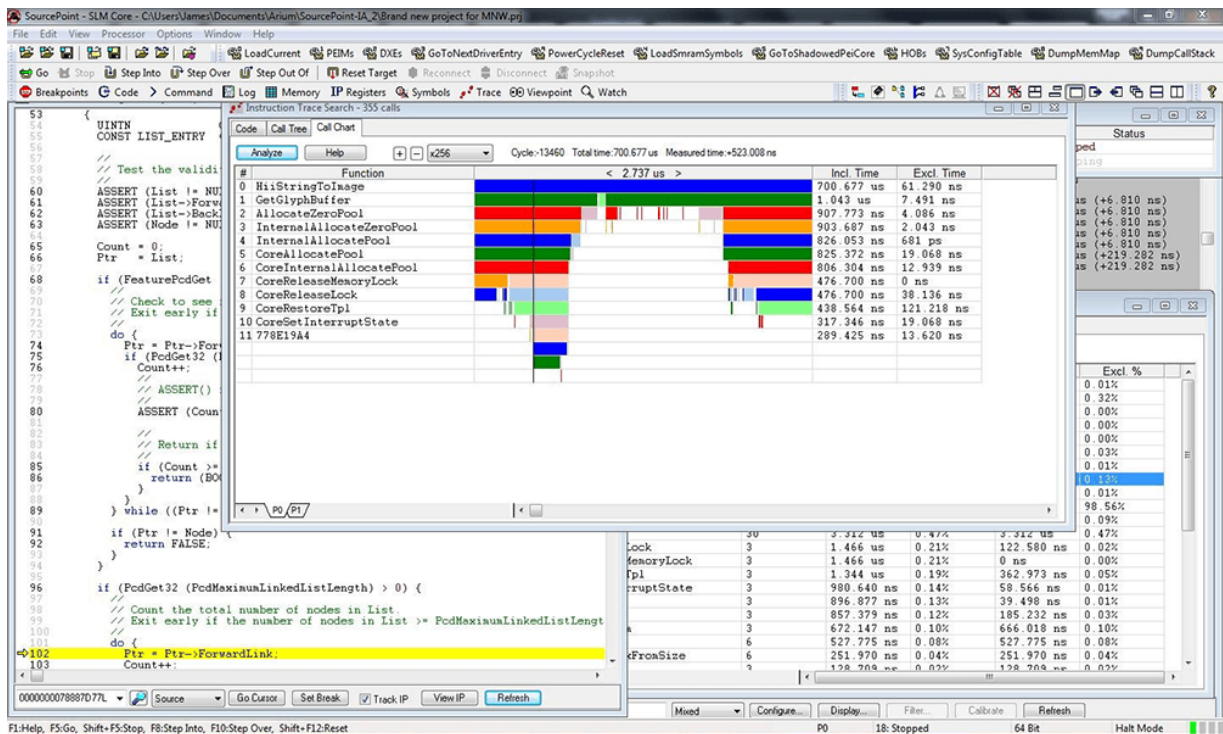




Once that was all configured, I reset the target and collected trace data while the target was running a UEFI shell script to output the configuration space of a PCI device. With SourcePoint, it's possible to see the code instruction trace in several views, all of which provide great power to the designer. Here is a view of the Call Tree and the timing statistics associated with the invocation of the stacked functions:



It's possible to use these very effectively to do a code walk-through and see where the firmware is spending its time. But, even more powerfully, the Call Chart tab in the Instruction Trace Search window provides a visual display of code execution:



The Call Graph display allows the SourcePoint user to look at large portions (or even all of) the trace buffer, and view it in a graph showing call depth. Each line in this graph represents a different function at a different point in time. Changes in color represent changes in a function. Each line moving downwards represents another level of call depth. A moveable cursor points to specific points on the timeline (the x-axis of graph). The left-hand column displays the names of the functions, at each level, at the point indicated by the cursor. And the controls above the graph allow the user to expand the graph (zoom in) at the point indicated by the cursor.

This was pretty amazing to see. I was under the impression that the older Bay Trail devices were limited to LBR and BTS trace capabilities. But with Instruction Trace, much greater debugging functionality is available. Tinkering around with this has really helped me understand the overall flow of execution of the UEFI code base.

Incidentally, an excellent treatise on Instruction Trace is available in the eBook [Intel Adds High-Speed Instruction Trace](#) (note: requires registration).

## Episode 12: Writing UEFI Applications

*April 9, 2017*

In Episode 10 of the MinnowBoard Chronicles, I created my first UEFI Shell script. Today, I “up the game” by writing an actual application in ‘C’.

In [Episode 10](#), I created a simple “Hello World” program that ran as a UEFI shell script. This is akin to writing a Basic program for beginning programmers. It was simple enough to create a short text file (with suffix .nsh) within the built-in UEFI shell editor, and then just run it from the command line. It’s analogous to Basic because the program just executes in an “interpreted” form, with no previous steps of compiling, linking, etc. So, it was very simple to get up and running.

With such simplicity, however, comes compromises. The UEFI shell scripting language is somewhat constrained; although it does have support for looping, conditional logic, and such, the language is unfamiliar, and creating a large application within it would be difficult. This is as intended: the Shell is simple and useful and lightweight, and the list of commands available to scripts using it is fairly limited.

Writing a full-fledged application or driver that executes directly within the UEFI environment is a much larger challenge, but even more rewarding. These can be written in ‘C’, so the power of that language is at your fingertips. And they can access services provided both by the UEFI and the shell, so they can do so much more.

I decided to start with the familiar “Hello World” program, written in ‘C’, as below:

```
#include <Uefi.h>
#include <Library/UefiApplicationEntryPoint.h>
#include <Library/UefiLib.h>
EFI_STATUS
EFIAPI
UefiMain (
    IN EFI_HANDLE          ImageHandle,
    IN EFI_SYSTEM_TABLE    *SystemTable
)
{
    Print(L"Hello World \n");
    return EFI_SUCCESS;
}
```

I will describe the process in more detail in an upcoming blog, but here's an outline of the steps needed to create this simple Hello World application:

1. Have the EDK II source build tree and tools available on your PC.
2. Run the edksetup script.
3. Create a new directory in your workspace and put the 'C' source file and its corresponding .inf file there.
4. Update an existing .dsc file and add in support for your MyHelloWorld .inf file. Specifically, right at the end of the [Components] section and right before the [BuildOptions] section, add in this line:

```
MyHelloWorld/MyHelloWorld.inf
```

For reference, the .inf file looks like the below:

```
## @file
# Brief Description of UEFI MyHelloWorld
#
##

[Defines]
  INF_VERSION           = 0x00010005
  BASE_NAME             = MyHelloWorld
  FILE_GUID             = 6467c5d1-d0f0-4b47-a6a4-0545624972ef
  MODULE_TYPE           = UEFI_APPLICATION
  VERSION_STRING        = 1.0
  ENTRY_POINT           = UefiMain
#
# The following information is for reference only and not required by the
# build.
#
  VALID_ARCHITECTURES   = X64
#

[Sources]
  MyHelloWorld.c

[Packages]
  MdePkg/MdePkg.dec

[LibraryClasses]
  UefiApplicationEntryPoint
  UefiLib

[Guids]

[Ppis]

[Protocols]
```

[FeaturePcd]

[Pcd]

Put these three files in the same directory, launch the Developer Command Prompt for VS2013, and type in:

```
Build -p MyHelloWorld/DuetPkgX64.dsc
```

It runs for a minute, but I got a lot of joy out of seeing (after numerous failures due to my own ineptitude) a successful compile:



After that, it was a simple matter of copying the MyHelloWorld.efi file over to a USB stick, launching the UEFI shell, and typing in MyHelloWorld at the UEFI shell prompt. I always get a kick out of seeing “Hello World!” show up on a screen.

Next week, I'll see what I can do to develop a more sophisticated application and explore what debugging tools are available for UEFI applications. In the meantime, to pay the bills, I'll direct you to more fascinating UEFI material, in particular our eBook on [UEFI Framework Debugging](#) (note: requires registration).



## Episode 13: UEFI Applications using Standard ‘C’

*April 17, 2017*

This week, I updated the firmware on the MinnowBoard Turbot to the latest release, and struggled with creating a UEFI application using standard ‘C’ functions.

I subscribe to the [@MinnowBoard](#) Twitter feed, and received a notice from [@Intel\\_Brian](#) that a new version of the firmware was available for download, and that it came highly recommended. So, as I last did in [Episode 4](#), I downloaded the full source tree and built the v0.95 image via the detailed [instructions](#) available with the release.

This time, I decided to build a Release version of the firmware, rather than the Debug version. The Debug version was quite interesting to work with, because it streamed out the BIOS printf messages to the CoolTerm application on my Mac via the MinnowBoard serial port. It was enlightening to look at some of these messages and trace them back to the source code, and see what it was doing at various stages of the boot process. But, the Debug image takes a lot longer to boot (55 seconds, as opposed to a few seconds for the Release version), so in the interest of saving time, since I was rebooting the platform a lot, I tried the Release version.

Alas, I found out that the default build options of the *Build\_IFWI.bat* command running under the *Developer Command Prompt for VS2013* yielded a build without the necessary symbols and source links. So, when I fired up [SourcePoint](#) to continue my explorations, all I could see was assembly code. This wouldn’t allow me to continue my learnings about the UEFI internals, so I fell back again to building a Debug version and reflashing the MinnowBoard with this release.

In [Episode 11](#), I built and ran a simple UEFI “Hello World” shell script. And in [Episode 12](#), I undertook the more complicated development of a real UEFI application. The latter is a precursor for building more sophisticated tools like drivers. But it involves a more complicated build process and requires a deeper understanding of the source tree file structure. Nonetheless, I succeeded in creating a simple “Hello World” application using the built-in UEFI “print” function, which is like “printf” in that it outputs the string to the standard console device.

The *Harnessing the UEFI Shell* book that I purchased refers to writing UEFI applications using standard ‘C’ member functions. It seemed to be a simple matter of doing for example a *#include*

<stdio.h> at the front end of the code, and then feeling free to use the standard ‘C’ functions that I’m familiar with. Now, that seemed like a good prospect for learning something new: I could develop a few more sophisticated applications using the built-in ‘C’ functions I’m comfortable with, and then graduate later to a deeper understanding of the UEFI-specific functions. My new *MyHelloWorld2.c* application using this approach looked like this:

```
#include <stdio.h>
int
main(int arg, char **argv)
{
    printf("Hello World!\n");
}
```

Looks familiar, right?

And the setup file for this application, *MyHelloWorld2.inf*, looks like:

```
## @file
# .inf file for MyHelloWorld2
##

[Defines]
INF_VERSION           = 0x00010005
BASE_NAME             = MyHelloWorld2
FILE_GUID             = 721f92fb-43f7-49b4-9a2a-142eac0d49ac
MODULE_TYPE           = UEFI_APPLICATION
VERSION_STRING        = 1.0
ENTRY_POINT           = UefiMain
#
# The following information is for reference only and not required by the
# build.
#
    VALID_ARCHITECTURES      = X64
#

[Sources]
    MyHelloWorld2.c

[Packages]
    StdLib/StdLib.dec
    MdePkg/MdePkg.dec

[LibraryClasses]
LibC

[Guids]

[Ppis]

[Protocols]
```

```
[FeaturePcd]
```

```
[Pcd]
```

Alas (and I realize this is the second “alas” within this article), after numerous attempts, I never could get this to successfully compile, always getting errors of the type:

Error 4000: Instance of library class [LibCType] is not found:

I suspect that I’m missing something out of the .dsc file that’s part of the build, but I’m not sure at the moment. I’ll return to this in a future exploration. Any ideas on how to proceed would be appreciated!



```
Administrator: Developer Command Prompt for VS2013

Architecture(s) = X64
Build target    = DEBUG
Toolchain      = US2012x86

Active Platform = c:\myworksspace2\MyHelloWorld2\DuetPkgX64.dsc
Flash Image Definition = c:\myworksspace2\DuetPkg\DuetPkg.fdf

Processing meta-data ...

build...
c:\myworksspace2\MyHelloWorld2\DuetPkgX64.dsc(...): error 4000: Instance of library class [LibCType] is not found
in [c:\myworksspace2\MdePkg\Library\Stdio\Stdio.inf] [X64]
consumed by module [c:\myworksspace2\MyHelloWorld2\MyHelloWorld2.inf]

- Failed -
Build end time: 03:15:02, Apr.17 2017
Build total time: 00:00:03

C:\MyWorkSpace2>build -p MyHelloWorld2\DuetPkgX64.dsc
Build environment: Windows-7-6.1.7601-SP1
Build start time: 03:20:22, Apr.17 2017

WORKSPACE      = c:\myworksspace2
ECP_SOURCE     = c:\myworksspace2\edkcompatibilitypkg
EDK_SOURCE     = c:\myworksspace2\edkcompatibilitypkg
EFI_SOURCE     = c:\myworksspace2\edkcompatibilitypkg
EDK_TOOLS_PATH = c:\myworksspace2\basetools

Architecture(s) = X64
Build target    = DEBUG
Toolchain      = US2012x86

Active Platform = c:\myworksspace2\MyHelloWorld2\DuetPkgX64.dsc
Flash Image Definition = c:\myworksspace2\DuetPkg\DuetPkg.fdf

Processing meta-data ...

build...
c:\myworksspace2\MyHelloWorld2\DuetPkgX64.dsc(...): error 4000: Instance of library class [LibCType] is not found
in [c:\myworksspace2\MyHelloWorld2\MyHelloWorld2.inf] [X64]
consumed by module [c:\myworksspace2\MyHelloWorld2\MyHelloWorld2.inf]

- Failed -
Build end time: 03:20:25, Apr.17 2017
Build total time: 00:00:03

C:\MyWorkSpace2>
```

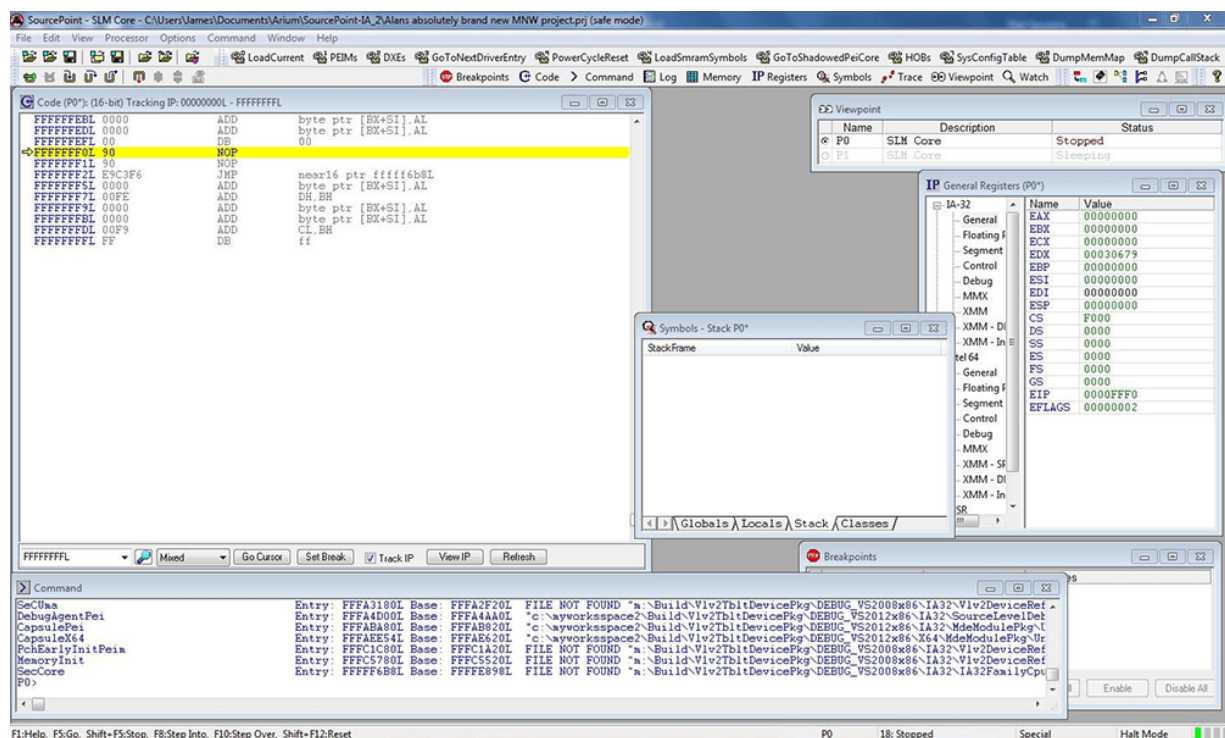
And now, a word from our sponsor: do you want to learn more about debugging UEFI applications? Check out this whitepaper here: [UEFI Framework Debugging](#) (note: requires registration).

## Episode 14: Poking around SecCore in UEFI

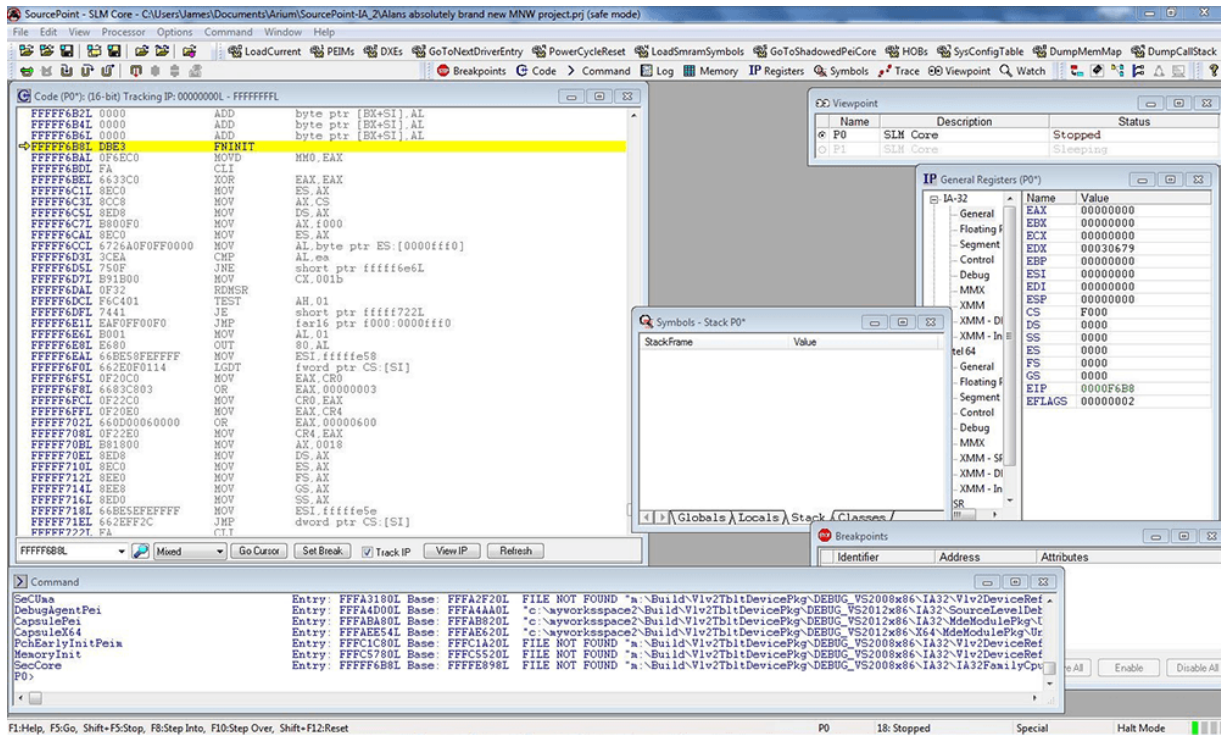
May 6, 2017

This week, I poke around the code from the reset vector, in the heart of the CPU initialization firmware, and try to reverse-engineer what is going on.

Using our JTAG hardware-assisted [SourcePoint](#) debugger, it is easy to power-cycle the MinnowBoard target and have the system stop right at the reset vector, which is at the well-known address X'FFFFFFFF'0'L. A screen shot of the Code view is as below:



The processor executes two NOP instructions, followed by a JMP to address FFFF6B8. This is where some interesting code shows up:



At the bottom of the screen, in the Command window, some of the last few lines are displayed where I've loaded the symbols into SourcePoint for the PEI modules. Note that the SecCore module has the entry address of FFFFFFF6B8, which is exactly where we are in the Code window. Note also that even though the Code window is in "Mixed" mode where we should see both source and object code, there are no symbols visible; you can see the "FILE NOT FOUND" in the Command window, which means the .pdb file is not available. This makes a lot of sense, since SEC (the UEFI Security phase) is the root of trust. So, this is part of a "binary blob", and all we have is the assembly language code to look at; but we can still peek and poke around and use SourcePoint to get some useful insights into what the platform is doing right out of reset.

The Stack Frame window is empty; this is because we are executing hand-crafted assembly code within the SPI flash, and there is no memory available for a stack yet.

The first instruction initializes the floating point unit (FPU).

The next couple of handful of instructions are the following, with my comments:

```

MOVQ MM0, EAX      // Stores the value of EAX into MM0 register
CLI                // Clear the interrupt flag in the EFLAGS register
XOR EAX, EAX        // Ensures there are all '0' stored in EAX

```

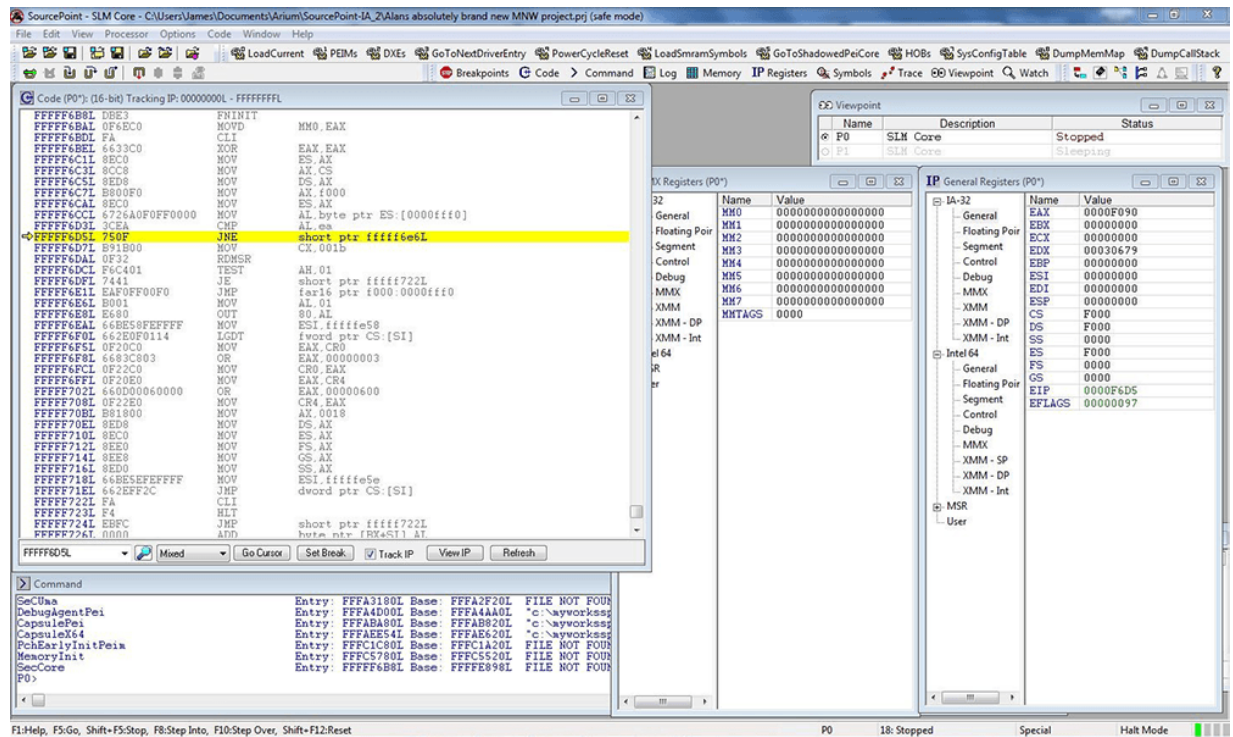


```

MOV AX, CS                // Puts the contents of the code segment register
(F000) in AX
MOV DS, AX                // Puts F000 in data segment register
MOV AX, F000              // Puts X'F000' in EAX (note it was already
there)
MOV ES, AX                // Puts X'F000' into "extra segment" register
MOV AL, byte ptr ES: [0000fff0] // Moves X'90' into the first byte of EAX
CMP AL, EA                // Compares X'EA' to the first 8 bits of EAX
(X'90')
JNE short ptr FFFFF6E6L   // If not equal (they are not), jumps to
FFFFF6E6
MOV CX, 001B              // Loads CX w/ X'1B'; address of
IA32_APIC_BASE
RDMSR                     // Reads contents of MSR into EDX:EAX
TEST AH, 01               // Compares EAX second byte (BSP FLAG) to '1'
JE short ptr FFFFF722L    // If equal, jump to FFFFF722

```

By single-stepping through the code, I can see the contents of the general-purpose registers (GPRs) changing and confirming what I think is happening. On every single step, the registers whose values are changing appear in green:

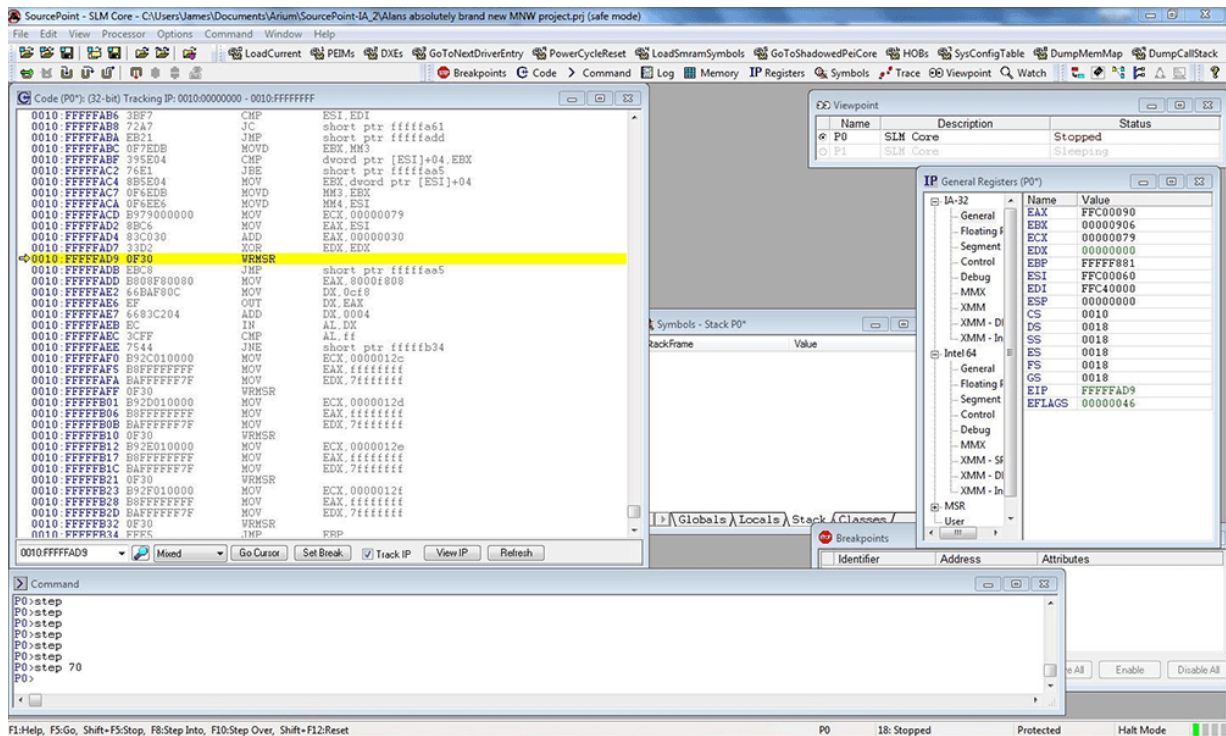


As expected, the code immediately after the JNE is not executed; instead, the CPU jumps ahead to address FFFFF6E6, and a lot more code executes.

Further single-stepping through the code and reverse-engineering it is a task for another day; for now, I decided to jump ahead and use the SourcePoint built in “step n” command, which allows me to step through “n” instructions.

After about X’80’ steps through the code, I came across something rather interesting: a write to the MSR at address X’79’, the IA32\_BIOS\_UPDT\_TRIG MSR. According to the [Intel Software Developer Manual](#), this causes a microcode update to be loaded into the processor.

But, as I tried to single-step through the MSR write instruction (WRMSR), the MinnowBoard went into never-never land; I tried repeatedly but could never successfully single-step through that WRMSR, without the platform hanging. The only way to recover the target was to power-cycle it.



I did find that I could “game the system” by changing the target of the WRMSR to a different MSR address, but that is a topic for a future episode.

This is pretty cool, isn’t it? Please feel free to make my employer happy by downloading a free eBook, such as [Hardware Assisted Debug and Trace within the Silicon](#) (note: requires (free) registration).

## Episode 15: More UEFI Application Development in ‘C’

*May 14, 2017*

In [Episode 12](#), I wrote a simple “Hello World!” application in ‘C’ using the built-in UEFI shell functions. In [Episode 13](#), I failed in an attempt to re-write that application using standard ‘C’ library functions, such as `printf()`. I’ve learned a lot since then – here’s how to write more sophisticated programs.

From [Episode 12 Writing UEFI Applications](#), I took a simple ‘C’ program and adapted it to print “Hello World” to the screen.

```
#include <Uefi.h>
#include <Library/UefiApplicationEntryPoint.h>
#include <Library/UefiLib.h>
EFI_STATUS
EFIAPI
UefiMain (
    IN EFI_HANDLE      ImageHandle,
    IN EFI_SYSTEM_TABLE *SystemTable
)
{
    Print(L"Hello World \n");
    return EFI_SUCCESS;
}
```

Note that this program differs from other simple ‘C’ programs in that the entry point is not the familiar `main(INT argc, CHAR16 **argv)` that I wanted to use to pass in a command line string. Also, it uses the UEFI shell “print” command rather than the “printf” that I am used to.

I decided to start with a program that echoes the command line to the screen, similar to the “echo” shell command.

So, one step at a time. I first wanted to learn how to modify my program to accept command line parameters and manipulate them. I found out that I needed to change the module entry point from “UefiMain” to “ShellAppMain” to pass parameters in on the command line. And, to do that, the `HelloWorld.inf` file must be updated to have `ENTRY_POINT` set to `ShellCEntryLib`, Packages must include `ShellPkg/ShellPkg.dec`, and `LibraryClasses` must include `ShellCEntryLib`. And finally, the `DuetPkgX64.dsc` file must have the path to `ShellCEntryLib` explicitly added:

*ShellCEntryLib|ShellPkg/Library/UefiShellCEntryLib/UefiShellCEntryLib.inf*

So here is a “new and improved” version of MyHelloWorld.c that takes the user input from the command line and echoes it back on the screen:

```
/**
My Hello World!
**/

#include <Uefi.h>
#include <Library/UefiApplicationEntryPoint.h>
#include <Library/UefiLib.h>

INTN
EFIAPI
ShellAppMain (
    IN UINTN Argc,
    IN CHAR16 **Argv[]
)
{
    int i;
    for (i = 1; i < Argc; i++)
        Print(L"%s ", Argv[i]);
    Print(L"\n");
    return EFI_SUCCESS;
}
```

The MyHelloWorld.inf file that is used within the build is as follows:

```
## @file
#
#
##

[Defines]
    INF_VERSION                = 0x00010006
    BASE_NAME                  = MyHelloWorld
    FILE_GUID                  = 6467c5d1-d0f0-4b47-a6a4-0545624972ef
    MODULE_TYPE                = UEFI_APPLICATION
    VERSION_STRING              = 1.0
    ENTRY_POINT                = ShellCEntryLib
#
# The following information is for reference only and not required by the
# build.
#
    VALID_ARCHITECTURES        = X64
#

[Sources]
    MyHelloWorld.c

[Packages]
    MdePkg/MdePkg.dec
    ShellPkg/ShellPkg.dec

[LibraryClasses]
    UefiApplicationEntryPoint
```

```

    UefiLib
    ShellCEntryLib

[Guids]

[Ppis]

[Protocols]

[FeaturePcd]

[Pcd]

```

And finally, the DuetPkgX64.dsc file which is the driver for the build (that is, it uses the defined source code, packages and library classes to build the application) is as follows:

```

## @file
# An EFI/Framework Emulation Platform with UEFI HII interface supported.
#
# Developer's UEFI Emulation. DUET provides an EFI/UEFI IA32/X64 environment
on legacy BIOS,
# to help developing and debugging native EFI/UEFI drivers.
#
# Copyright (c) 2010 - 2013, Intel Corporation. All rights reserved.<BR>
#
# This program and the accompanying materials
# are licensed and made available under the terms and conditions of the BSD
License
# which accompanies this distribution. The full text of the license may be
found at
# http://opensource.org/licenses/bsd-license.php
#
# THE PROGRAM IS DISTRIBUTED UNDER THE BSD LICENSE ON AN "AS IS" BASIS,
# WITHOUT WARRANTIES OR REPRESENTATIONS OF ANY KIND, EITHER EXPRESS OR
IMPLIED.
#
##

#####
###
#
# Defines Section - statements that will be processed to create a Makefile.
#
#####
###
[Defines]
    PLATFORM_NAME                = DuetPkg
    PLATFORM_GUID                 = 199E24E0-0989-42aa-87F2-611A8C397E72
    PLATFORM_VERSION              = 0.4
    DSC_SPECIFICATION             = 0x00010005
    OUTPUT_DIRECTORY              = Build/DuetPkgX64
    SUPPORTED_ARCHITECTURES       = X64
    BUILD_TARGETS                 = DEBUG
    SKUID_IDENTIFIER              = DEFAULT
    FLASH_DEFINITION               = DuetPkg/DuetPkg.fdf

```



```
#####
###
#
# Library Class section - list of all Library Classes needed by this
Platform.
#
#####
###
[LibraryClasses]
#
# Entry point
#
PeimEntryPoint|MdePkg/Library/PeimEntryPoint/PeimEntryPoint.inf
DxeCoreEntryPoint|MdePkg/Library/DxeCoreEntryPoint/DxeCoreEntryPoint.inf

UefiDriverEntryPoint|MdePkg/Library/UefiDriverEntryPoint/UefiDriverEntryPoint
.inf

UefiApplicationEntryPoint|MdePkg/Library/UefiApplicationEntryPoint/UefiApplic
ationEntryPoint.inf
#
# Basic
#
BaseLib|MdePkg/Library/BaseLib/BaseLib.inf

SynchronizationLib|MdePkg/Library/BaseSynchronizationLib/BaseSynchronizationL
ib.inf
BaseMemoryLib|MdePkg/Library/BaseMemoryLib/BaseMemoryLib.inf
PrintLib|MdePkg/Library/BasePrintLib/BasePrintLib.inf
CpuLib|MdePkg/Library/BaseCpuLib/BaseCpuLib.inf
IoLib|MdePkg/Library/BaseIoLibIntrinsic/BaseIoLibIntrinsic.inf
PciLib|MdePkg/Library/BasePciLibCf8/BasePciLibCf8.inf
PciCf8Lib|MdePkg/Library/BasePciCf8Lib/BasePciCf8Lib.inf
PciExpressLib|MdePkg/Library/BasePciExpressLib/BasePciExpressLib.inf

CacheMaintenanceLib|MdePkg/Library/BaseCacheMaintenanceLib/BaseCacheMaintenan
ceLib.inf
PeCoffLib|MdePkg/Library/BasePeCoffLib/BasePeCoffLib.inf

PeCoffGetEntryPointLib|MdePkg/Library/BasePeCoffGetEntryPointLib/BasePeCoffGe
tEntryPointLib.inf
#
# UEFI & PI
#

UefiBootServicesTableLib|MdePkg/Library/UefiBootServicesTableLib/UefiBootServ
icesTableLib.inf

UefiRuntimeServicesTableLib|MdePkg/Library/UefiRuntimeServicesTableLib/UefiRu
ntimeServicesTableLib.inf
UefiRuntimeLib|MdePkg/Library/UefiRuntimeLib/UefiRuntimeLib.inf
UefiLib|MdePkg/Library/UefiLib/UefiLib.inf

UefiHiiServicesLib|MdeModulePkg/Library/UefiHiiServicesLib/UefiHiiServicesLib
.inf
HiiLib|MdeModulePkg/Library/UefiHiiLib/UefiHiiLib.inf
DevicePathLib|MdePkg/Library/UefiDevicePathLib/UefiDevicePathLib.inf
```

```

UefiDecompressLib|MdePkg/Library/BaseUefiDecompressLib/BaseUefiDecompressLib.
inf
  DxeServicesLib|MdePkg/Library/DxeServicesLib/DxeServicesLib.inf

DxeServicesTableLib|MdePkg/Library/DxeServicesTableLib/DxeServicesTableLib.in
f
  UefiCpuLib|UefiCpuPkg/Library/BaseUefiCpuLib/BaseUefiCpuLib.inf
  ShellCEntryLib|ShellPkg/Library/UefiShellCEntryLib/UefiShellCEntryLib.inf

#
# Generic Modules
#
  UefiUsbLib|MdePkg/Library/UefiUsbLib/UefiUsbLib.inf
  UefiScsiLib|MdePkg/Library/UefiScsiLib/UefiScsiLib.inf

OemHookStatusCodeLib|MdeModulePkg/Library/OemHookStatusCodeLibNull/OemHookSta
tusCodeLibNull.inf

GenericBdsLib|IntelFrameworkModulePkg/Library/GenericBdsLib/GenericBdsLib.inf

SecurityManagementLib|MdeModulePkg/Library/DxeSecurityManagementLib/DxeSecuri
tyManagementLib.inf
  CapsuleLib|MdeModulePkg/Library/DxeCapsuleLibNull/DxeCapsuleLibNull.inf

PeCoffExtraActionLib|MdePkg/Library/BasePeCoffExtraActionLibNull/BasePeCoffEx
traActionLibNull.inf

CustomizedDisplayLib|MdeModulePkg/Library/CustomizedDisplayLib/CustomizedDisp
layLib.inf
#
# Platform
#
  PlatformBdsLib|DuetPkg/Library/DuetBdsLib/PlatformBds.inf
  TimerLib|DuetPkg/Library/DuetTimerLib/DuetTimerLib.inf
#
# Misc
#

PerformanceLib|MdePkg/Library/BasePerformanceLibNull/BasePerformanceLibNull.i
nf
  DebugAgentLib|MdeModulePkg/Library/DebugAgentLibNull/DebugAgentLibNull.inf
  PcdLib|MdePkg/Library/BasePcdLibNull/BasePcdLibNull.inf

MemoryAllocationLib|MdePkg/Library/UefiMemoryAllocationLib/UefiMemoryAllocati
onLib.inf
  HobLib|MdePkg/Library/DxeHobLib/DxeHobLib.inf

ExtractGuidedSectionLib|MdePkg/Library/DxeExtractGuidedSectionLib/DxeExtractG
uidSectionLib.inf

PlatformHookLib|MdeModulePkg/Library/BasePlatformHookLibNull/BasePlatformHook
LibNull.inf

SerialPortLib|MdeModulePkg/Library/BaseSerialPortLib16550/BaseSerialPortLib16
550.inf
  MtrrLib|UefiCpuPkg/Library/MtrrLib/MtrrLib.inf

```

```

LockBoxLib|MdeModulePkg/Library/LockBoxNullLib/LockBoxNullLib.inf

CpuExceptionHandlerLib|UefiCpuPkg/Library/CpuExceptionHandlerLib/DxeCpuExcept
ionHandlerLib.inf
LocalApicLib|UefiCpuPkg/Library/BaseXApicLib/BaseXApicLib.inf

#
# To save size, use NULL library for DebugLib and ReportStatusCodeLib.
# If need status code output, do library instance overridden as below
DxeMain.inf does
#
DebugLib|MdePkg/Library/BaseDebugLibNull/BaseDebugLibNull.inf

DebugPrintErrorLevelLib|MdePkg/Library/BaseDebugPrintErrorLevelLib/BaseDebugP
rintErrorLevelLib.inf

ReportStatusCodeLib|MdePkg/Library/BaseReportStatusCodeLibNull/BaseReportStat
usCodeLibNull.inf

[LibraryClasses.common.DXE_CORE]
HobLib|MdePkg/Library/DxeCoreHobLib/DxeCoreHobLib.inf

MemoryAllocationLib|MdeModulePkg/Library/DxeCoreMemoryAllocationLib/DxeCoreMe
moryAllocationLib.inf

#####
###
#
# Pcd Section - list of all EDK II PCD Entries defined by this Platform
#
#####
###
[PcdsFixedAtBuild]
gEfiMdePkgTokenSpaceGuid.PcdReportStatusCodePropertyMask|0x0
gEfiMdePkgTokenSpaceGuid.PcdDebugPropertyMask|0x0
gEfiMdePkgTokenSpaceGuid.PcdDebugPrintErrorLevel|0x0
gEfiMdeModulePkgTokenSpaceGuid.PcdResetOnMemoryTypeInformationChange|FALSE

[PcdsFeatureFlag]
gEfiMdeModulePkgTokenSpaceGuid.PcdTurnOffUsbLegacySupport|TRUE

#####
#####
#
# Components Section - list of the modules and components that will be
processed by compilation
#
# tools and the EDK II tools to generate PE32/PE32+/Coff
image files.
#
# Note: The EDK II DSC file is not used to specify how compiled binary images
get placed
#
# into firmware volume images. This section is just a list of modules
to compile from
#
# source into UEFI-compliant binaries.
#
# It is the FDF file that contains information on combining binary
files into firmware

```

```

#       volume images, whose concept is beyond UEFI and is described in PI
specification.
#       Binary modules do not need to be listed in this section, as they
should be
#       specified in the FDF file. For example: Shell binary
(Shell_Full.efi), FAT binary (Fat.efi),
#       Logo (Logo.bmp), and etc.
#       There may also be modules listed in this section that are not
required in the FDF file,
#       When a module listed here is excluded from FDF file, then UEFI-
compliant binary will be
#       generated for it, but the binary will not be put into any firmware
volume.
#
#####
#####
[Components]
  DuetPkg/DxeIpl/DxeIpl.inf {
    <LibraryClasses>
      #
      # If no following overridden for ReportStatusCodeLib library class,
      # All other module can *not* output debug information even they are use
not NULL library
      # instance for DebugLib and ReportStatusCodeLib
      #

ReportStatusCodeLib|MdeModulePkg/Library/DxeReportStatusCodeLib/DxeReportStat
usCodeLib.inf
  }

  MdeModulePkg/Core/Dxe/DxeMain.inf {
    #
    # Enable debug output for DxeCore module, this is a sample for how to
enable debug output
    # for a module. If need turn on debug output for other module, please
copy following overridden
    # PCD and library instance to other module's override section.
    #
    <PcdsFixedAtBuild>
      gEfiMdePkgTokenSpaceGuid.PcdReportStatusCodePropertyMask|0x07
      gEfiMdePkgTokenSpaceGuid.PcdDebugPropertyMask|0x2F
      gEfiMdePkgTokenSpaceGuid.PcdDebugPrintErrorLevel|0x80000042
    <LibraryClasses>

DebugLib|IntelFrameworkModulePkg/Library/PeiDxeDebugLibReportStatusCode/PeiDx
eDebugLibReportStatusCode.inf

ReportStatusCodeLib|DuetPkg/Library/DxeCoreReportStatusCodeLibFromHob/DxeCore
ReportStatusCodeLibFromHob.inf
  }

  MdeModulePkg/Universal/PCD/Dxe/Pcd.inf
  MdeModulePkg/Universal/WatchdogTimerDxe/WatchdogTimer.inf
  MdeModulePkg/Core/RuntimeDxe/RuntimeDxe.inf

MdeModulePkg/Universal/MonotonicCounterRuntimeDxe/MonotonicCounterRuntimeDxe.
inf

```

```

DuetPkg/FSVariable/FSVariable.inf
MdeModulePkg/Universal/CapsuleRuntimeDxe/CapsuleRuntimeDxe.inf
MdeModulePkg/Universal/MemoryTest/NullMemoryTestDxe/NullMemoryTestDxe.inf
MdeModulePkg/Universal/SecurityStubDxe/SecurityStubDxe.inf
MdeModulePkg/Universal/Console/ConPlatformDxe/ConPlatformDxe.inf
MdeModulePkg/Universal/Console/ConSplitterDxe/ConSplitterDxe.inf {
    <LibraryClasses>
        PcdLib|MdePkg/Library/DxePcdLib/DxePcdLib.inf
    }
MdeModulePkg/Universal/HiiDatabaseDxe/HiiDatabaseDxe.inf
MdeModulePkg/Universal/SetupBrowserDxe/SetupBrowserDxe.inf
MdeModulePkg/Universal/DisplayEngineDxe/DisplayEngineDxe.inf

MdeModulePkg/Universal/Console/GraphicsConsoleDxe/GraphicsConsoleDxe.inf
MdeModulePkg/Universal/Console/TerminalDxe/TerminalDxe.inf
MdeModulePkg/Universal/DevicePathDxe/DevicePathDxe.inf
MdeModulePkg/Universal/SmbiosDxe/SmbiosDxe.inf

DuetPkg/SmbiosGenDxe/SmbiosGen.inf
#DuetPkg/FvbRuntimeService/DUETFWH.inf
DuetPkg/EfiLdr/EfiLdr.inf {
    <LibraryClasses>
        DebugLib|MdePkg/Library/BaseDebugLibNull/BaseDebugLibNull.inf
}

NULL|IntelFrameworkModulePkg/Library/LzmaCustomDecompressLib/LzmaCustomDecomp
ressLib.inf
}
IntelFrameworkModulePkg/Universal/BdsDxe/BdsDxe.inf {
    <LibraryClasses>
        PcdLib|MdePkg/Library/DxePcdLib/DxePcdLib.inf
    }
MdeModulePkg/Universal/EbcDxe/EbcDxe.inf
UefiCpuPkg/CpuIo2Dxe/CpuIo2Dxe.inf
UefiCpuPkg/CpuDxe/CpuDxe.inf
PcAtChipsetPkg/8259InterruptControllerDxe/8259.inf
DuetPkg/AcpiResetDxe/Reset.inf
DuetPkg/LegacyMetronome/Metronome.inf

PcAtChipsetPkg/PcatRealTimeClockRuntimeDxe/PcatRealTimeClockRuntimeDxe.inf
PcAtChipsetPkg/8254TimerDxe/8254Timer.inf
DuetPkg/PciRootBridgeNoEnumerationDxe/PciRootBridgeNoEnumeration.inf
DuetPkg/PciBusNoEnumerationDxe/PciBusNoEnumeration.inf
IntelFrameworkModulePkg/Bus/Pci/VgaMiniPortDxe/VgaMiniPortDxe.inf
IntelFrameworkModulePkg/Universal/Console/VgaClassDxe/VgaClassDxe.inf

# IDE/AHCI Support
DuetPkg/SataControllerDxe/SataControllerDxe.inf
MdeModulePkg/Bus/Ata/AtaAtapiPassThru/AtaAtapiPassThru.inf
MdeModulePkg/Bus/Ata/AtaBusDxe/AtaBusDxe.inf
MdeModulePkg/Bus/Scsi/ScsiBusDxe/ScsiBusDxe.inf
MdeModulePkg/Bus/Scsi/ScsiDiskDxe/ScsiDiskDxe.inf

# Usb Support
MdeModulePkg/Bus/Pci/UhciDxe/UhciDxe.inf
MdeModulePkg/Bus/Pci/EhciDxe/EhciDxe.inf

```



```
MdeModulePkg/Bus/Usb/UsbBusDxe/UsbBusDxe.inf
MdeModulePkg/Bus/Usb/UsbKbDxe/UsbKbDxe.inf
MdeModulePkg/Bus/Usb/UsbMassStorageDxe/UsbMassStorageDxe.inf

# ISA Support
PcAtChipsetPkg/IsaAcpiDxe/IsaAcpi.inf
IntelFrameworkModulePkg/Bus/Isa/IsaBusDxe/IsaBusDxe.inf
IntelFrameworkModulePkg/Bus/Isa/IsaSerialDxe/IsaSerialDxe.inf
IntelFrameworkModulePkg/Bus/Isa/Ps2KeyboardDxe/Ps2keyboardDxe.inf
IntelFrameworkModulePkg/Bus/Isa/IsaFloppyDxe/IsaFloppyDxe.inf

MdeModulePkg/Universal/Disk/DiskIoDxe/DiskIoDxe.inf
MdeModulePkg/Universal/Disk/UnicodeCollation/EnglishDxe/EnglishDxe.inf
MdeModulePkg/Universal/Disk/PartitionDxe/PartitionDxe.inf

# Bios Thunk
DuetPkg/BiosVideoThunkDxe/BiosVideo.inf

#
# Sample Application
#
# MdeModulePkg/Application/HelloWorld/HelloWorld.inf

MyHelloWorld/MyHelloWorld.inf

#####

BuildOptions Section - Define the module specific tool chain flags that
could be used as the default flags for a module. These flags are
opened to any standard flags that are defined by the build
process. They can be applied for any modules or only those modules with
ne specific module style (EDK or EDKII) specified in
Components] section.

#####

BuildOptions]
MSFT:* * * CC FLAGS = /FAsc /FR$(@R).SBR
```

Now I'll go through a detailed step-by-step process description for the build.

I put the 'C' source code, MyHelloWorld.inf file and the modified DuetPkgX64.dsc file into a folder entitled MyHelloWorld within the MyWorkSpace folder.

Firstly, launch “Developer Command Prompt for VS2013”.

Navigate (using the “cd” change directory command) to the MyWorkSpace directory that contains all the build files.

Type in “edksetup”.

Type in “build -p MyHelloWorld/DuetPkgX64.dsc”.

The application builds perfectly and echoes the command line arguments out to the screen. For example, if you type:

*MyHelloWorld.efi This is a test!*

You see the “This is a test!” echoed back on the following line. Pretty cool.

A couple of interesting notes:

On the HDMI monitor I’ve hooked the MinnowBoard to, I simply see the text “This is a test!” echoed to the screen. But on the CoolTerm application I’ve got hooked into the serial port, I see the following:

```
FS0:\> MyHelloWorld.efi This is a test!
InstallProtocolInterface: 5B1B31A1-9562-11D2-8E3F-00A0C969723B 762CBC00
    PDB =
c:\myworksspace2\Build\DuetPkgX64\DEBUG_VS2012x86\X64\MyHelloWorld5\MyHelloWo
rld\DEBUG\MyHelloWorld.pdb
Loading driver at 0x00077CD1000 EntryPoint=0x00077CD12C0 MyHelloWorld.efi
InstallProtocolInterface: BC62157E-3E33-4FEC-9920-2D3B36D750DF 762BAC58
InstallProtocolInterface: 752F3136-4E16-4FDC-A22A-E5F46812F4CA 78851848
This is a test!
```

Also, I’ve noted that the compiler rejects the following:

*for (int i = 1; i > Argc; i++)*

But, rather, it wants the variable declaration to be in a distinct statement:

```
int i;
for (i = 1; i > Argc; i++)
```

I don’t know why that is. Maybe I am using an older version of Visual Studio (VS2013)? That’s something to figure out for another day.

To summarize, in this episode I’ve graduated from creating a simple ‘C’ program that printed “Hello World” to the screen, to actually taking shell command parameters and echoing those out to the terminal. It may seem like a small step, but it really helped me understand how the ‘C’

source and build files interact with each other. This should enable me to write more sophisticated code going forward, and to understand how more complex programs, like drivers, are put together.

For others who might also like to take such a self-taught journey, a good debugger is indispensable. SourcePoint is probably the best UEFI hardware-assisted tool available; read more at the product website page [SourcePoint for Intel Platforms](#).

## Episode 16: Delving into LBR Trace

*May 21, 2017*

This week, I decided to do a thorough analysis of how Last Branch Record (LBR) trace works, to see how useful it is in tracing back what might be the root cause of system failures.

LBR trace within [SourcePoint](#) displays a history of executed instructions. The last branch recording mechanism tracks not only branch instructions (like JMP, Jcc, LOOP and CALL instructions), but also other operations that cause a change in the instruction pointer (like external interrupts, traps and faults).

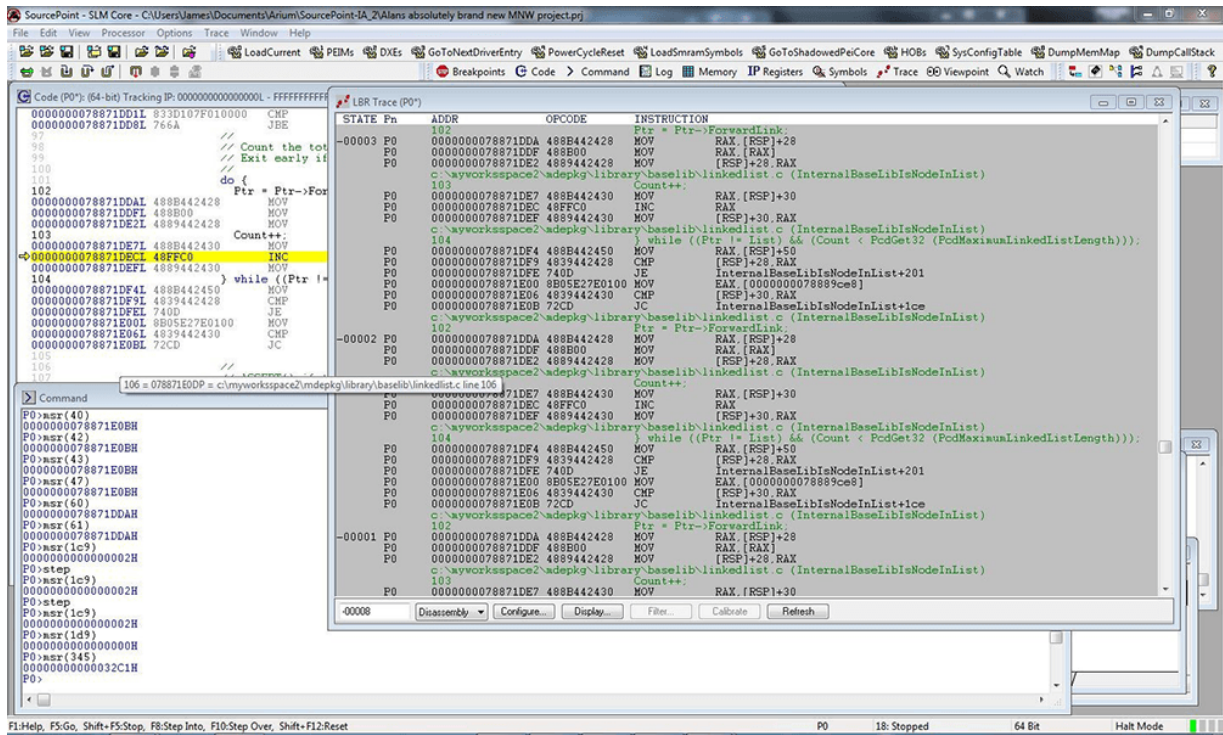
The advantage of LBR trace is it is non-intrusive. The processor can run at full speed when using LBR trace. The disadvantage of LBR trace is the limited number of LBRs available (on the MinnowBoard BayTrail-I CPU, which is based on the Intel Silvermont core architecture, there are eight). A branch record consists of a branch-from and a branch-to instruction address.

If you assume an average of 5 instructions between branches, then roughly the last 40 instructions executed are traced.

I learned all I know about how LBR works from the [Intel Software Developer Manual](#). Volume 3 deals with how to configure the platform to use LBR. In particular, there are certain registers, such as MSRs IA32\_DEBUGCTL[0] and IA32\_PERF\_CAPABILITIES[5:0], that must be used to activate LBR and define the format of the addresses defined within the LBR stack. And the source and destination instruction addresses of recent branches are contained within MSRs. For example:

```
MSR_LASTBRANCH_0_FROM_IP stores a source address.
MSR_LASTBRANCH_0_TO_IP stores a destination address.
MSR_LASTBRANCH_TOS contains a pointer to the MSR in the LBR stack that
contains the most recent branch, interrupt, or exception recorded.
```

Let's see how this works within SourcePoint. I booted up the MinnowBoard to the UEFI shell and then launched SourcePoint, turned on LBR, and then hit Run. Just as a starting effort, I typed in "pci 00 00 00 -i" into the shell, which provides a verbose display of the PCI config information for bus 0, device 0, function 0, and then halted the processor. I then used the "msr" command to examine some of the content of the LBR MSRs. Here's what I saw:



The result is fascinating and a real learning experience to delve into.

Firstly, the Code window shows that the instruction pointer is at 78871DEC which is within a “do” loop that is incrementing the value of the “Count” variable.

The LBR Trace window shows the backtrace of what’s happening. You can see the exact ‘C’ program and the function therein that’s being executed at the time the platform was halted. The actual file is at:

C:\myworksspac2\mdepkg\library\baselib\linkedlist.c

and the function we’re in is:

InternalBaseLibIsNodeInList

There’s a lot of detail displayed in the Trace window, including the line number within the source code file, and the source/symbols with associated disassembly. It’s pretty cool that I can go back to the source within the build on my PC and see what the purpose of this function is.

Here’s an excerpt:



```

C:\MyWorkspace2\MdePkg\Library\BaseLib\LinkedList.c - Notepad++ [Administrator]
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

LinkedList.c
20 By searching the List, finds the location of the Node in List. At the same time,
21 verifies the validity of this list.
22
23 If List is NULL, then ASSERT().
24 If List->ForwardLink is NULL, then ASSERT().
25 If List->backLink is NULL, then ASSERT().
26 If Node is NULL, then ASSERT().
27 If PcdVerifyNodeInList is TRUE and DoMembershipCheck is TRUE and Node
28 is in not a member of List, then return FALSE
29 If PcdMaximumLinkedListLength is not zero, and List contains more than
30 PcdMaximumLinkedListLength nodes, then ASSERT().
31
32 @param List A pointer to a node in a linked list.
33 @param Node A pointer to a node in a linked list.
34 @param VerifyNodeInList TRUE if a check should be made to see if Node is a
35 member of List. FALSE if no membership test should
36 be performed.
37
38 @retval TRUE if PcdVerifyNodeInList is FALSE
39 @retval TRUE if DoMembershipCheck is FALSE
40 @retval TRUE if PcdVerifyNodeInList is TRUE and DoMembershipCheck is TRUE
41 and Node is a member of List.
42 @retval FALSE if PcdVerifyNodeInList is TRUE and DoMembershipCheck is TRUE
43 and Node is in not a member of List.
44
45 /**
46 BOOLEAN
47 EFIAPI
48 InternalBaseLibIsNodeInList (
49 IN CONST LIST_ENTRY *List,
50 IN CONST LIST_ENTRY *Node,
51 IN BOOLEAN VerifyNodeInList

```

Having access to the comments within the source code really adds to my understanding of what the software is doing.

It's also fascinating to correlate what is on the SourcePoint screen with the contents of the LBR-related MSRs. As you can see, I ran the SourcePoint "msr()" command on many of the source and destination address MSRs. They are at 78871E0B and 78871DDA respectively. This makes sense; looking at the Trace window, at address 78871E0B is the *JC InternalBaseLibIsNodeInList* back to the top of the "do/while" loop, and 78871DDA is the *MOV RAX, [RSP]+28* that is the beginning of the first instruction (related to the 'C' line *Ptr = Ptr->ForwardLink*; that is the branch at the top of the "do/while" loop.

This particular LBR traceback isn't particularly interesting, because we're just stuck in a loop outputting information to the screen. That's why all the source and destination addresses don't change; they just repeat as far back as the limited number of LBR MSRs will allow. In my next blog, I'll break somewhere more interesting, so we'll have a more dynamic traceback. I'll also weave together what we see in SourcePoint with what we see in the source build tree to get the big picture on the value of trace.

And a final word from our sponsor: the beautiful thing about SourcePoint is that it shows the traced instructions in a meaningful context with the code that is currently executing. But, we do know that interrupts, exceptions, and other logic elements running in parallel with the mainline code can contribute to system bugs and failures. For a good eBook on how Trace can be used to track down bugs from parallel and preemptive multitasking execution on Intel-based designs, see our eBook, [Hardware Assisted Debug and Trace within the Silicon](#) (note: it's free, but requires registration).

## Episode 17: Using LBR Trace without Source Code

*May 26, 2017*

My curiosity got the better of me this week. I decided to play detective and see what I can learn from LBR trace data if I pretend I don't have access to the source code.

[In Episode 16](#), I learned about how Last Branch Record (LBR) trace can be used to look at the true flow of program execution, which may or may not be easily gleaned from just looking at the static view within the [SourcePoint](#) Code window. Having access to the source code where the system “breaks” makes it very easy to understand what is going on. But there are situations where the source code and symbols are unavailable: it is obfuscated within some confidential parts of UEFI, comes from a third-party driver, etc. Or alternatively, there are situations where the symbols are available but not the source code. Becoming a top-notch debugger sometimes means we need to roll up our sleeves and make do with what we have.

To simulate this kind of debugging experience, I decided to run a simulation with SourcePoint. I would break at a random point within DXE, with LBR Trace active. Then I would backtrace code execution by looking at the LBR MSR source and destination addresses and see what I might be able to glean.

There were a couple of preparation steps I needed to take beforehand. Firstly, I wanted to write a short SourcePoint macro that dumped all the LBR addresses, so I wouldn't have to do that by hand tediously. This macro simply looks like this:

```
define ord8 i=40
define ord8 msrvalue_from_address = 0
define ord8 msrvalue_to_address = 0
for (I = 40; I <= 47; i++) {
    msrvalue_from_address = msr(i)
    msrvalue_to_address = msr(i + 20)
    printf("%x %x %x \n", i, msrvalue_from_address, msrvalue_to_address)
    i += 1
}
```

Secondly, although the DXE modules are relocatable, I've found that from boot to boot, the entry point addresses of the DXE modules do not change. In fact, when I run the DXE macro within SourcePoint, I always get the same output, an excerpt of which is below.

```
DxeCore                                Entry: 0000000078852300L Base:
0000000078852000L
"c:\myworksspace2\Build\Vlv2TbлтDevicePkg\DEBUG_VS2012x86\X64\MdeModulePkg\Co
re\Dxe\DxeMain\DEBUG\DxeCore.efi"
```

```
PcdDxe                                Entry: 0000000077BA62FCL Base:
0000000077BA6000L
"c:\myworksspace2\Build\Vlv2TbлтDevicePkg\DEBUG_VS2012x86\X64\MdeModulePkg\Un
iversal\PCD\Dxe\Pcd\DEBUG\PcdDxe.efi"
```

```
ReportStatusCodeRouterRuntimeDxe      Entry: 00000000780A62FCL Base:
00000000780A6000L
"c:\myworksspace2\Build\Vlv2TbлтDevicePkg\DEBUG_VS2012x86\X64\MdeModulePkg\Un
iversal\ReportStatusCodeRouter\RuntimeDxe\ReportStatusCodeRouterRuntimeDxe\DE
BUG\ReportStatusCodeRouterRuntimeDxe.efi"
```

```
StatusCodeHandlerRuntimeDxe           Entry: 000000007809E2FCL Base:
000000007809E000L
"c:\myworksspace2\Build\Vlv2TbлтDevicePkg\DEBUG_VS2012x86\X64\MdeModulePkg\Un
iversal\StatusCodeHandler\RuntimeDxe\StatusCodeHandlerRuntimeDxe\DEBUG\Status
CodeHandlerRuntimeDxe.efi"
```

So, I want to put the addresses of the DXE module entry points in a table, so I can easily map the addresses I get out of LBR trace to a piece of software. An excerpt of this is below:

Module	Entry	Base
PchReset	7807B03C	7807A000
SmmControl	7808603C	78085000
RuntimeDxe	780902FC	78090000
CpuIoDxe	780982FC	78098000
StatusCodeHandlerRuntimeDxe	7809E2FC	78090000
ReportStatusCodeRouterRuntimeDxe	780A62FC	780A6000
DxeCore	78852300	78852000

So, here we go. I boot the MinnowBoard and then halt it while in the DXE phase, as it waits to get to the shell. Here's the output of my macro:

```
40 78098e3f 780986d0
41 780986d5 780986e3
42 780986f7 78098704
43 7809872f 7809873d
44 78098743 78098a05
45 78098a10 78098a1c
46 78098a5a 78098a93
47 78098aa7 78099180
```

Also, typing in `msr(1C9)` gives `x'7'`, and given that the lowest significant 3 bits of the TOS Pointer MSR (`MSR_LASTBRANCH_TOS`, address `1C9H`) contains a pointer to the MSR in the

LBR stack that contains the most recent branch, interrupt, or exception recorded, the last “from\_address” is x’78098aa7’ and the last “to\_address” is x’78099180’.

Pretty cool, huh? Even without source code, based upon the addresses, you can see that the software is likely somewhere within CpuIoDxe.

And, if you want to cheat a little, taking the last branch “to\_address”, x’78098190’, is an offset X’e84’ (or decimal 3,716) from the entry point of CpuIoDxe, address x’780982fc’. You can look at the CpuIoDxe.map file in the source build to estimate what function within CpuIoDxe we might be in. As it turns out, it’s somewhere within MmioWrite64():

Address	Public by Value
0001:00000bb8	InternalMathDivRemU64x32
0001:00000c10	MmioRead8
0001:00000c38	MmioWrite8
0001:00000c64	MmioRead16
0001:00000cc8	MmioWrite16
0001:00000d30	MmioRead32
0001:00000d90	MmioWrite32
0001:00000df4	MmioRead64
0001:00000e58	MmioWrite64
0001:00000ec0	IoRead8

For a great video on what the SourcePoint GUI looks like, take a peek at our webpage here:

[SourcePoint for Intel](#).

A good eBook on trace features is at [Intel Trace Hub](#) (note: it’s free, but requires registration).



## Episode 18: Reverse-Engineering Code Execution

*June 11, 2017*

In my last article, I used Last Branch Record (LBR) Trace to manually capture UEFI program flow source and destination addresses. This week, I look at the associated instruction opcodes and mnemonics and try to figure out what is going on.

In last week's MinnowBoard Chronicles, [Episode 17: Using LBR Trace without Source Code](#), we stopped somewhere in DXE and dumped all of the branch-from and branch-to instruction address pairs, up to a maximum of 8 within the Intel Silvermont architecture.

Why is this interesting? Well, there may be an event you want to debug on an Intel platform where the only “breadcrumbs” are the last branch addresses of code execution immediately prior. As we learned in [Episode 16](#), these are captured within some model-specific registers (MSRs) dedicated to this purpose. On the MinnowBoard, based upon an Intel BayTrail-I processor (that has Silvermont cores), these source/destination pairs are MSR addresses x'40' through x'47' and x'60' through x'67'.

Recall that the LBR recording mechanism tracks not only branch instructions (like JMP, Jcc, LOOP and CALL instructions), but also other operations that cause a change in the instruction pointer (like external interrupts, traps and faults). It has the advantage of being active soon after reset if needed; whereas other tracing mechanisms, such as Branch Trace Store (BTS) and Intel Processor Trace, require system memory to be initialized. LBR is the most “low-level” of tracing features on Intel silicon, so to speak.

To follow up on [Episode 17](#), this week I again halted the system and put it into probe mode within DXE. Then I ran my LBR MSR dump macro to see the branch-from and branch-to address pairs. The address traceback looked like this:

```
From:          To:
7785b0e4 7785b0c4
7785b0d1 77855c10
77855c29 7785b0d6
7785b0e4 7785b0c4
7785b0d1 77855c10
7785b0e4 7785b0c4
7785b0d1 77855c10
```

77855c29 7785b0d6

We also know from the [Intel Architectures Software Developers Manuals](#) that the TOS Pointer MSR (MSR\_LASTBRANCH\_TOS, address x'1C9') contains a pointer to the MSR in the LBR stack that contains the most recent branch, interrupt, or exception recorded. In this case, using the SourcePoint *msr(1C9)* command, I found that it equaled x'04', so the last “from\_address” is x'7785b0d1' and the last “to\_address” is x'77855c10' from above. Also, I could see from the SourcePoint Code window that the instruction pointer is at x'77855c1f'. And then the branch traceback goes backwards from there.

Going into the SourcePoint Code window, with its built-in disassembler, we can easily see the assembly language code flow as we go backwards in time. There's a lot of code here. Let's look at the individual “chunks” of code sorted by the above “from” and “to” addresses:

```

MSR_LASTBRANCH_0_FROM_IP
000000007785B0E4L 74DE          JE          short ptr 000000007785b0c4L
MSR_LASTBRANCH_0_TO_IP
000000007785B0C4L 0FB7057D320000 MOVZX         EAX,word ptr
[000000007785e348]
000000007785B0CBL 83C005         ADD          EAX,00000005
000000007785B0CEL 4863C8         MOVSXD        RCX,EAX
000000007785B0D1L E83AABFFFF     CALL          0000000077855c10L

MSR_LASTBRANCH_1_FROM_IP:
000000007785B0D1L E83AABFFFF     CALL          0000000077855c10L
MSR_LASTBRANCH_1_TO_IP:
0000000077855C10L 48894C2408     MOV          qword ptr [RSP]+08,RCX
0000000077855C15L 4883EC18       SUB          RSP,00000018
0000000077855C19L 0FB7542420     MOVZX        EDX,word ptr [RSP]+20
0000000077855C1EL EC             IN           AL,DX
0000000077855C1FL 880424         MOV          byte ptr [RSP],AL
0000000077855C22L 8A0424         MOV          AL,byte ptr [RSP]
0000000077855C25L 4883C418       ADD          RSP,00000018
0000000077855C29L C3             RETN

MSR_LASTBRANCH_2_FROM_IP:
0000000077855C29L C3             RET
MSR_LASTBRANCH_2_TO_IP
000000007785B0D6L 88442428       MOV          byte ptr [RSP]+28,AL
000000007785B0DAL 0FB6442428     MOVZX        EAX,byte ptr [RSP]+28
000000007785B0DFL 83E020         AND          EAX,00000020
000000007785B0E2L 85C0          TEST         EAX,EAX
000000007785B0E4L 74DE          JE          short ptr 000000007785b0c4L

MSR_LASTBRANCH_3_FROM_IP
000000007785B0E4L 74DE          JE          short ptr 000000007785b0c4L
MSR_LASTBRANCH_3_TO_IP
000000007785B0C4L 0FB7057D320000 MOVZX         EAX,word ptr
[000000007785e348]

```

000000007785B0CBL 83C005	ADD	EAX,00000005
000000007785B0CEL 4863C8	MOVSXD	RCX,EAX
000000007785B0D1L E83AABFFFF	CALL	0000000077855c10L
MSR_LASTBRANCH_4_FROM_IP		
000000007785B0D1L E83AABFFFF	CALL	0000000077855c10L
MSR_LASTBRANCH_4_TO_IP		
0000000077855C10L 48894C2408	MOV	qword ptr [RSP]+08,RCX
0000000077855C15L 4883EC18	SUB	RSP,00000018
0000000077855C19L 0FB7542420	MOVZX	EDX,word ptr [RSP]+20
0000000077855C1EL EC	IN	AL,DX
0000000077855C1FL 880424	MOV	byte ptr [RSP],AL //
Instruction pointer!		
0000000077855C22L 8A0424	MOV	AL,byte ptr [RSP]
0000000077855C25L 4883C418	ADD	RSP,00000018
0000000077855C29L C3	RETN	
MSR_LASTBRANCH_5_FROM_IP		
000000007785B0E4L 74DE	JE	short ptr 000000007785b0c4L
MSR_LASTBRANCH_5_TO_IP		
000000007785B0C4L 0FB7057D320000	MOVZX	EAX,word ptr
[000000007785e348]		
000000007785B0CBL 83C005	ADD	EAX,00000005
000000007785B0CEL 4863C8	MOVSXD	RCX,EAX
000000007785B0D1L E83AABFFFF	CALL	0000000077855c10L
MSR_LASTBRANCH_6_FROM_IP		
000000007785B0D1L E83AABFFFF	CALL	0000000077855c10L
MSR_LASTBRANCH_6_TO_IP		
0000000077855C10L 48894C2408	MOV	qword ptr [RSP]+08,RCX
0000000077855C15L 4883EC18	SUB	RSP,00000018
0000000077855C19L 0FB7542420	MOVZX	EDX,word ptr [RSP]+20
0000000077855C1EL EC	IN	AL,DX
0000000077855C1FL 880424	MOV	byte ptr [RSP],AL
0000000077855C22L 8A0424	MOV	AL,byte ptr [RSP]
0000000077855C25L 4883C418	ADD	RSP,00000018
0000000077855C29L C3	RETN	
MSR_LASTBRANCH_7_FROM_IP		
0000000077855C29L C3	RETN	
MSR_LASTBRANCH_7_TO_IP		
000000007785B0D6L 88442428	MOV	byte ptr [RSP]+28,AL
000000007785B0DAL 0FB6442428	MOVZX	EAX,byte ptr [RSP]+28
000000007785B0DFL 83E020	AND	EAX,00000020
000000007785B0E2L 85C0	TEST	EAX,EAX
000000007785B0E4L 74DE	JE	short ptr 000000007785b0c4L

Seeing the actual flow of the code without any source code or automated tools is challenging, but it is do-able. I wanted to be able to simulate a debugging scenario whereby you might have access to extracting MSR data (as with a [ScanWorks Embedded Diagnostics \(SED\)](#) On-Target Diagnostic (OTD)), as opposed to a benchtop debugger (such as [SourcePoint](#)). Knowing where the instruction pointer is, and working backwards, the actual code flow is reconstructed below:

000000007785B0E4L 74DE	JE	short ptr 000000007785b0c4L
000000007785B0C4L 0FB7057D320000 [000000007785e348]	MOVZX	EAX,word ptr
000000007785B0CBL 83C005	ADD	EAX,00000005
000000007785B0CEL 4863C8	MOVSXD	RCX,EAX
000000007785B0D1L E83AABFFFF	CALL	0000000077855c10L
0000000077855C10L 48894C2408	MOV	qword ptr [RSP]+08,RCX
0000000077855C15L 4883EC18	SUB	RSP,00000018
0000000077855C19L 0FB7542420	MOVZX	EDX,word ptr [RSP]+20
0000000077855C1EL EC	IN	AL,DX
0000000077855C1FL 880424	MOV	byte ptr [RSP],AL
0000000077855C22L 8A0424	MOV	AL,byte ptr [RSP]
0000000077855C25L 4883C418	ADD	RSP,00000018
0000000077855C29L C3	RETN	
000000007785B0D6L 88442428	MOV	byte ptr [RSP]+28,AL
000000007785B0DAL 0FB6442428	MOVZX	EAX,byte ptr [RSP]+28
000000007785B0DFL 83E020	AND	EAX,00000020
000000007785B0E2L 85C0	TEST	EAX,EAX
000000007785B0E4L 74DE	JE	short ptr 000000007785b0c4L
000000007785B0C4L 0FB7057D320000 [000000007785e348]	MOVZX	EAX,word ptr
000000007785B0CBL 83C005	ADD	EAX,00000005
000000007785B0CEL 4863C8	MOVSXD	RCX,EAX
000000007785B0D1L E83AABFFFF	CALL	0000000077855c10L
0000000077855C10L 48894C2408	MOV	qword ptr [RSP]+08,RCX
0000000077855C15L 4883EC18	SUB	RSP,00000018
0000000077855C19L 0FB7542420	MOVZX	EDX,word ptr [RSP]+20
0000000077855C1EL EC	IN	AL,DX
0000000077855C1FL 880424	MOV	byte ptr [RSP],AL
0000000077855C22L 8A0424	MOV	AL,byte ptr [RSP]
0000000077855C25L 4883C418	ADD	RSP,00000018
0000000077855C29L C3	RETN	
000000007785B0D6L 88442428	MOV	byte ptr [RSP]+28,AL
000000007785B0DAL 0FB6442428	MOVZX	EAX,byte ptr [RSP]+28
000000007785B0DFL 83E020	AND	EAX,00000020
000000007785B0E2L 85C0	TEST	EAX,EAX
000000007785B0E4L 74DE	JE	short ptr 000000007785b0c4L
000000007785B0C4L 0FB7057D320000 [000000007785e348]	MOVZX	EAX,word ptr
000000007785B0CBL 83C005	ADD	EAX,00000005
000000007785B0CEL 4863C8	MOVSXD	RCX,EAX
000000007785B0D1L E83AABFFFF	CALL	0000000077855c10L
0000000077855C10L 48894C2408	MOV	qword ptr [RSP]+08,RCX
0000000077855C15L 4883EC18	SUB	RSP,00000018
0000000077855C19L 0FB7542420	MOVZX	EDX,word ptr [RSP]+20
0000000077855C1EL EC	IN	AL,DX
0000000077855C1FL 880424	MOV	byte ptr [RSP],AL

This is a little better. The location of the instruction pointer is at the bottom and highlighted. I've put spaces between the "cycles" associated with branches to make the code more readable. You can see how powerful Trace is, because it goes backwards in time – as opposed to purely run-control, which stops at an event and allows you to single-step forward in time. The dynamic flow of the code is visible, and the direction taken by the conditional branches gives you a sense of the program logic; for example, the two instructions:

```
TEST EAX, EAX
JE short ptr 000000007785b0c4L
```

Yield a jump to address x'7785b0c4' if the outcome of the TEST instruction yields a zero flag of one (ZF = 1) within the EFLAGS register. The only way that the zero flag will be set by TEST EAX, EAX is if the contents of the EAX register is zero. So, you can see that the jump actually happens, without explicitly having knowledge of or access to the contents of the EAX register. This is often the case if you are using SED for backtracing program flow prior to a catastrophic event, such as perhaps a CATERR or IERR.

By going back to the source build for the MinnowBoard, I note that the code I'm in is within PchInitDxe. And I don't have the source code for that; it's part of one of the binary blobs within the build. All I have are associated files with suffixes .efi, .depex, .inf and .pdb. What should I do next? Maybe acquire a copy of [IDA Pro](#) to help me decompile the code? So much to learn, so little time...

Of course, I wouldn't even have gotten this far without easy access to [SourcePoint](#). Register for our [UEFI Framework Debugging eBook](#) to learn more about JTAG-assisted debug and trace.



## Episode 19: The Yocto Project

July 16, 2017

After working with UEFI over the last 18 episodes of the MinnowBoard Chronicles, I've decided to install Linux on my MinnowBoard. It is turning out to be harder than I thought.

Installing Ubuntu 16.04.1 LTS on the MinnowBoard Turbot is actually pretty easy. There is a tutorial available online at <https://MinnowBoard.org/tutorials/installing-ubuntu-16.04-on-MinnowBoardmax>, and it is fairly clear and easy to understand and follow. But, never one to be satisfied with doing the obvious, I decided to do my own Linux build for the MinnowBoard using the [Yocto Project](#). I'm using this as a "training exercise" for doing a complete OpenBMC build on the new [Portwell Neptune Alpha](#) board I've got:



I intend to document my OpenBMC work within a forthcoming series of The Neptune Alpha Chronicles, but, first, I wanted to get comfortable with the Yocto Project first.

What is the Yocto Project? It's described on its website as:

*The Yocto Project is an open source collaboration project that provides templates, tools and methods to help you create custom Linux-based systems for embedded products regardless of the hardware architecture. It was founded in 2010 as a collaboration among many hardware*

*manufacturers, open-source operating systems vendors, and electronics companies to bring some order to the chaos of embedded Linux development.*

*Why use the Yocto Project? It's a complete embedded Linux development environment with tools, metadata, and documentation - everything you need. The free tools are easy to get started with, powerful to work with (including emulation environments, debuggers, an Application Toolkit Generator, etc.) and they allow projects to be carried forward over time without causing you to lose optimizations and investments made during the project's prototype phase. The Yocto Project fosters community adoption of this open source technology allowing its users to focus on their specific product features and development.*

In other words, Yocto will allow me to build a complete open source Linux image for the MinnowBoard, and then ultimately for the Neptune Alpha. That sounds like fun!

A good place to start for this little effort of mine is at the [Yocto Quick Start Guide](#). In fact, the Quick Start Guide has an example specifically targeted for the MinnowBoard, which is ideal.

I quickly realized that I had some work to do to get the build host minimum configuration set up:

*A build host with a minimum of 50 Gbytes of free disk space that is running a supported Linux distribution (i.e. recent releases of Fedora, openSUSE, CentOS, Debian, or Ubuntu).*

Not having a Linux box at home (remember, I'm not really an engineer; just a salesperson with a keen interest in technology, so I lack high-powered hardware!), it was time to set up a Linux VM on the pokey dual-core Windows desktop I have at home. Spending many hours on this (that may be a subject of a future blog by itself), I finally got it working thanks to directions from [Linux Fundamentals](#) by Paul Cobbaut, and the [VirtualBox User Manual](#) from Oracle.

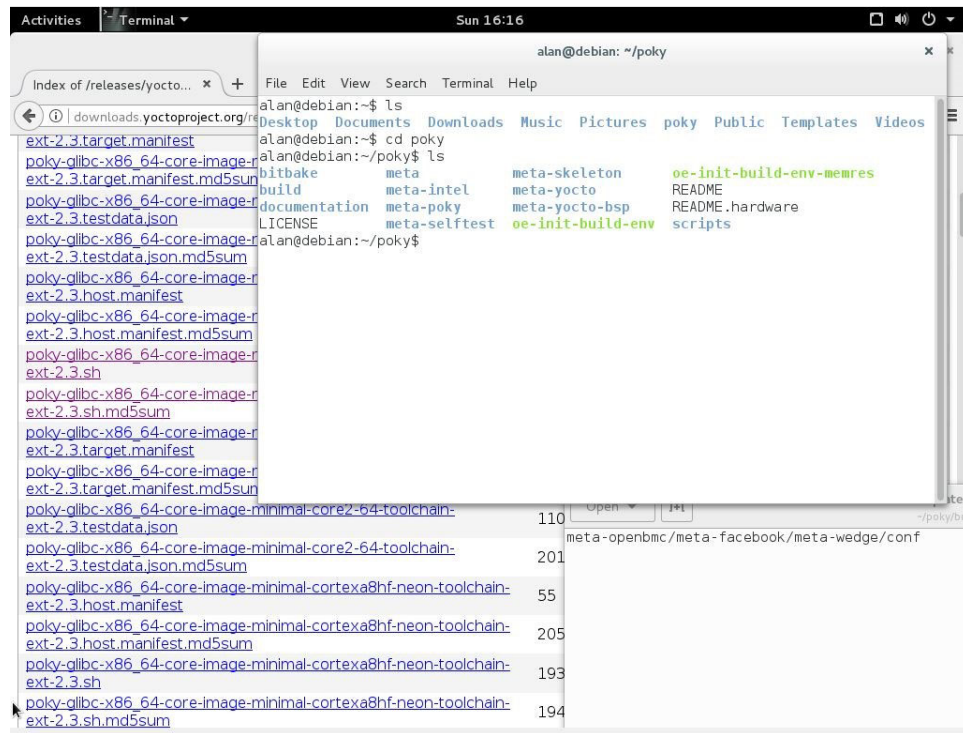
Following the Yocto Quick Start Guide, I first need to install lots of new packages, and then clone the poky repository and check out the latest Yocto Project Release (as of the time of this writing, 2.3):

```
$ sudo apt-get install gawk wget git-core diffstat unzip texinfo gcc-multilib \
\
build-essential chrpath socat cpio python python3 python3-pip python3-pexpect \
\
xz-utils debianutils iputils-ping libssl1.2-dev xterm
```

```

$ git clone git://git.yoctoproject.org/poky
Cloning into 'poky'...
remote: Counting objects: 361782, done.
remote: Compressing objects: 100% (87100/87100), done.
remote: Total 361782 (delta 268619), reused 361439 (delta 268277)
Receiving objects: 100% (361782/361782), 131.94 MiB | 6.88 MiB/s, done.
Resolving deltas: 100% (268619/268619), done.
Checking connectivity... done.
$ git checkout pyro

```



That's as far as I got this week. Next week, I'll do the image build, maybe try it out on the QEMU emulator, and then install it into my MinnowBoard.

And now, a word from my sponsor: I eventually plan to debug this with ASSET's [SourcePoint](#) JTAG hardware-assisted debugger. And, my interest in OpenBMC stems from our ScanWorks Embedded Diagnostics remote JTAG run-control solution for hyperscale platforms. For more information, read our [technical overview](#) (note: requires registration).

## Episode 20: Building and Installing Linux

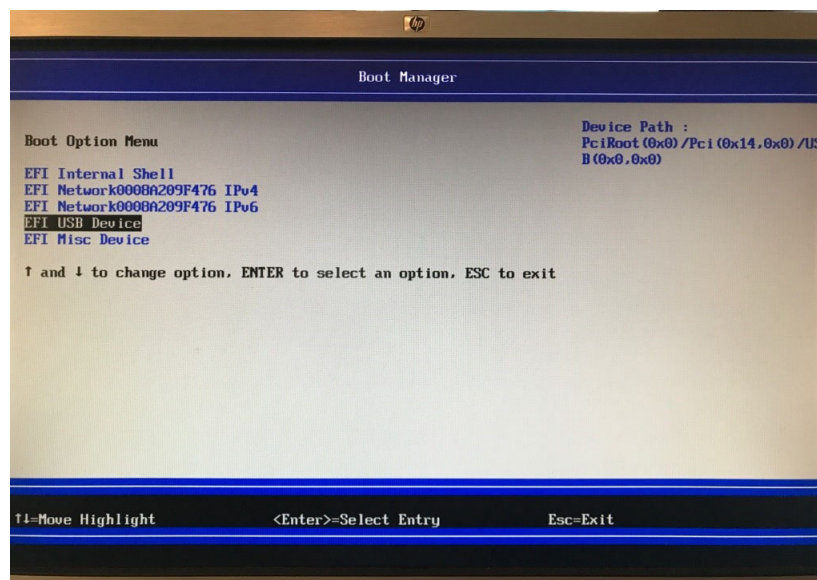
*July 25, 2017*

I have a little secret to share: even though I may make it look easy, getting Linux onto my MinnowBoard is hard, hard work.

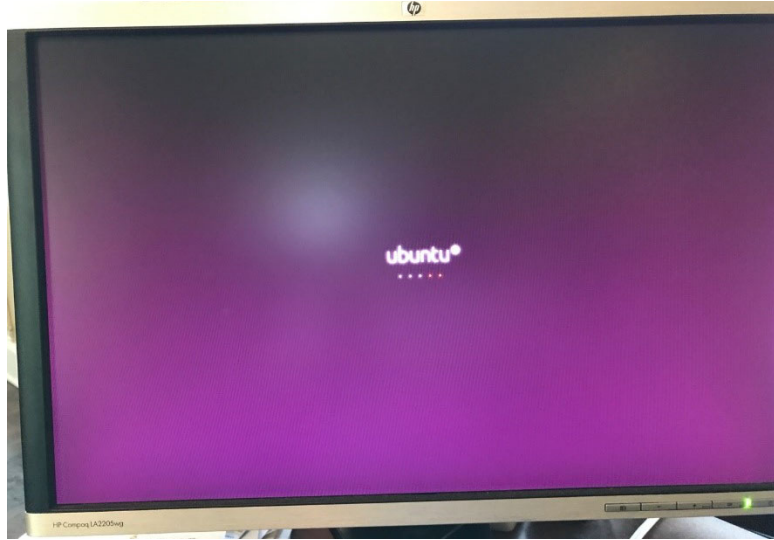
In [Episode 19](#) of the MinnowBoard Chronicles, I mentioned that doing a Yocto image build from scratch for the MinnowBoard was at the top of my to-do list. As a matter of fact, after spending a few hours unsuccessfully at this task, I decided to return to the basics: follow the tutorial at <https://MinnowBoard.org/tutorials/installing-ubuntu-16.04-on-MinnowBoardmax> first to install Ubuntu, then return to the more complex task.

Unfortunately, somewhere in the middle of all this, my HP USB keyboard stopped working on the MinnowBoard. At first it worked intermittently; then it stopped working entirely. I used CoolTerm on my Mac to do some rudimentary command line entries over the serial connection, but as soon as I exited the UEFI shell and went into the Boot Options screen, I became stuck (the arrow keys did not work).

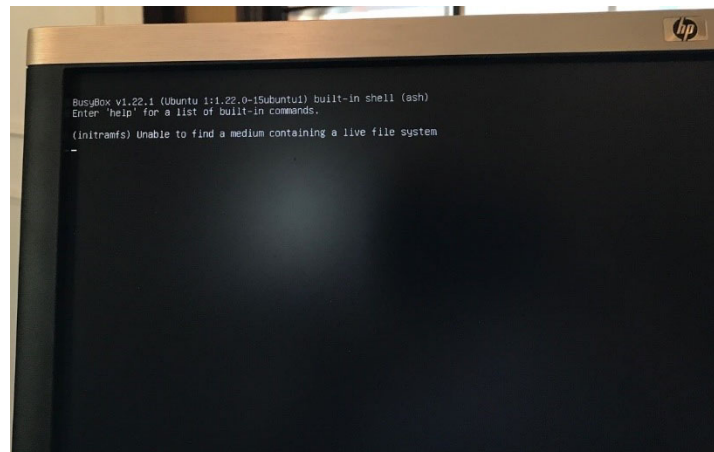
I tried a lot of different things, including looking at the serial output of the debug log with CoolTerm (to try to trace back to the UEFI code that was failing to initialize with the keyboard), but eventually, I gave up and bought a new keyboard. Eureka! That did the trick. I got to the Boot Manager screen, and selected the EFI USB Device:



Then the Ubuntu install began, with the dots marching across the screen:



But, then, boom! I got an error message from initramfs:



No matter what I did, I continued to get this error. I tried two different USB sticks and still the problem persists. So, I decided to put this activity on hold for now, and return to my goal of using Yocto to build an image, run it on the QEMU emulator, and then flash it into my MinnowBoard. I'll certainly get back to this later – it's a necessary step in my learning process.

As I mentioned in Episode 19, I am using a VM on a slow Windows PC at home. But, following the instructions in the [Yocto Quick Start Guide](#), all seemed to be going according to plan....until I ran out of memory on my hard drive! So, I have to do some cleanup before I can try this again. Stay tuned for Episode 21!



For those who may learn from my mistakes, below is the complete console file from my efforts building the image. I almost got there! And at the bottom is the error message displayed on the screen by the VM, warning of my out-of-memory condition. At the very bottom is a good graphic of my feelings after doing all of this work.

I can't wait to get this finished so I can begin debugging with [SourcePoint](#)! In particular, I'll be wanting to debug the [ASPEED AST25xx](#), once I'm done.

```
alan@debian:~$ git clone git://git.yoctoproject.org/poky
Cloning into 'poky'...
remote: Counting objects: 371109, done.
remote: Compressing objects: 100% (89063/89063), done.
remote: Total 371109 (delta 275824), reused 370786 (delta 275514)
Receiving objects: 100% (371109/371109), 134.20 MiB | 3.02 MiB/s, done.
Resolving deltas: 100% (275824/275824), done.
Checking connectivity... done.
Checking out files: 100% (5301/5301), done.

alan@debian:~$ ls
Desktop  Documents  Downloads  Music  Pictures  poky  Public  Templates  Videos

alan@debian:~$ cd poky

alan@debian:~/poky$ ls
bitbake          meta          meta-skeleton  oe-init-build-env  README.hardware
documentation    meta-poky     meta-yocto     oe-init-build-env-memres  README.LSB
LICENSE          meta-selftest meta-yocto-bsp  README              scripts

alan@debian:~/poky$ git checkout pyro
Branch pyro set up to track remote branch pyro from origin.
Switched to a new branch 'pyro'

alan@debian:~/poky$ ls
bitbake          meta          meta-skeleton  oe-init-build-env  README.hardware
documentation    meta-poky     meta-yocto     oe-init-build-env-memres  scripts
LICENSE          meta-selftest meta-yocto-bsp  README

alan@debian:~/poky$ git checkout -b pyro origin/pyro
fatal: A branch named 'pyro' already exists.

alan@debian:~/poky$ cd ~/poky

alan@debian:~/poky$ git checkout -b pyro origin/pyro
fatal: A branch named 'pyro' already exists.

alan@debian:~/poky$ ls -l
total 68
drwxr-xr-x  6 alan alan  4096 Jul  5 03:52 bitbake
drwxr-xr-x 14 alan alan  4096 Jul  5 03:52 documentation
-rw-r--r--  1 alan alan   515 Jul  5 03:52 LICENSE
drwxr-xr-x 20 alan alan  4096 Jul  5 03:52 meta
drwxr-xr-x  5 alan alan  4096 Jul  5 03:52 meta-poky
drwxr-xr-x  8 alan alan  4096 Jul  5 03:52 meta-selftest
drwxr-xr-x  7 alan alan  4096 Jul  5 03:52 meta-skeleton
drwxr-xr-x  3 alan alan  4096 Jul  5 03:52 meta-yocto
drwxr-xr-x  9 alan alan  4096 Jul  5 03:52 meta-yocto-bsp
-rwxr-xr-x  1 alan alan  2121 Jul  5 03:52 oe-init-build-env
-rwxr-xr-x  1 alan alan  2559 Jul  5 03:52 oe-init-build-env-memres
-rw-r--r--  1 alan alan  2467 Jul  5 03:52 README
-rw-r--r--  1 alan alan 14836 Jul  5 03:52 README.hardware
drwxr-xr-x  8 alan alan  4096 Jul  5 03:52 scripts

alan@debian:~/poky$ source oe-init-build-env
You had no conf/local.conf file. This configuration file has therefore been
created for you with some default values. You may wish to edit it to, for
example, select a different MACHINE (target hardware). See conf/local.conf
for more information as common configuration options are commented.

You had no conf/bblayers.conf file. This configuration file has therefore been
created for you with some default values. To add additional metadata layers
into your configuration please add entries to conf/bblayers.conf.
```



The Yocto Project has extensive documentation about OE including a reference manual which can be found at:  
<http://yoctoproject.org/documentation>

For more information about OpenEmbedded see their website:  
<http://www.openembedded.org/>

### Shell environment set up for builds. ###

You can now run 'bitbake <target>'

Common targets are:  
 core-image-minimal  
 core-image-sato  
 meta-toolchain  
 meta-ide-support

You can also run generated qemu images with a command like 'runqemu qemux86'

```
alan@debian:~/poky/build$ ls -l
total 4
drwxr-xr-x 2 alan alan 4096 Jul  5 03:55 conf
```

```
alan@debian:~/poky/build$ cd conf
```

```
alan@debian:~/poky/build/conf$ ls
bblayers.conf  local.conf  templateconf.cfg
```

```
alan@debian:~/poky/build/conf$ ls -l
total 20
-rw-r--r-- 1 alan alan  280 Jul  5 03:55 bblayers.conf
-rw-r--r-- 1 alan alan 10293 Jul  5 03:55 local.conf
-rw-r--r-- 1 alan alan   15 Jul  5 03:55 templateconf.cfg
```

```
alan@debian:~/poky/build$ bitbake core-image-sato
WARNING: /home/alan/poky/meta/recipes-core/images/core-image-tiny-initramfs.bb: Exception during
build_dependencies for create_shar
WARNING: /home/alan/poky/meta/recipes-core/images/core-image-tiny-initramfs.bb: Error during finalise of
/home/alan/poky/meta/recipes-core/images/core-image-tiny-initramfs.bb
ERROR: ExpansionError during parsing /home/alan/poky/meta/recipes-core/images/core-image-tiny-initramfs.bb
Traceback (most recent call last):
bb.data_smart.ExpansionError: Failure expanding variable create_shar, expression was # copy in the template
shar extractor script
cp /home/alan/poky/meta/files/toolchain-shar-extract.sh /home/alan/poky/build/tmp/work/qemux86-poky-
linux/core-image-tiny-initramfs/1.0-r0/x86_64-deploy-core-image-tiny-initramfs-populate-sdk/poky-glibc-x86_64-
core-image-tiny-initramfs-i586-toolchain-2.3.1.sh
```

```
rm -f /home/alan/poky/build/tmp/work/qemux86-poky-linux/core-image-tiny-initramfs/1.0-
r0/temp/pre_install_command /home/alan/poky/build/tmp/work/qemux86-poky-linux/core-image-tiny-initramfs/1.0-
r0/temp/post_install_command
```

```
if [ 1 -eq 1 ] ; then
cp /home/alan/poky/meta/files/toolchain-shar-relocate.sh
/home/alan/poky/build/tmp/work/qemux86-poky-linux/core-image-tiny-initramfs/1.0-r0/temp/post_install_command
fi
cat << "EOF" >> /home/alan/poky/build/tmp/work/qemux86-poky-linux/core-image-tiny-initramfs/1.0-
r0/temp/pre_install_command
```

EOF

```
cat << "EOF" >> /home/alan/poky/build/tmp/work/qemux86-poky-linux/core-image-tiny-initramfs/1.0-
r0/temp/post_install_command
```

EOF

```
sed -i -e '/@SDK_PRE_INSTALL_COMMAND@/r /home/alan/poky/build/tmp/work/qemux86-poky-linux/core-image-
tiny-initramfs/1.0-r0/temp/pre_install_command' \
-e '/@SDK_POST_INSTALL_COMMAND@/r /home/alan/poky/build/tmp/work/qemux86-poky-linux/core-
image-tiny-initramfs/1.0-r0/temp/post_install_command' \
/home/alan/poky/build/tmp/work/qemux86-poky-linux/core-image-tiny-initramfs/1.0-r0/x86_64-
deploy-core-image-tiny-initramfs-populate-sdk/poky-glibc-x86_64-core-image-tiny-initramfs-i586-toolchain-
2.3.1.sh
```

```
# substitute variables
sed -i -e 's#@SDK_ARCH#@x86_64#g' \
-e 's#@SDKPATH#@/opt/poky/2.3.1#g' \
-e 's#@SDKEXTPATH#@~/poky_sdk#g' \
-e 's#@OLDEST_KERNEL#@2.6.32#g' \
-e 's#@REAL_MULTIMACH_TARGET_SYS#@i586-poky-linux#g' \
-e 's#@SDK_TITLE@#${@d.getVar("SDK_TITLE").replace('&','\&')}#g' \
-e 's#@SDK_VERSION#@2.3.1#g' \
```

```

-e '@SDK_PRE_INSTALL_COMMAND@/d' \
-e '@SDK_POST_INSTALL_COMMAND@/d' \
-e 's#@SDK_GCC_VER@${oe.utils.host_gcc_version(d)}#g' \
/home/alan/poky/build/tmp/work/qemux86-poky-linux/core-image-tiny-initramfs/1.0-r0/x86_64-
deploy-core-image-tiny-initramfs-populate-sdk/poky-glibc-x86_64-core-image-tiny-initramfs-i586-toolchain-
2.3.1.sh

# add execution permission
chmod +x /home/alan/poky/build/tmp/work/qemux86-poky-linux/core-image-tiny-initramfs/1.0-r0/x86_64-
deploy-core-image-tiny-initramfs-populate-sdk/poky-glibc-x86_64-core-image-tiny-initramfs-i586-toolchain-
2.3.1.sh

# append the SDK tarball
cat /home/alan/poky/build/tmp/work/qemux86-poky-linux/core-image-tiny-initramfs/1.0-r0/x86_64-deploy-
core-image-tiny-initramfs-populate-sdk/poky-glibc-x86_64-core-image-tiny-initramfs-i586-toolchain-2.3.1.tar.xz
>> /home/alan/poky/build/tmp/work/qemux86-poky-linux/core-image-tiny-initramfs/1.0-r0/x86_64-deploy-core-image-
tiny-initramfs-populate-sdk/poky-glibc-x86_64-core-image-tiny-initramfs-i586-toolchain-2.3.1.sh

# delete the old tarball, we don't need it anymore
rm /home/alan/poky/build/tmp/work/qemux86-poky-linux/core-image-tiny-initramfs/1.0-r0/x86_64-deploy-
core-image-tiny-initramfs-populate-sdk/poky-glibc-x86_64-core-image-tiny-initramfs-i586-toolchain-2.3.1.tar.xz
which triggered exception OSError: [Errno 12] Cannot allocate memory

```

Summary: There were 2 WARNING messages shown.

Summary: There was 1 ERROR message shown, returning a non-zero exit code.

alan@debian:~/poky/build\$ bitbake core-image-sato

Loading cache: 100% |#####| Time: 0:00:02

Loaded 922 entries from dependency cache.

Parsing recipes: 100% |#####| Time: 0:01:36

Parsing of 830 .bb files complete (583 cached, 247 parsed). 1299 targets, 48 skipped, 0 masked, 0 errors.

NOTE: Resolving any missing task queue dependencies

Build Configuration:

```

BB_VERSION      = "1.34.0"
BUILD_SYS       = "x86_64-linux"
NATIVELSBSTRING = "debian-8"
TARGET_SYS      = "i586-poky-linux"
MACHINE         = "qemux86"
DISTRO          = "poky"
DISTRO_VERSION  = "2.3.1"
TUNE_FEATURES   = "m32 i586"
TARGET_FPU      = ""
meta
meta-poky
meta-yocto-bsp  = "pyro:0920b28c93632ed53e1d50c24f260f9359fcc150"

```

NOTE: Fetching univariate binary shim from [http://downloads.yoctoproject.org/releases/univariate/1.6/x86\\_64-nativesdk-libc.tar.bz2](http://downloads.yoctoproject.org/releases/univariate/1.6/x86_64-nativesdk-libc.tar.bz2);

sha256sum=2b4fffa308d9f19e0742a1a404ff42495fb50c165e5ca0458cedca157372691a

--2017-07-05 04:19:24-- [http://downloads.yoctoproject.org/releases/univariate/1.6/x86\\_64-nativesdk-libc.tar.bz2](http://downloads.yoctoproject.org/releases/univariate/1.6/x86_64-nativesdk-libc.tar.bz2)

Resolving downloads.yoctoproject.org (downloads.yoctoproject.org)... 198.145.20.127

Connecting to downloads.yoctoproject.org (downloads.yoctoproject.org)|198.145.20.127|:80... connected.

HTTP request sent, awaiting response... 200 OK

Length: 2535308 (2.4M) [application/octet-stream]

Saving to:

,/home/alan/poky/build/downloads/univariate/2b4fffa308d9f19e0742a1a404ff42495fb50c165e5ca0458cedca157372691a/x86\_64-nativesdk-libc.tar.bz2, saved

2017-07-05 04:19:28 (584 KB/s) -

,/home/alan/poky/build/downloads/univariate/2b4fffa308d9f19e0742a1a404ff42495fb50c165e5ca0458cedca157372691a/x86\_64-nativesdk-libc.tar.bz2, saved [2535308/2535308]

Initialising tasks: 100% |#####| Time: 0:01:02

NOTE: Executing SetScene Tasks

NOTE: Executing RunQueue Tasks

WARNING: The free space of /home/alan/poky/build/sstate-cache (rootfs) is running low (0.991GB left)

ERROR: No new tasks can be executed since the disk space monitor action is "STOPTASKS"!

WARNING: The free space of /home/alan/poky/build/downloads (rootfs) is running low (0.990GB left)

ERROR: No new tasks can be executed since the disk space monitor action is "STOPTASKS"!

WARNING: The free space of /home/alan/poky/build/tmp (rootfs) is running low (0.990GB left)

ERROR: No new tasks can be executed since the disk space monitor action is "STOPTASKS"!

NOTE: Tasks Summary: Attempted 102 tasks of which 0 didn't need to be rerun and all succeeded.

Summary: There were 3 WARNING messages shown.

Summary: There were 3 ERROR messages shown, returning a non-zero exit code.

alan@debian:~/poky/build\$ ls

bitbake.lock cache conf downloads sstate-cache tmp

alan@debian:~/poky/build\$ gedit local.conf

\*\* (gedit:12624): WARNING \*\*: Error when getting information for file '/home/alan/poky/build/local.conf': No such file or directory

```

alan@debian:~/poky/build$ ls -l
total 20
-rw-r--r-- 1 alan alan 0 Jul 5 04:17 bitbake.lock
drwxr-xr-x 2 alan alan 4096 Jul 5 04:20 cache
drwxr-xr-x 2 alan alan 4096 Jul 5 03:59 conf
drwxr-xr-x 4 alan alan 4096 Jul 5 04:57 downloads
drwxr-xr-x 43 alan alan 4096 Jul 5 04:58 sstate-cache
drwxr-xr-x 12 alan alan 4096 Jul 5 04:55 tmp
alan@debian:~/poky/build$ cd conf
alan@debian:~/poky/build/conf$ ls
bblayers.conf local.conf sanity_info templateconf.cfg
alan@debian:~/poky/build/conf$ gedit local.conf
alan@debian:~/poky/build/conf$ cd ..
alan@debian:~/poky/build$ ls
bitbake.lock cache conf downloads sstate-cache tmp
alan@debian:~/poky/build$ bitbake core-image-sato
WARNING: /home/alan/poky/meta/recipes-core/ovmf/ovmf-shell-image.bb: Exception during build_dependencies for
create_shar
WARNING: /home/alan/poky/meta/recipes-core/ovmf/ovmf-shell-image.bb: Error during finalise of
/home/alan/poky/meta/recipes-core/ovmf/ovmf-shell-image.bb
ERROR: ExpansionError during parsing /home/alan/poky/meta/recipes-core/ovmf/ovmf-shell-image.bb
Traceback (most recent call last):
bb.data_smart.ExpansionError: Failure expanding variable create_shar, expression was # copy in the template
shar extractor script
cp /home/alan/poky/meta/files/toolchain-shar-extract.sh /home/alan/poky/build/tmp/work/qemux86-poky-
linux/ovmf-shell-image/1.0-r0/x86_64-deploy-ovmf-shell-image-populate-sdk/poky-glibc-x86_64-ovmf-shell-image-
i586-toolchain-2.3.1.sh

rm -f /home/alan/poky/build/tmp/work/qemux86-poky-linux/ovmf-shell-image/1.0-
r0/temp/pre_install_command /home/alan/poky/build/tmp/work/qemux86-poky-linux/ovmf-shell-image/1.0-
r0/temp/post_install_command

if [ 1 -eq 1 ] ; then
cp /home/alan/poky/meta/files/toolchain-shar-relocate.sh
/home/alan/poky/build/tmp/work/qemux86-poky-linux/ovmf-shell-image/1.0-r0/temp/post_install_command
fi
cat << "EOF" >> /home/alan/poky/build/tmp/work/qemux86-poky-linux/ovmf-shell-image/1.0-
r0/temp/pre_install_command

EOF

cat << "EOF" >> /home/alan/poky/build/tmp/work/qemux86-poky-linux/ovmf-shell-image/1.0-
r0/temp/post_install_command

EOF

sed -i -e '/@SDK_PRE_INSTALL_COMMAND@/r /home/alan/poky/build/tmp/work/qemux86-poky-linux/ovmf-shell-
image/1.0-r0/temp/pre_install_command' \
-e '/@SDK_POST_INSTALL_COMMAND@/r /home/alan/poky/build/tmp/work/qemux86-poky-linux/ovmf-
shell-image/1.0-r0/temp/post_install_command' \
/home/alan/poky/build/tmp/work/qemux86-poky-linux/ovmf-shell-image/1.0-r0/x86_64-deploy-ovmf-
shell-image-populate-sdk/poky-glibc-x86_64-ovmf-shell-image-i586-toolchain-2.3.1.sh

# substitute variables
sed -i -e 's#@SDK_ARCH@#x86_64#g' \
-e 's#@SDKPATH@#/opt/poky/2.3.1#g' \
-e 's#@SDKEXTPATH@#~/poky_sdk#g' \
-e 's#@OLDEST_KERNEL@#2.6.32#g' \
-e 's#@REAL_MULTIMACH_TARGET_SYS@#i586-poky-linux#g' \
-e 's#@SDK_TITLE@#{@d.getVar("SDK_TITLE").replace('&', '\&')}#g' \
-e 's#@SDK_VERSION@#2.3.1#g' \
-e '/@SDK_PRE_INSTALL_COMMAND@d' \
-e '/@SDK_POST_INSTALL_COMMAND@d' \
-e 's#@SDK_GCC_VER@#{@oe.utils.host_gcc_version(d)}#g' \
/home/alan/poky/build/tmp/work/qemux86-poky-linux/ovmf-shell-image/1.0-r0/x86_64-deploy-ovmf-
shell-image-populate-sdk/poky-glibc-x86_64-ovmf-shell-image-i586-toolchain-2.3.1.sh

# add execution permission
chmod +x /home/alan/poky/build/tmp/work/qemux86-poky-linux/ovmf-shell-image/1.0-r0/x86_64-deploy-ovmf-
shell-image-populate-sdk/poky-glibc-x86_64-ovmf-shell-image-i586-toolchain-2.3.1.sh

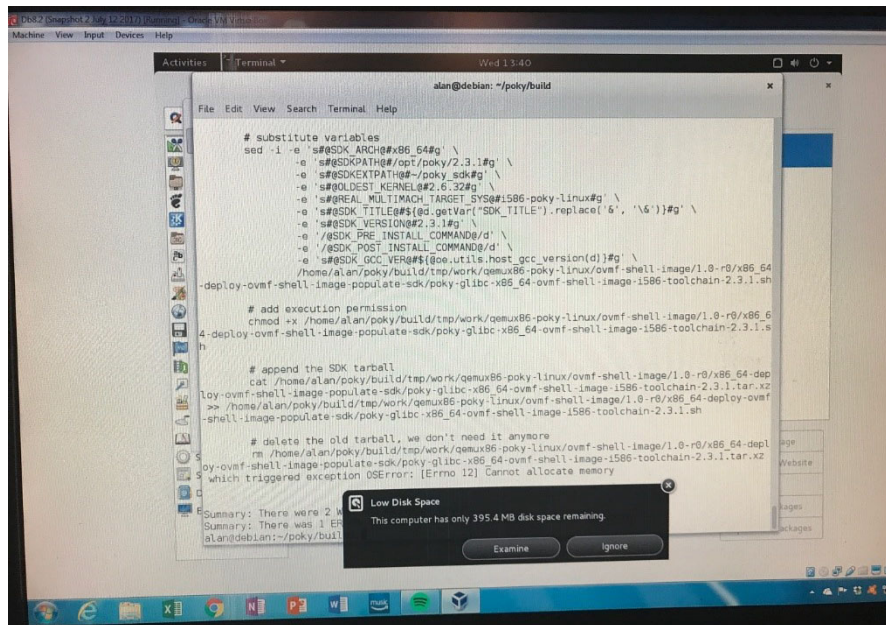
# append the SDK tarball
cat /home/alan/poky/build/tmp/work/qemux86-poky-linux/ovmf-shell-image/1.0-r0/x86_64-deploy-ovmf-
shell-image-populate-sdk/poky-glibc-x86_64-ovmf-shell-image-i586-toolchain-2.3.1.tar.xz >>
/home/alan/poky/build/tmp/work/qemux86-poky-linux/ovmf-shell-image/1.0-r0/x86_64-deploy-ovmf-shell-image-
populate-sdk/poky-glibc-x86_64-ovmf-shell-image-i586-toolchain-2.3.1.sh

# delete the old tarball, we don't need it anymore
rm /home/alan/poky/build/tmp/work/qemux86-poky-linux/ovmf-shell-image/1.0-r0/x86_64-deploy-ovmf-shell-
image-populate-sdk/poky-glibc-x86_64-ovmf-shell-image-i586-toolchain-2.3.1.tar.xz
which triggered exception OSError: [Errno 12] Cannot allocate memory

Summary: There were 2 WARNING messages shown.

```

Summary: There was 1 ERROR message shown, returning a non-zero exit code.



```

# substitute variables
sed -i -e 's#@SDK_ARCH#@x86_64#g' \
-e 's#@SDK_PATH#@opt/poky/2.3.1#g' \
-e 's#@SDK_EXTPATH#@~/poky_sdk#g' \
-e 's#@OLDEST_KERNEL#@2.6.32#g' \
-e 's#@REAL_MUIMACH_TARGET_SYS#@i586-poky-linux#g' \
-e 's#@SDK_TITLE#@$(sed.getVar("SDK_TITLE").replace(' ', '\ '))#g' \
-e 's#@SDK_VERSION#@2.3.1#g' \
-e '/@SDK_PRE_INSTALL_COMMAND@d/' \
-e '/@SDK_POST_INSTALL_COMMAND@d/' \
-e 's#@SDK_QCC_VERSION#@$(sed.getVar("QCC_VERSION").replace(' ', '\ '))#g' \
/home/alan/poky/build/tmp/work/qemu86-poky-linux/ovmf-shell-image/1.0-r0/x86_64
deploy-ovmf-shell-image-populate-sdk/poky-glibc-x86_64-ovmf-shell-image-i586-toolchain-2.3.1.sh

# add execution permission
chmod +x /home/alan/poky/build/tmp/work/qemu86-poky-linux/ovmf-shell-image/1.0-r0/x86_64
4-deploy-ovmf-shell-image-populate-sdk/poky-glibc-x86_64-ovmf-shell-image-i586-toolchain-2.3.1.sh

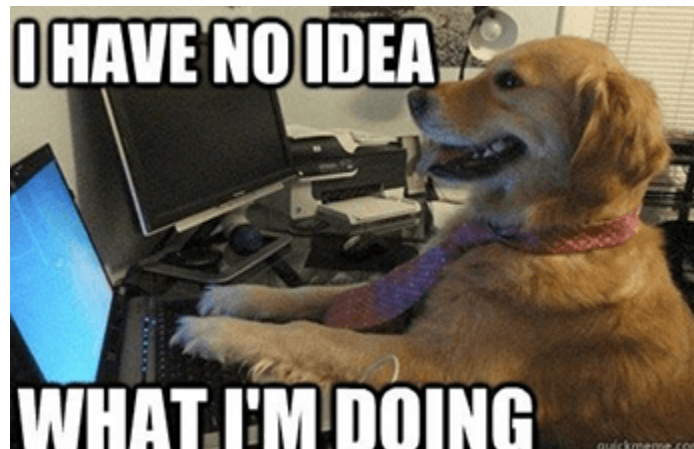
# append the SDK tarball
cat /home/alan/poky/build/tmp/work/qemu86-poky-linux/ovmf-shell-image/1.0-r0/x86_64-deploy-ovmf-shell-image-populate-sdk/poky-glibc-x86_64-ovmf-shell-image-i586-toolchain-2.3.1.tar.xz
>> /home/alan/poky/build/tmp/work/qemu86-poky-linux/ovmf-shell-image/1.0-r0/x86_64-deploy-ovmf-shell-image-populate-sdk/poky-glibc-x86_64-ovmf-shell-image-i586-toolchain-2.3.1.sh

# delete the old tarball, we don't need it anymore
rm /home/alan/poky/build/tmp/work/qemu86-poky-linux/ovmf-shell-image/1.0-r0/x86_64-deploy-ovmf-shell-image-populate-sdk/poky-glibc-x86_64-ovmf-shell-image-i586-toolchain-2.3.1.tar.xz
which triggered exception OSError: [Errno 12] Cannot allocate memory

Summary: There were 2 warnings and 1 error message shown, returning a non-zero exit code.
Summary: There was 1 ERROR message shown, returning a non-zero exit code.
alan@debian:~/poky/build
  
```

Low Disk Space  
This computer has only 395.4 MB disk space remaining.

Examine Ignore



## Episode 21: Building and Installing Linux, Part 2

July 31, 2017

This week, I had some success building a Linux image using the Yocto Project. But, that's the good news.

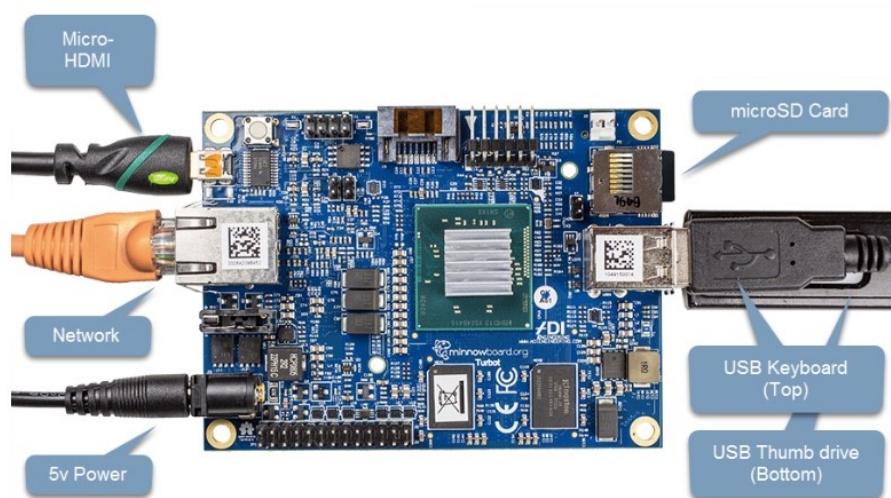
From last week's [Episode 20 of the MinnowBoard Chronicles](#), I described the trials and tribulations of the two projects I have underway: installing Ubuntu 16.04.1 LTS on my MinnowBoard Turbo, and building a complete Linux image using the Yocto Project.

To install the Ubuntu image, I've just been following the directions at the [MinnowBoard site's tutorial](#). The procedure is very straightforward, but the bad news is that I haven't yet gotten it completed. Last week, I kept getting the following error message:

*(intramfs) Unable to find a medium containing a live file system*

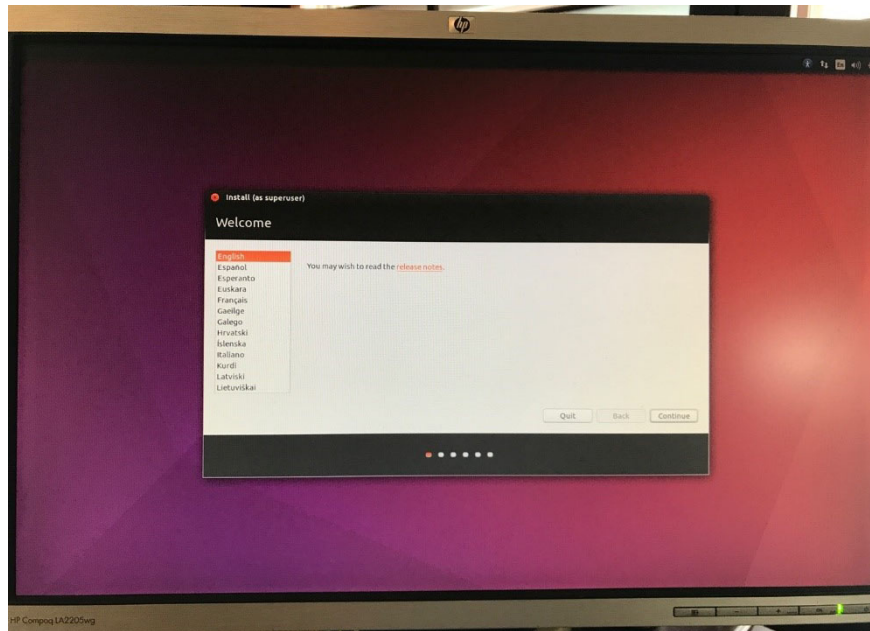
I found that I could get past this stage, and actually get to the main installer page, by putting the USB stick with the ISO image on the top USB port, and the keyboard input on the bottom USB port, opposite to what is described in the [tutorial](#), as diagrammed below:

The above is what's in the tutorial.





When I swap them, I don't get the intramfs error, and am able to get to the first installation screen:



But, I can't get any further! The MinnowBoard won't accept any keyboard or mouse input at that point. I think maybe it wants the keyboard/mouse USB hub to be plugged into USB port 1. I'm not exactly sure what is going on here; I think I'll need to try yet another USB flash stick, as my next step.

So, being not easily discouraged, I returned to my second project in-work, which is building the Yocto image. Last week, the build blew up due to a lack of space on my PC hard drive. As I mentioned before, I'm building this within a Debian VM using VirtualBox on my old, slow Windows PC at home. By poking around a little, I discovered that I had only allocated 8GB of hard disk space to the VM, as per the [Linux tutorial](#) I had used months ago to help me learn about virtual machines and Linux. I quote from page 20:

***“8GB should be plenty for learning about Linux servers.”***

But, of course, learning about Linux, and building an image, are two different things. I decided I needed about 10X that amount of space to do a complete build.



So, that's what I did. It took me a couple of hours to create the bigger VM, and then re-follow the Yocto build instructions. After 15 hours of my home PC running flat out, below is the screen I saw:

```

alan@debian: ~/poky/build
File Edit View Search Terminal Help
127
Connecting to downloads.yoctoproject.org (downloads.yoctoproject.org) |198.145.20
.127|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 2535308 (2.4M) [application/octet-stream]
Saving to: '/home/alan/poky/build/downloads/uninative/2b4ffa308d9f19e0742a1a404
ff42495fb50c165e5ca0458cedca157372691a/x86_64-nativesdk-libc.tar.bz2'

2017-07-30 15:16:00 (579 KB/s) - '/home/alan/poky/build/downloads/uninative/2b4f
ffa308d9f19e0742a1a404ff42495fb50c165e5ca0458cedca157372691a/x86_64-nativesdk-li
bc.tar.bz2' saved [2535308/2535308]

Initialising tasks: 100% |#####| Time: 0:00:51
NOTE: Executing SetScene Tasks
NOTE: Executing RunQueue Tasks
WARNING: openssl-native-1.0.2k-r0 do_fetch: Failed to fetch URL http://www.opens
sl.org/source/openssl-1.0.2k.tar.gz, attempting MIRRORS if available
WARNING: libpng-native-1.6.28-r0 do_fetch: Failed to fetch URL http://distfiles.
gentoo.org/distfiles/libpng-1.6.28.tar.xz, attempting MIRRORS if available
Currently 1 running tasks (1230 of 6060) 20% |#####|
0: icu-58.2-r0 do_compile - 671s (pid 18401)

```

1,230 of 6,060 tasks have been executed after 15 hours. So, at this rate, it's going to take a few days to complete the build!

I know, I know, don't laugh. I've got to get a faster PC. This machine is about seven years old. And I shouldn't be running in a VM. So, that's a project for another day: building a state-of-the-art faster machine. I hear there are good prices available for the new AMD Ryzen 16-core Threadripper, or maybe an Intel Skylake-X....and I'd like to get a good graphics card with it too (for gaming, and also to learn something about cryptocurrency mining). All these hobbies, though, take me away from blogging (and, oh yes, there's that work thing too....hopefully my boss hasn't read this far).

In any event, I hope to finish the Yocto build in time for Episode 22!

Of course, all this work is aimed at debugging UEFI, GRUB, and ultimately Linux on the MinnowBoard, with our [SourcePoint](#) product. I'm particularly interested in using Intel Processor Trace (see [Episode 11](#)) to watch the boot flow. A great eBook on that subject is at [Intel Adds High-Speed Instruction Trace](#) (note: requires registration).

## Episode 22: Project Yocto success!

*August 7, 2017*

In the last episode of the MinnowBoard Chronicles, I shared my progress with building a qemu86 reference image using the Yocto Project. I'm pleased to say that the build is complete! Here's how I did it.

In [Episode 21](#), I at least got the image build started on Sunday afternoon, and as of Monday morning it had completed 1,230 out of 6,060 tasks. I've been simply following the [Yocto Project Quick Start](#) tutorial instructions, and kicked off the build with:

*bitbake core-image-sato*

I wanted to start building an image for the QEMU emulator first, and then move to creating an actual MinnowBoard Turbot image.

The two WARNINGS in yellow below were worrisome, but it seemed to keep going, so I just let it run through Monday afternoon and evening:

```

alan@debian: ~/poky/build
File Edit View Search Terminal Help
127
Connecting to downloads.yoctoproject.org (downloads.yoctoproject.org)|198.145.20
.127|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 2535308 (2.4M) [application/octet-stream]
Saving to: '/home/alan/poky/build/downloads/uninative/2b4fffa308d9f19e0742a1a404
ff42495fb50c165e5ca0458cedca157372691a/x86_64-nativesdk-libc.tar.bz2'

2017-07-30 15:16:00 (579 KB/s) - '/home/alan/poky/build/downloads/uninative/2b4ff
ffa308d9f19e0742a1a404ff42495fb50c165e5ca0458cedca157372691a/x86_64-nativesdk-li
bc.tar.bz2' saved [2535308/2535308]

Initialising tasks: 100% |#####| Time: 0:00:51
NOTE: Executing SetScene Tasks
NOTE: Executing RunQueue Tasks
#####: openssl-native-1.0.2k-r0 do_fetch: Failed to fetch URL http://www.opens
sl.org/source/openssl-1.0.2k.tar.gz, attempting MIRRORS if available
#####: libpng-native-1.6.28-r0 do_fetch: Failed to fetch URL http://distfiles.
gentoo.org/distfiles/libpng-1.6.28.tar.xz, attempting MIRRORS if available
Currently 1 running tasks (1230 of 6060) 20% |#####|
0: icu-58.2-r0 do_compile - 671s (pid 18401)

```

Tuesday morning, I got up, and the first thing I did was to go look at the progress of the build.

## Disaster!

```

alan@debian: ~/poky/build
File Edit View Search Terminal Help
Cloning into bare repository '/home/alan/poky/build/downloads/git2/github.com.r
m-software-management/createrepo_c'...
fatal: unable to connect to github.com:
github.com: Name or service not known

ERROR: createrepo-c-native-0.10.0+gitAUTOINC+748891ff8e-r0 do_fetch: Fetcher fai
lure for URL: 'git://github.com/rpm-software-management/createrepo_c'. Unable to
fetch URL from any source.
ERROR: createrepo-c-native-0.10.0+gitAUTOINC+748891ff8e-r0 do_fetch: Function fa
iled: base.do_fetch
ERROR: Logfile of failure stored in: /home/alan/poky/build/tmp/work/x86_64-linux
/createrepo-c-native/0.10.0+gitAUTOINC+748891ff8e-r0/temp/log.do_fetch.17944
ERROR: Task (virtual:native:/home/alan/poky/meta/recipes-devtools/createrepo-c/c
reaterepo-c_git.bb:do_fetch) failed with exit code '1'
NOTE: Tasks Summary: Attempted 4049 tasks of which 0 didn't need to be rerun and
1 failed.

Summary: 1 task failed:
  virtual:native:/home/alan/poky/meta/recipes-devtools/createrepo-c/createrepo-c
_git.bb:do_fetch
Summary: There were 4 WARNING messages shown.
Summary: There were 3 ERROR messages shown, returning a non-zero exit code.
alan@debian:~/poky/build$

```

Out of the 6,060 tasks, it completed 4,049 of them, and then crashed!

The error messages weren't particularly meaningful, and I did notice that we had had a home router outage sometime on Monday night (thanks Spectrum!), so I wondered if those were related. So, I just fired up bitbake again, and it proceeded where it left off: no work was lost!

When I got home from work on Tuesday, the build was finished! Yahoo! It only took about 48 hours:

```

alan@debian: ~/poky/build
File Edit View Search Terminal Help
...
NOTE: Resolving any missing task queue dependencies
...
Build Configuration:
  BB_VERSION      = "1.34.0"
  BUILD_SYS       = "x86_64-linux"
  NATIVELSBSTRING = "universal-4.9"
  TARGET_SYS      = "i586-poky-linux"
  MACHINE         = "qemux86"
  DISTRO          = "poky"
  DISTRO_VERSION  = "2.3.1"
  TUNE_FEATURES   = "m32 i586"
  TARGET_FPU      = ""
  meta
  meta-poky
  yocto-bsp       = "pyro:4a39979c8d1e560fa54240e99734a651dfbaa63a"
...
Initialising tasks: 100% |#####| Time: 0:00:52
NOTE: Executing SetScene Tasks
NOTE: Executing RunQueue Tasks
NOTE: Tasks Summary: Attempted 6060 tasks of which 4054 didn't need to be rerun
and all succeeded.
alan@debian:~/poky/build$
...

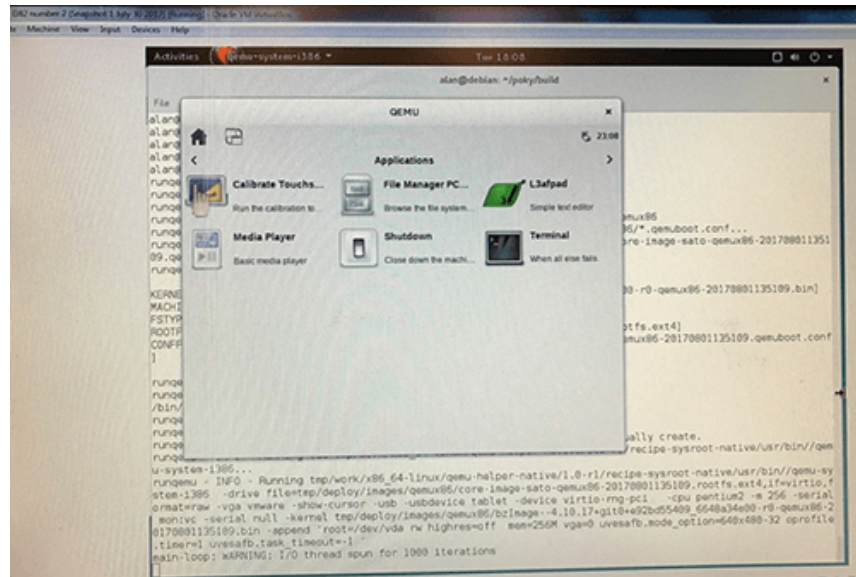
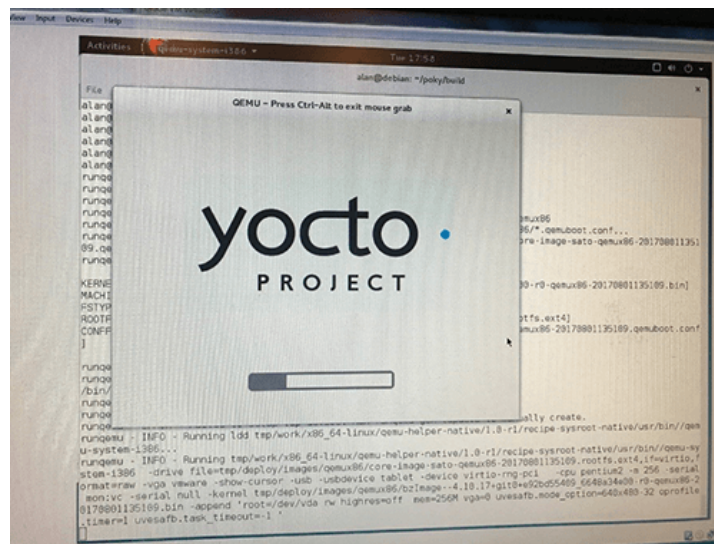
```



The ultimate test, of course, is to see if the image works. For that, I wanted to run the image within the QEMU emulator, with this command:

***runqemu qemux86***

That worked too! I was delighted to see it launch the first time. Here are a couple of screenshots:



That's all I had time for this week. In my next installment, I'll build another image for the MinnowBoard, and have it running on real hardware. Stay tuned!

And now, a word from our sponsor: once I've got your image built, I'm going to want to debug the lowest levels of the boot process with SourcePoint. A good introduction to this technology is in our eBook, [Intel Trace Hub | Finding Root Cause](#) (note: requires registration).

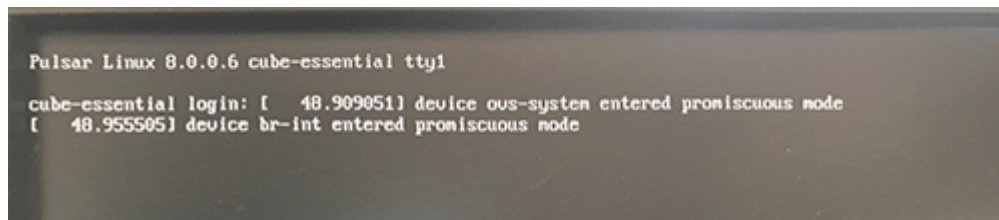
## Episode 23: Trying Wind River Pulsar Linux, and taking a break

*September 4, 2017*

I've been stymied on both fronts: my MinnowBoard Turbot finally stopped recognizing the keyboard, and my home computer build machine has crashed!

In [Episode 21, Building and Installing Linux Part 2](#), I vented my frustrations in getting the MinnowBoard to install Ubuntu Linux. Once I got past the EFI shell, the Minnow would stop taking keyboard input. So, I'd be sitting there staring at the Ubuntu installation screen; so near, and yet so far. Everything I tried, including using different keyboards, swapping the keyboard and USB flash stick between USB ports, using different USB flash sticks, etc. etc. were to no avail.

I even tried installing Wind River Pulsar Linux instead of Ubuntu. There are some very clear instructions on the MinnowBoard site [here](#). But, alas, I would again get to a stage in the installation process and could get no further. Note to self: look up "linux promiscuous mode" sometime:

A screenshot of a terminal window with a dark background and light-colored text. The text shows the boot process of Pulsar Linux 8.0.0.6. It starts with 'Pulsar Linux 8.0.0.6 cube-essential tty1', followed by 'cube-essential login: [ 48.909051] device ous-system entered promiscuous mode', and then '[ 48.955505] device br-int entered promiscuous mode'.

And now, the Minnow won't take keyboard commands even at the EFI shell (this used to work). I've googled this extensively, and the closest I can seem to get to a diagnosis is in some of its Amazon reviews, where someone said that "The USB ports are underpowered/not load protected enough...overall, it's a great board, if you can avoid damaging the USB ports". In any event, I think I've tried everything, and I'm pretty sure what I'm left with is a hardware failure. I plan to RMA the board with Netgate soon.

So, following up on [Episode 22, Project Yocto Success!](#), I continued working on building a MinnowBoard Linux image using the Yocto project. From last time, I was halfway through the build. But then my seven-year-old dual-core home computer crashed. And the hard disk was wiped out. I guess the 48-hour Yocto builds took too great a toll.



Where does this leave me? Well, this weekend, I finished ordering all of the parts for my replacement dream machine. AMD Ryzen 7 1700X (sorry Intel, I couldn't resist the prices; my next dream machine will be Intel-based) with eight cores and 16 threads, 16GB RAM, 500GB SSD, 2TB HDD, NVIDIA GTX 1060. Those Yocto builds should scream then. I'll keep you all posted on the dream machine build! It might take me a couple of weekends, so stay tuned.

It'll be nice to get a new, fast machine. Some of the work I need to do with SourcePoint, such as running CScripts, requires a lot of horsepower from the remote host. For more information on some of these Python-based massive scripts, see our eBook, [SourcePoint CScripts Support](#) (note: requires registration).

## Episode 24: New MinnowBoard, New PC, and a nod to Netgate

*October 11, 2017*

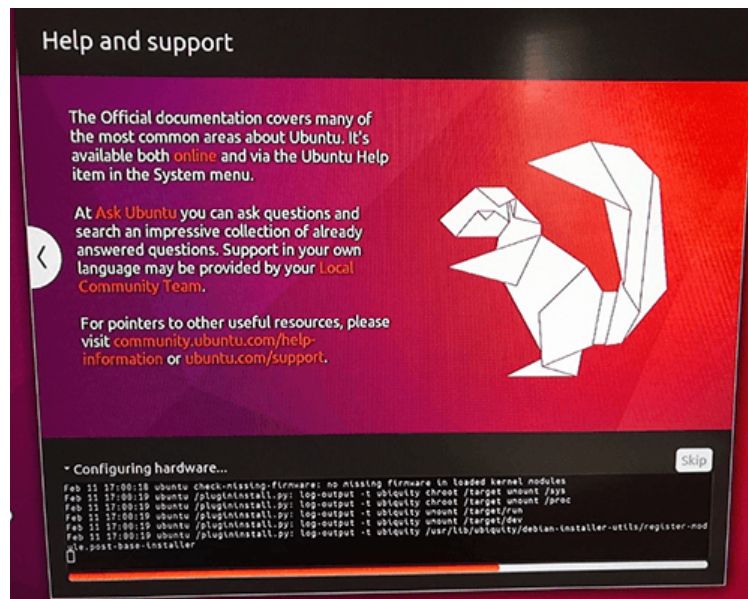
In Episode 23, I mentioned taking a short sabbatical, because my MinnowBoard USB ports stopped working, and my 7-year-old home PC build machine crashed. The good news is both issues are both fixed! Let the fun begin again.

In [Episode 23](#), I experienced the “perfect storm” of failures: after an extended period of intermittent behavior, my MinnowBoard Turbot finally stopped recognizing the keyboard, mouse, and USB flash sticks entirely. This made installing a Linux image impossible. And, my seven-year-old, Intel Core 2 Duo machine I was using to build a Yocto Linux image for the MinnowBoard crashed for good in the middle of one of the interminable 48-hour builds.

For the MinnowBoard USB problem, I tried dozens of different things (different keyboard, different mouse, different USB port, etc.) to try to fix or work around this problem, to no avail; it was toast. I finally threw in the towel and decided to contact the supplier for an RMA. Having purchased the Minnow six months ago (it was still under warranty), I had to go back and find my purchase materials – as it turns out, I purchased it through Amazon, but the supplier was [Netgate](#), and the board was built by [ADI Engineering](#).

I have to say, that the experience with Netgate was a delight. I always approach these kinds of situations with a bit of trepidation – you never know what kind of customer service you’re going to receive. In this instance, I simply created a userid on their Support Portal, and got a Live Chat going with one of their staff (note that this was on a holiday weekend!). After exchanging some information, I received an RMA number, and shipped the Minnow back to them. As it turns out, they performed a post-mortem on the unit I sent to them and were able to verify the symptoms. I received my new MinnowBoard that same week. That’s great turnaround and customer service!

The proof of the pudding, of course, was whether the new Minnow was able to install Ubuntu Linux. There’s a great tutorial on the MinnowBoard site on [Installing Ubuntu 16.04.1 LTS](#) that walks you through it. And the installation started, with full keyboard and mouse control. I was ecstatic:



But, when trying to do:

```
sudo apt-get -y update
sudo apt-get -y dist-upgrade
```

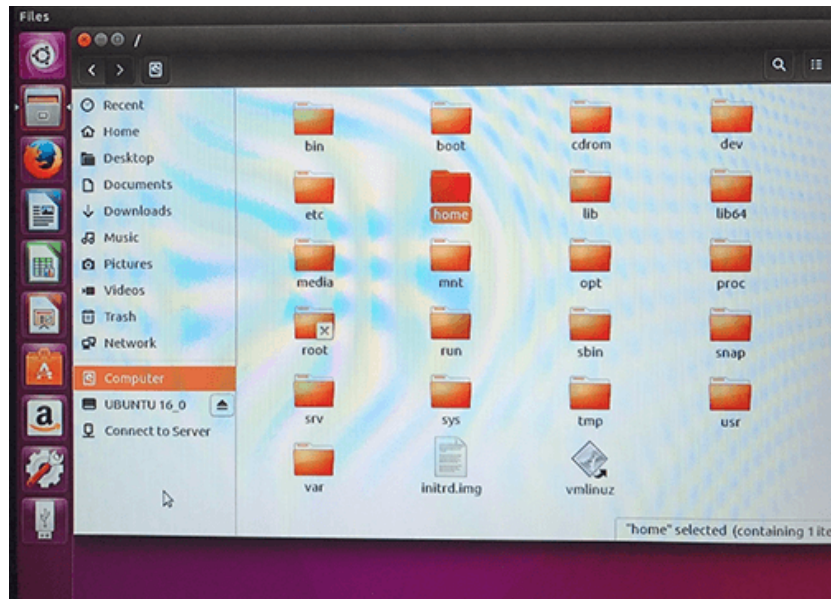
to bring Ubuntu up-to-date with patches, I got an error message:

***Problem executing scripts APT::Update::Post-Invoke-Success***

Googling this led me to the workaround to remove libappstream3 with the CLI command:

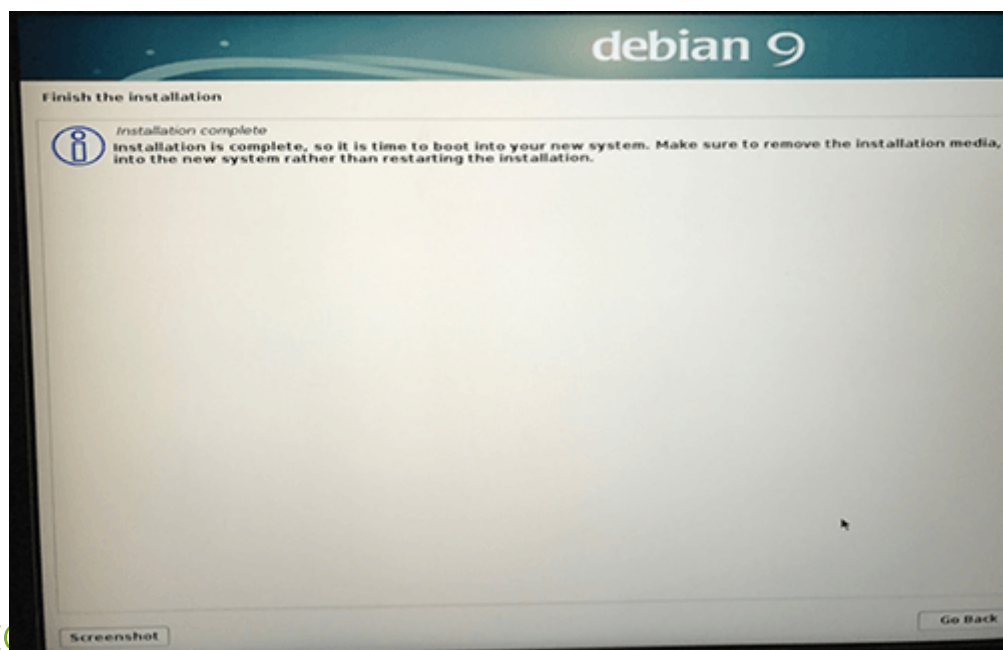
```
sudo apt-get remove libappstream3
```

That did the trick. The install completed successfully, and I got a working version of Ubuntu on my Minnow:



So, while all this was going on, I finished building my new home PC to replace the aging seven-year-old Yocto build machine. I equipped it with the AMD Ryzen 7 1700X CPU, 16GB RAM, 500GB SSD, NVIDIA GeForce GTX 1060 video card, and Windows 10 Home. I was one of the lucky people that build the machine and have it boot up the first time (well, there were a couple of scary moments, but I'll gloss over those for now).

With the machine assembled and Windows booting off of the SSD (what a joy to boot Windows in seconds versus about four minutes on my old machine), I installed a 2TB 7200RPM SATA III hard drive and installed Debian 9 Linux on it. This is where I plan to do most of the Yocto builds:



Following the tutorial in the [Yocto Project Quick Start Guide](#), I quickly fired up an image build for emulation (QEMU). Running the command:

*bitbake core-image-sato*

took 1 hour, 5 minutes; compared to 48 hours on the old machine!!! So, about 50X faster. My productivity should leap forward now:

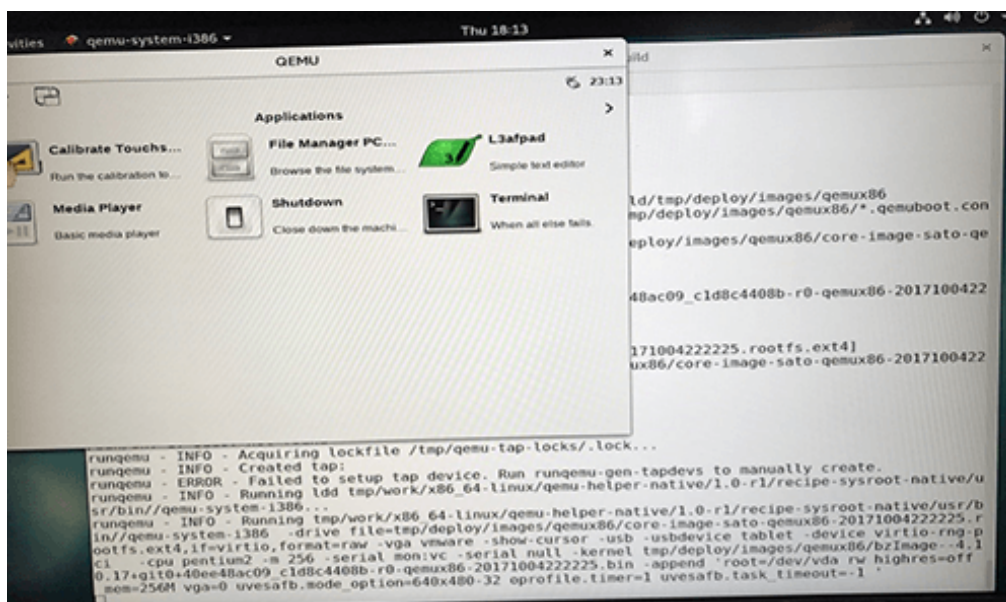
```

alansguigna@debian: ~/poky/build
bitbake core-image-sato
...
request sent, awaiting response... 200 OK
Content-Length: 2286285 (2.2M) [application/octet-stream]
Saving to: '/home/alansguigna/poky/build/downloads/uninative/ed033c868b87852b07957a4400f3b7440346eala59b6d3e03075e/x86_64-nativesdk-libc.tar.bz2'

7-10-04 17:22:29 (859 KB/s) - '/home/alansguigna/poky/build/downloads/uninative/ed033c868b87852b07957a4400f3b7440346eala59b6d3e03075e/x86_64-nativesdk-libc.tar.bz2' saved [2286285/2286285]

Initialising tasks: 100% [#####] Time: 0:00
NOTE: Executing SetScene Tasks
NOTE: Executing RunQueue Tasks
...
libpng-1.6.28.tar.xz: attempting MIRROR if available
...
currently 15 running tasks (1378 of 6060) 22% [#####]
1: binutils-cross-i586-2.28-r0 do fetch - 257s (pid 29886) | 737
2: linux-yocto-4.10.17+gitAUTOINC+40ee48ac09_cld8c4408b-r0 do fetch (pid 9438) 14% | 771
3: cmake-native-3.7.2-r0 do configure - 87s (pid 7427)
4: glib-2.0-native-1.2.50.3-r0 do configure - 30s (pid 30557)
5: perl-native-5.24.1-r0 do install - 26s (pid 32403)
6: swig-native-3.0.12-r0 do compile - 13s (pid 26837)
7: libice-native-1.1.0.9-r0 do configure - 13s (pid 27255)
8: python-native-2.7.13-r1.1 do install - 12s (pid 27652)
9: orc-native-0.4.26-r0 do compile - 11s (pid 30913)
10: libxml2-native-2.9.4-r0 do compile - 9s (pid 4154)
11: cross-localedef-native-2.25-r0 do configure - 4s (pid 11971)
12: rpm-native-1.4.13.90+gitAUTOINC+a8e51b3bb0-r0 do compile - 3s (pid 13077)
13: xserver-xorg-2.1.19.1-r0 do unpack - 0s (pid 19457)
14: iso-codes-3.74-r0 do populate_sysroot - 0s (pid 19973)

```





If you compare the top above screen with that in [Episode 21](#), you can see that the new computer is running the compilation multi-threaded (as opposed to only two running tasks/threads on the old machine). That speeds things up tremendously. And the Yocto tutorial says that subsequent builds past the first one are much faster, because the OpenEmbedded build system re-uses files from previous builds as much as possible. Sweet!

Next time, I'll do a Yocto build for the MinnowBoard Turbot, and install it. Those who follow the MinnowBoard Chronicles know that I've tried this before and failed. But, with a faster machine and a new Minnow, we'll see!

And now, a word from our sponsor: my employer is kind enough to allow me to use my weekends exploring technology and writing about my experiences. If you are enjoying this series, and are also interested in learning more about this fascinating technology, please feel free to register for one of our eBooks at [ASSET's eResources on Software Debug](#).



## Episode 25: Yocto builds for the MinnowBoard and the Portwell Neptune Alpha

*October 16, 2017*

In Episode 24, I finished off my new build machine, successfully did a QEMU image build on it, and loaded an off-the-shelf Ubuntu image into my new MinnowBoard Turbot. This week, I tackled a MinnowBoard Linux image build using Yocto, loaded it into my MinnowBoard, and also set about doing a Yocto image build for the Portwell Neptune Alpha board. But I ran into some problems.

At the end of [Episode 24](#), I was floating on Cloud 9: my new PC did a Yocto image build for the QEMU emulator screamingly fast: about fifty times faster than my old PC. So, I looked forward to having a lot more fun exploring this technology, and faster too. On the other hand, another way to look at this is, I could make 50X more mistakes in the same amount of time.

To put it to the test, I tackled a MinnowBoard Turbot Yocto image build, using the instructions in the [Yocto Project Quick Start Guide](#). Granted, these instructions are for the MinnowBoard MAX, opposed to the MinnowBoard Turbot that I have, but I figured they were close enough that it should just work. I fired up the bitbake core-image-base as per the instructions, and after about 45 minutes, the build completed!

```
alansguigna@debian: ~/poky/build
File Edit View Search Terminal Help
alansguigna@debian:~/poky/build$ bitbake core-image-base
WARNING: Most distribution "debian-9" has not been validated with this version of the build
libly experience unexpected failures. It is recommended that you use a tested distribution.
Parsing recipes: 100% |#####
Parsing of 857 .bb files complete (0 cached, 857 parsed). 1328 targets, 56 skipped, 0 masked
NOTE: Resolving any missing task queue dependencies

Build Configuration:
BB_VERSION      = "1.34.0"
BUILD_SYS       = "x86_64-linux"
NATIVE_SBSRING = "universal"
TARGET_SYS      = "x86_64-poky-linux"
MACHINE         = "intel-corei7-64"
DISTRO          = "poky"
DISTRO_VERSION  = "2.3.2"
TUNE_FEATURES   = "m64 corei7"
TARGET_FPU      = ""
meta
meta-poky
meta-yocto-bsp  = "pyro:717303e6fbcbbel81ad9645d762eb5a85d934523"
meta-intel      = "pyro:63de2abadfa8164e28575e412e2bd9c315840ddb"

Initialising tasks: 100% |#####
NOTE: Executing SetScene Tasks
NOTE: Executing RunQueue Tasks
NOTE: Tasks Summary: Attempted 3899 tasks of which 1234 didn't need to be rerun and all succe

Summary: There was 1 WARNING message shown.
alansguigna@debian:~/poky/build$
```

Needless to say, I was pretty excited at this point. If you've been following the [MinnowBoard Chronicles series](#), you'll know that I've been working on this for quite some time. It was about time I got a break!

Alas, after using the Linux “dd” command to create a bootable image on a USB stick, and booting it in the MinnowBoard, I got the following messages up-front, after which the Minnow just hung:

```

3.372321 sit: IPv6, IPv4 and MPLS over IPv4 tunneling driver
3.372323 NET: Registered protocol family 17
3.372391 NET: Registered protocol family 36
3.373011 Key type dns_resolver registered
3.376521 microcode: sig=0x30679, pf=0x1, revision=0x0
3.378737 microcode: Microcode Update Driver: v2.01 (CiprianBuzian.Finet,
o.ub), Peter Oruba
3.380721 SSB version of gcc_enc/tsc engaged.
3.381791 Write loaded, crc32-v32-generic
3.383471 Key type encrypted registered
3.446987 random: fast init done
3.510720 rnc2: new high speed SMC card at address 0007
3.510751 rnc102: rnc2:0007 30326 29.8 GHz
3.661812 rnc102: pl x2 0
3.704990 console [netcon0] enabled
3.7091441 netconsole: network logging started
3.714313 rtc_cmos 00:00: setting system clock to 2016-01-29 00:00:23 UTC (
4539522)
3.723551 ALSA device list:
3.726912  No soundcards found.
3.817511 usb 1-2: new high-speed USB device number 3 using xhci-hcd
3.849061 clocksource: Switched to clocksource tsc
3.975991 hub 1-2:1.0: USB hub found
4.001941 hub 1-2:1.0: 4 ports detected
4.304092 f10cm: intelhubf (f10) is primary device
4.340074 usb 1-2.1: new low-speed USB device number 4 using xhci-hcd
4.352501 Console: switching to colour frame buffer device 240x67
4.432631 PHY 0000:00:02.0: f10: intelhubf frame buffer device
4.446591 rd: Waiting for all devices to be available before autodetect
4.453829 rd: If you don't use raid, use raid=nom autodetect
4.461870 rd: Autodetecting RAID arrays.
4.466000 rd: Scanned 0 and added 0 devices.
4.471241 rd: automn ...
4.494251 rd: ... automn DONE.
4.478431 Waiting for root device PARTUUID=5446c305-244f-4e90-804f-69c1f135277, ...
4.487591 input: Logitech USB Keyboard as /dev/input/lk0000:00:14.0.usb1-1-2.1-2.1-1.1-2.1.1-0.0003:0463:C3IC.0001/input/input4
4.534993 hid-generic 0003:0463:C3IC.0001: input: USB HID v1.10 Keyboard [Logitech USB Keyboard] on usb-0000:00:14.0-2.1/input0
4.562541 input: Logitech USB Keyboard as /dev/input/lk0000:00:14.0.usb1-1-2.1-2.1-1.1-2.1.1-0.0003:0463:C3IC.0002: input: USB HID v1.10 Keyboard [Logitech USB Keyboard] on usb-0000:00:14.0-2.1/input5
4.743091 acpi 0:0:0: Direct-Access Generic USBIX 1.0a PQ: 0 #MS: 2
4.753410 sd 0:0:0: Attached scsi generic sg type 0
4.760011 usb 1-2.2: new low-speed USB device number 5 using xhci-hcd
4.760421 sd 0:0:0: [sda] 391320 512-byte logical blocks: (1.99 GiB/1.66 GiB)
4.776344 sd 0:0:0: [sda] Write Protect is off
4.782397 sd 0:0:0: [sda] Write cache: disabled, read cache: enabled, doesn't support DPO or FUA
4.786997 sda: sda1
4.800793 sd 0:0:0: [sda] Attached SCSI removable disk
4.822409 input: Logitech USB-PS/2 Optical Mouse as /dev/input/lm0000:00:14.0.usb1-1-2.1-2.1-2.1-2.1.1-0.0003:0463:C00E.0003/input/input6
4.937700 hid-generic 0003:0463:C00E.0003: input: USB HID v1.10 Mouse [Logitech USB-PS/2 Optical Mouse] on usb-0000:00:14.0-2.1/input0

```

After trying it several times with different USB sticks, I kept getting the same installation failure. So, I decided to take a rest, and do something else for a while.

At the office, our engineering group is doing some development for our [ScanWorks Embedded Diagnostics](#) embedded JTAG product on a board we procured from Portwell. This [Portwell board](#), part of the [Neptune Alpha OpenBMC Development Kit](#), has an ASPEED AST2500 BMC that we have ported our [embedded Intel x86 hardware-assisted debugging agent](#) onto. Our engineers are kind enough to allow me to tinker with it after-hours. But, first, I wanted to use Yocto myself to build an image for this board.

Instructions on how to work with OpenBMC are on the [Facebook OpenBMC GitHub](#). The directions are fairly straightforward, and fairly similar to building an image for the

MinnowBoard. I'm pretty sure that the Neptune Alpha platform is meta-fbtp in the meta-openbmc/meta-facebook directory. So I fired up bitbake again:

```
alansguigna@debian:~/poky/build$ bitbake fbtp-image
NOTE: Your conf/bblayers.conf has been automatically updated.
WARNING: Host distribution "Debian-9.1" has not been validated with this version of
the build system; you may possibly experience unexpected failures. It is recommended
that you use a tested distribution.
Parsing recipes: 100% |#####| Time: 00:00:15
Parsing of 1912 .bb files complete (0 cached, 1912 parsed). 2460 targets, 379 skipped,
0 masked, 0 errors.
NOTE: Resolving any missing task queue dependencies

Build Configuration:
BB_VERSION      = "1.30.0"
BUILD_SYS       = "x86_64-linux"
NATIVELSBSTRING = "Debian-9.1"
TARGET_SYS      = "arm-fb-linux-gnueabi"
MACHINE         = "fbtp"
DISTRO          = "poky"
DISTRO_VERSION  = "0.4"
TUNE_FEATURES   = "arm armv6"
TARGET_FPU      = "soft"
meta
meta-yocto
meta-yocto-bsp   = "krogoth:426bc4c3575a85391a60328edb1f7c6a6bdb95fd"
meta-oe
meta-networking
meta-python     = "krogoth:55c8a76da5dc099a7bc3838495c672140cedb78e"
meta-openbmc
meta-aspeed
meta-facebook
meta-fbtp       = "helium:900b1f1e10b3d4a3b7ce9b8db01182f79f0831ea"

NOTE: Fetching uninative binary shim from
http://downloads.yoctoproject.org/releases/uninative/1.0.1/x86_64-nativesdk-
libc.tar.bz2;sha256sum=acfle44a0ac2e855e81da6426197d36358bf7b4e88e552ef933128498c8910f
8
NOTE: Preparing RunQueue
NOTE: Checking sstate mirror object availability (for 1101 objects)
NOTE: Executing SetScene Tasks
NOTE: Executing RunQueue Tasks
WARNING: byacc-native-20150711-r0 do_fetch: Failed to fetch URL ftp://invisible-
island.net/byacc/byacc-20150711.tgz, attempting MIRRORS if available
WARNING: logrotate-3.9.1-r0 do_fetch: Checksum mismatch for local file
/home/alansguigna/poky/build/downloads/logrotate-3.9.1.tar.gz
Cleaning and trying again.
WARNING: logrotate-3.9.1-r0 do_fetch: Renaming
/home/alansguigna/poky/build/downloads/logrotate-3.9.1.tar.gz to
/home/alansguigna/poky/build/downloads/logrotate-3.9.1.tar.gz_bad-
checksum_e475e2e83d8c63dc7efe648cc50aabbf6
WARNING: logrotate-3.9.1-r0 do_fetch: Checksum failure encountered with download of
https://fedorahosted.org/releases/l/o/logrotate/logrotate-3.9.1.tar.gz - will attempt
other sources if available
WARNING: lmsensors-3.4.0-r0 do_fetch: Failed to fetch URL http://dl.lm-sensors.org/lm-
sensors/releases/lm_sensors-3.4.0.tar.bz2, attempting MIRRORS if available
WARNING: bios-util-0.2-r1 do_package_qa: QA Issue: /usr/bin/bios-util contained in
package bios-util requires /usr/bin/python, but no providers found in RDEPENDS_bios-
util? [file-rdeps]
```

```

WARNING: mTerm-0.1-r1 do_package_qa: QA Issue:
/usr/local/fbpackages/mTerm/mTerm_server contained in package mTerm requires
libc.so.6(GLIBC 2.4), but no providers found in RDEPENDS_mTerm? [file-rdeps]
ERROR: lzo-2.09-r0 do_configure: autoreconf execution failed.
ERROR: lzo-2.09-r0 do_configure: Function failed: do_configure (log file is located at
/home/alansguigna/poky/build/tmp/work/armv6-fb-linux-gnueabi/lzo/2.09-
r0/temp/log.do_configure.30301)
ERROR: Logfile of failure stored in: /home/alansguigna/poky/build/tmp/work/armv6-fb-
linux-gnueabi/lzo/2.09-r0/temp/log.do_configure.30301
Log data follows:
| DEBUG: Executing python function sysroot_cleansstate
| DEBUG: Python function sysroot_cleansstate finished
| DEBUG: SITE files ['endian-little', 'bit-32', 'arm-common', 'arm-32', 'common-
linux', 'common-glibc', 'arm-linux', 'arm-linux-gnueabi', 'common']
| DEBUG: Executing shell function autotools_preconfigure
| DEBUG: Shell function autotools_preconfigure finished
| DEBUG: Executing python function autotools_copy_aclocals
| DEBUG: SITE files ['endian-little', 'bit-32', 'arm-common', 'arm-32', 'common-
linux', 'common-glibc', 'arm-linux', 'arm-linux-gnueabi', 'common']
| DEBUG: Python function autotools_copy_aclocals finished
| DEBUG: Executing shell function do_configure
| Unescaped left brace in regex is deprecated, passed through in regex; marked by <--
HERE in m/\${ <-- HERE ([^ \t=:+{}}+)])/ at
/home/alansguigna/poky/build/tmp/sysroots/x86_64-linux/usr/bin/automake line 3939.
| Unescaped left brace in regex is deprecated, passed through in regex; marked by <--
HERE in m/\${ <-- HERE ([^ \t=:+{}}+)])/ at
/home/alansguigna/poky/build/tmp/sysroots/x86_64-linux/usr/bin/automake line 3939.
| automake (GNU automake) 1.15
| Copyright (C) 2014 Free Software Foundation, Inc.
| License GPLv2+: GNU GPL version 2 or later <http://gnu.org/licenses/gpl-2.0.html>
| This is free software: you are free to change and redistribute it.
| There is NO WARRANTY, to the extent permitted by law.
|
| Written by Tom Tromey <tromey@redhat.com>
| and Alexandre Duret-Lutz <adl@gnu.org>.
| AUTOV is 1
| NOTE: Executing ACLOCAL="aclocal --system-
acdir=/home/alansguigna/poky/build/tmp/work/armv6-fb-linux-gnueabi/lzo/2.09-
r0/build/aclocal-copy/" autoreconf --verbose --install --force --exclude=autopoint -I
/home/alansguigna/poky/build/tmp/work/armv6-fb-linux-gnueabi/lzo/2.09-r0/lzo-
2.09/autoconf/
| autoreconf: Entering directory `.'
| autoreconf: configure.ac: not using Gettext
| autoreconf: running: aclocal --system-
acdir=/home/alansguigna/poky/build/tmp/work/armv6-fb-linux-gnueabi/lzo/2.09-
r0/build/aclocal-copy/ -I /home/alansguigna/poky/build/tmp/work/armv6-fb-linux-
gnueabi/lzo/2.09-r0/lzo-2.09/autoconf/ -I /home/alansguigna/poky/build/tmp/work/armv6-
fb-linux-gnueabi/lzo/2.09-r0/lzo-2.09/autoconf/ --force
| acinclude.m4:162: warning: the serial number must appear before any macro definition
| acinclude.m4:206: warning: the serial number must appear before any macro definition
| Segmentation fault
| aclocal: error: echo failed with exit status: 139
| autoreconf: aclocal failed with exit status: 139
| WARNING: exit code 1 from a shell command.
| ERROR: autoreconf execution failed.
| ERROR: Function failed: do_configure (log file is located at
/home/alansguigna/poky/build/tmp/work/armv6-fb-linux-gnueabi/lzo/2.09-
r0/temp/log.do_configure.30301)
ERROR: Task 2622 (/home/alansguigna/poky/meta/recipes-support/lzo/lzo_2.09.bb,
do_configure) failed with exit code '1'
NOTE: Tasks Summary: Attempted 1906 tasks of which 15 didn't need to be rerun and 1
failed.
Waiting for 0 running tasks to finish:

```

```
Summary: 1 task failed:
/home/alansguigna/poky/meta/recipes-support/lzo/lzo_2.09.bb, do_configure
Summary: There were 8 WARNING messages shown.
Summary: There were 2 ERROR messages shown, returning a non-zero exit code.
```

Okay, something basic is going wrong. I have to figure it out. I've googled some of this, without any success yet. One tip might be that I'm building on Debian 9 on my new machine, and I got the same warning from both the MinnowBoard Turbot and the Neptune Alpha builds:

```
WARNING: Host distribution "Debian-9.1" has not been validated with this version of
the build system; you may possibly experience unexpected failures. It is recommended
that you use a tested distribution.
```

Stay tuned!

Just as an aside, I'm keenly interested in the Neptune Alpha board, because it bills itself as the platform for OpenBMC development. OpenBMC is, of course, the toolchain for system management for most if not all hyperscale cloud computing environments. Embedded JTAG control, or [ScanWorks Embedded Diagnostics](#), adds tremendous value to system management for said environments. I'll write more on this topic later.



## Episode 26: Linux image build segmentation faults on AMD?

October 22, 2017

It has been quite an adventure over the last week. I'm getting intermittent segmentation faults during my Yocto Linux image builds. Could it be a problem with my new AMD Ryzen 7 1700X CPU?

In [Episode 24](#), having built my new screamingly-fast AMD Ryzen 7 1700X based machine, I used Yocto to successfully build a new QEMU image in record time. But in last week's [Episode 25](#), I had a mixed bag of results. I did successfully build a Yocto image for my MinnowBoard, but unfortunately it failed to boot on my hardware. And when I tried to build a Yocto image for the [Portwell Neptune Alpha](#), it failed.

Last week, I presumed that the source of the problems was that I was building the images using Debian 9.1. I would always get the following message right after the bitbake started:

***WARNING: Host distribution "Debian-9.1" has not been validated with this version of the build system; you may possibly experience unexpected failures. It is recommended that you use a tested distribution.***

So, I proceeded to somewhat haphazardly try to troubleshoot this, by first trying to re-install an earlier version of Debian on my build machine. I rationalized this by remembering that when I was doing builds under Virtualbox on my old PC, I was running off of Debian 8.2, and those worked. So, I tried that first.

Alas, Debian 8.2 refused to install on my new machine. I tried the same thing with the most current "obsolete stable" release of Debian 8 ("jessie"), 8.9. I got the same error message at the beginning of each install, and it just hangs:

***core perfctr but no constraints; unknown hardware!***

```
[ 0.364897] core perfctr but no constraints; unknown hardware!
Loading, please wait...
/dev/sdb2: clean, 154577/121020416 files, 8598897/484069888 blocks
[ 7.507853] kvm: disabled by bios
-
```



I'm guessing here that these older version of Debian won't work with the Ryzen 7 chip; the last release of jessie was dated July 22, 2017.

So, it's back to Debian 9.1. At least I know that I can successfully install that version on my PC. Eventually, the Yocto project will do some testing on this release, and do some updates.

But, this time, when I tried to do the QEMU image build, it crashed!

This time (and thanks to my colleague, Adam Ley, for reminding me), I went into the log file at:

*~/poky/build/tmp/work/x86\_64-linux/qemu-native/2.8.0-r0/temp/log.do\_compile.23003*

and saw this error message:

*/home/alan/poky/build/tmp/work/x86\_64-linux/qemu-native/2.8.0-r0/qemu-2.8.0/tcg/tcg.c:2800:12: internal compiler error: Segmentation fault*

The "Segmentation fault" error got my attention. What's that all about? I happened to google this topic, and saw the article at [New Ryzen Is Running Solid Under Linux, No Compiler Segmentation Fault Issue](#). These segmentation fault issues seemed to happen on earlier Ryzen chips, under heavy loads such as Linux compiles.

Could I have possibly received an older part (manufactured prior to Week 25) that exhibits this fault under very special conditions? I'm going to do some more testing and see if this was a coincidence or happens repeatedly. I've read that there's a "[Kill Ryzen](#)" script that can manifest the issue. If this is my situation, it's reassuring to know that AMD has an [RMA process](#) for this issue.

My end goal, of course, is to have my build platform rock-solid, so I can build images for the Portwell Neptune Alpha board. This target is a development vehicle for OpenBMC, and supports the ASPEED AST2500 BMC, the most common service processor on cloud computing servers. Our [ScanWorks Embedded Diagnostics](#) team is using this board for its in-house development.

The MinnowBoard Chronicles Episode 27: Segfault on my AMD Ryzen 7 1700X

Last week, I suspected that I might be seeing segmentation fault failures on my new AMD Ryzen 7 1700X computer. I dug into this some more this week and learned a lot!

I'm pretty conservative when it comes to calling suppliers with problems regarding my electronics at home. I tend to want to dig into the issue and try to figure it out myself, often spending hours in the process. Some might call this a waste of time, but I often learn a lot in the process. And as an outcome, I really know what I'm talking about, when it comes time to call Tech Support. My wife thinks that this inclination is related to my refusal to ask for directions when I'm driving. Fortunately, in this era of Google Maps and built-in navigation systems, the latter is no longer an issue.

So, when my new AMD-based PC started throwing segmentation faults during my Yocto Linux builds ([see Episode 26](#)), I figured I should dig into it a little bit first. As I was tinkering around, I got a notification on my Debian 9.1 home page that a new release was available. I clicked on the "Updates" button, and soon enough, I now had Debian 9.2 on-board.

Also interestingly, about mid-week last week, I noticed that documentation for Yocto has been updated for Yocto: <http://www.yoctoproject.org/docs/2.4/yocto-project-qs/yocto-project-qs.html> has been updated to version 2.4 ("Rocko"), while before I was using version 2.3 ("Pyro"). So, I had to do a little work to get onto the new update.

I then jumped in with both feet, and did a build for the [Portwell Neptune Alpha](#), and it succeeded! And I no longer got the warnings about Debian incompatibility, so between the jump to Rocko and the update to Debian 9.2, that somehow resolved itself. Very encouraging!

Emboldened, I backed up and then did a build for QEMU (Quick Emulator). But, it crashed with a segmentation fault!

```
Finished binary package job, result 0, filename /home/alan/poky/build/tmp/work/i586-poky-linux/gcc-runtime/7.2.0-r0/deploy-rpms/i586/libssp-dev-6.2.0-r0.i586.rpm
```

**Segmentation fault**

```
WARNING: exit code 139 from a shell command
```

```
DEBUG: Python function do_package_rpm finished
```



```

checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... /bin/bash: line 22: 2529 Segmentation
fault      /bin/bash $s/$module_srcdir/configure --
srcdir=${topdir}/${module_srcdir} --cache-file=./config.cache '--disable-
multilib' '--enable-languages=c,c++,fortran,lto,objc' --program-transform-
name='s,y,y,' --disable-option-checking --build=x86_64-pc-linux-gnu --
host=x86_64-pc-linux-gnu --target=x86_64-pc-linux-gnu --disable-intermodule -
-enable-checking=yes,types --disable-coverage --enable-languages="c,c++,lto"
--disable-build-format-warnings
Makefile:12563: recipe for target 'configure-stage1-libdecnumber' failed
make[2]: *** [configure-stage1-libdecnumber] Error 139
make[2]: Leaving directory '/mnt/ramdisk/workdir/buildloop.d/loop-6'
Makefile:27079: recipe for target 'stage1-bubble' failed
make[1]: *** [stage1-bubble] Error 2
make[1]: Leaving directory '/mnt/ramdisk/workdir/buildloop.d/loop-6'
Makefile:941: recipe for target 'all' failed
make: *** [all] Error 2

```

So, it is time to contact AMD. I placed a ticket in their [online support system](#). Let's keep our fingers crossed!

Why am I doing all this? Well, partly it's a public service, as I'm doing a lot of Linux builds as I explore OpenBMC for the ASPEED AST2500 for our [ScanWorks for Embedded Diagnostics](#) product line. In particular, I'm interested in applying boundary-scan test technology on Intel-based servers using the ASPEED BMC. You can read more about the power of in-situ JTAG-based boundary-scan test in our eBook, [Embedded JTAG for Boundary-Scan Test](#) (note: requires registration).

## Episode 27: Segfault on my AMD Ryzen 7 1700X

October 29, 2017

Last week, I suspected that I might be seeing segmentation fault failures on my new AMD Ryzen 7 1700X computer. I dug into this some more this week and learned a lot!

I'm pretty conservative when it comes to calling suppliers with problems regarding my electronics at home. I tend to want to dig into the issue and try to figure it out myself, often spending hours in the process. Some might call this a waste of time, but I often learn a lot in the process. And as an outcome, I really know what I'm talking about, when it comes time to call Tech Support. My wife thinks that this inclination is related to my refusal to ask for directions when I'm driving. Fortunately, in this era of Google Maps and built-in navigation systems, the latter is no longer an issue.

So, when my new AMD-based PC started throwing segmentation faults during my Yocto Linux builds ([see Episode 26](#)), I figured I should dig into it a little bit first. As I was tinkering around, I got a notification on my Debian 9.1 home page that a new release was available. I clicked on the "Updates" button, and soon enough, I now had Debian 9.2 on-board.

Also interestingly, about mid-week last week, I noticed that documentation for Yocto has been updated for Yocto: <http://www.yoctoproject.org/docs/2.4/yocto-project-qs/yocto-project-qs.html> has been updated to version 2.4 ("Rocko"), while before I was using version 2.3 ("Pyro"). So, I had to do a little work to get onto the new update.

I then jumped in with both feet, and did a build for the [Portwell Neptune Alpha](#), and it succeeded! And I no longer got the warnings about Debian incompatibility, so between the jump to Rocko and the update to Debian 9.2, that somehow resolved itself. Very encouraging!

Emboldened, I backed up and then did a build for QEMU (Quick Emulator). But, it crashed with a segmentation fault!

*Finished binary package job, result 0, filename /home/alan/poky/build/tmp/work/i586-poky-linux/gcc-runtime/7.2.0-r0/deploy-rpms/i586/libssp-dev-6.2.0-r0.i586.rpm*

**Segmentation fault**

*WARNING: exit code 139 from a shell command*

*DEBUG: Python function do\_package\_rpm finished*

*DEBUG: Python function do\_package\_write\_rpm finished*

*: Function failed: BUILDSPEC (log file is located at /home/ala/poky/build/tmp/work/i586-poky-linux/gcc-runtime/7.2.0-r0/temp/log.do\_package\_write\_rpm.30372)*

I then ran several different builds, for QEMU, the MinnowBoard, and the Neptune Alpha; and sometimes it would fail, and sometimes succeed. But mostly it would fail. So, it was time to get more rigorous on this. Having read the articles at [Ryzen Is Running Solid Under Linux, No Compiler Segmentation Fault Issue](#) and about the [Kill Ryzen](#) script, I began to suspect that maybe there was something wrong with my CPU, and it was an older model. So I used Git to download the Kill Ryzen script, and ran it:

```

kill-ryzen.sh
~/Downloads/Kill-Ryzen/ryzen-test-master

alan@debian: ~/Downloads/Kill-Ryzen/ryzen-test-master
File Edit View Search Terminal Help
Using 16 parallel processes
Hint: You are currently not seeing messages from other users and the system.
Users in the 'systemd-journal' group can see all messages. Pass -q to
turn off this notice.
No journal files were opened due to insufficient permissions.
[loop-0] Sun Oct 29 14:45:30 CDT 2017 start 0
[loop-1] Sun Oct 29 14:45:31 CDT 2017 start 0
[loop-2] Sun Oct 29 14:45:32 CDT 2017 start 0
[loop-3] Sun Oct 29 14:45:33 CDT 2017 start 0
[loop-4] Sun Oct 29 14:45:34 CDT 2017 start 0
[loop-5] Sun Oct 29 14:45:35 CDT 2017 start 0
[loop-6] Sun Oct 29 14:45:36 CDT 2017 start 0
[loop-7] Sun Oct 29 14:45:37 CDT 2017 start 0
[loop-8] Sun Oct 29 14:45:38 CDT 2017 start 0
[loop-9] Sun Oct 29 14:45:39 CDT 2017 start 0
[loop-10] Sun Oct 29 14:45:40 CDT 2017 start 0
[loop-11] Sun Oct 29 14:45:41 CDT 2017 start 0
[loop-12] Sun Oct 29 14:45:42 CDT 2017 start 0
[loop-13] Sun Oct 29 14:45:43 CDT 2017 start 0
[loop-14] Sun Oct 29 14:45:44 CDT 2017 start 0
[loop-15] Sun Oct 29 14:45:45 CDT 2017 start 0
[loop-6] Sun Oct 29 14:50:41 CDT 2017 build failed
[loop-6] TIME TO FAIL: 311 s

```

Yes, it crashed after five minutes. And this happened repeatedly.

But the screen shot didn't say why it crashed. I found from the Kill-Ryzen script README.md that I had to go into the /mnt/ramdisk/workdir/buildloop.d/loop-6/build.log to see the details behind the failure (note that the "6" comes from the signified "loop-6" failure in the screenshot above).



And the failure logged was, indeed, a segmentation fault, as can be seen from the fifth line below of the last lines in the log:

```
checking for suffix of executables...
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... /bin/bash: line 22: 2529 Segmentation
fault /bin/bash $s/$module_srcdir/configure --
srcdir=${topdir}/${module_srcdir} --cache-file=./config.cache '--disable-
multilib' '--enable-languages=c,c++,fortran,lto,objc' --program-transform-
name='s,y,y,' --disable-option-checking --build=x86_64-pc-linux-gnu --
host=x86_64-pc-linux-gnu --target=x86_64-pc-linux-gnu --disable-intermodule -
-enable-checking=yes,types --disable-coverage --enable-languages="c,c++,lto"
--disable-build-format-warnings
Makefile:12563: recipe for target 'configure-stagel-libdecnumber' failed
make[2]: *** [configure-stagel-libdecnumber] Error 139
make[2]: Leaving directory '/mnt/ramdisk/workdir/buildloop.d/loop-6'
Makefile:27079: recipe for target 'stagel-bubble' failed
make[1]: *** [stagel-bubble] Error 2
make[1]: Leaving directory '/mnt/ramdisk/workdir/buildloop.d/loop-6'
Makefile:941: recipe for target 'all' failed
make: *** [all] Error 2
```

So, it is time to contact AMD. I placed a ticket in their [online support system](#). Let's keep our fingers crossed!

Why am I doing all this? Well, partly it's a public service, as I'm doing a lot of Linux builds as I explore OpenBMC for the ASPEED AST2500 for our [ScanWorks for Embedded Diagnostics](#) product line. In particular, I'm interested in applying boundary-scan test technology on Intel-based servers using the ASPEED BMC. You can read more about the power of in-situ JTAG-based boundary-scan test in our eBook, [Embedded JTAG for Boundary-Scan Test](#) (note: requires registration).

## Episode 28: Returning my AMD Ryzen 7 1700X CPU

*November 29, 2017*

Yes, I received an older AMD Ryzen 7 1700X CPU from Amazon. It's RMA time!

In [Episode 27](#), I wrote about intermittently getting segmentation faults on my new AMD-based PC whenever I did Yocto Linux builds. I found some information online in [New Ryzen Is Running Solid Under Linux, No Compiler Segmentation Fault Issue](#) and [AMD Replaces Ryzen CPUs for Users Affected By Rare Linux Bug](#).

It seems that earlier production runs of the chip had a problem with cache coherency. So, after I returned from my vacation, I put in for an RMA number from AMD, and proceeded to remove and box up the faulty CPU. Whenever I've done PC builds, installing the CPU and heatsink are always the most hair-raising part, and taking this apart was a little nerve-wracking. Nonetheless, after CPU was removed and the thermal paste wiped off, the markings did in fact tell the tale:



The “UA 1709PGT” on the second line designated that this was a work week 09 production run, and all CPUs prior to WW25 are expected to have the segfault issue.

So, it's back to AMD for the bad CPU. So far, their RMA process has been pretty responsive. I expect the new CPU back next week and will keep you posted.

Once I get my machine back together again, I'll resume my efforts on the MinnowBoard Turbot, to install a working implementation of Linux. I expect to do some basic source-level Linux kernel debug using [SourcePoint](#). Also, I'm planning on further investigations into the Portwell Neptune Alpha board, which supports the ASPEED AST2500 BMC, our development platform for the [ScanWorks Embedded Diagnostics](#) JTAG-based run-control debug product.

## Episode 29: My new AMD Ryzen 7 CPU works, kind of

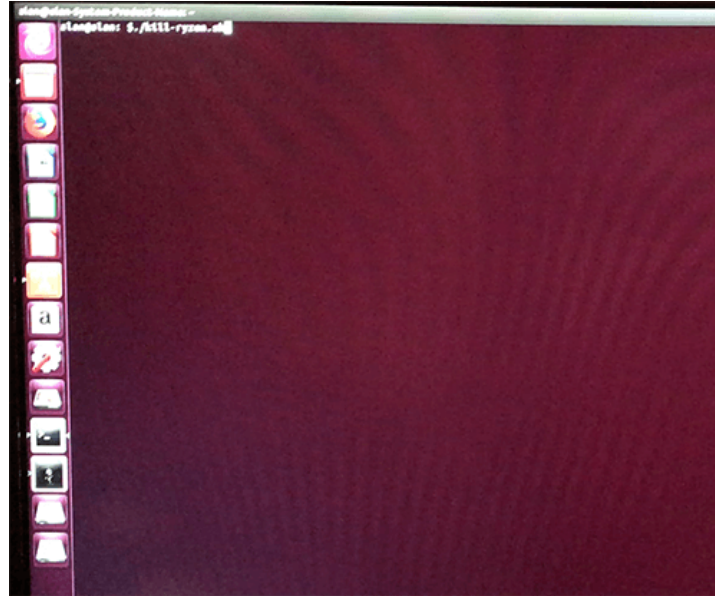
*December 25, 2017*

Out of the frying pan, and into the fire: I beat the #&@! out of my new CPU from AMD, and the segmentation faults have gone away. But, now the new system is crashing!

In the MinnowBoard Chronicles Episodes [27](#) and [28](#), I wrote about my discovery that I had acquired an older AMD Ryzen 7 1700X CPU from Amazon. The older production runs of these chips exhibited problems with cache coherency, that only manifest themselves rarely when you're cranking all 16 threads simultaneously. And that was just what I was doing with my Yocto Linux builds for the MinnowBoard Turbot and Portwell Neptune Alpha boards: the compilation process maxes out CPU utilization. This past month, I RMA'ed my older CPU (that had a datestamp of work week 09) to AMD, and very promptly got a replacement in the mail (a nod to AMD for responding so quickly and efficiently). When I unwrapped it, I was delighted to see that it was a much more current production run:

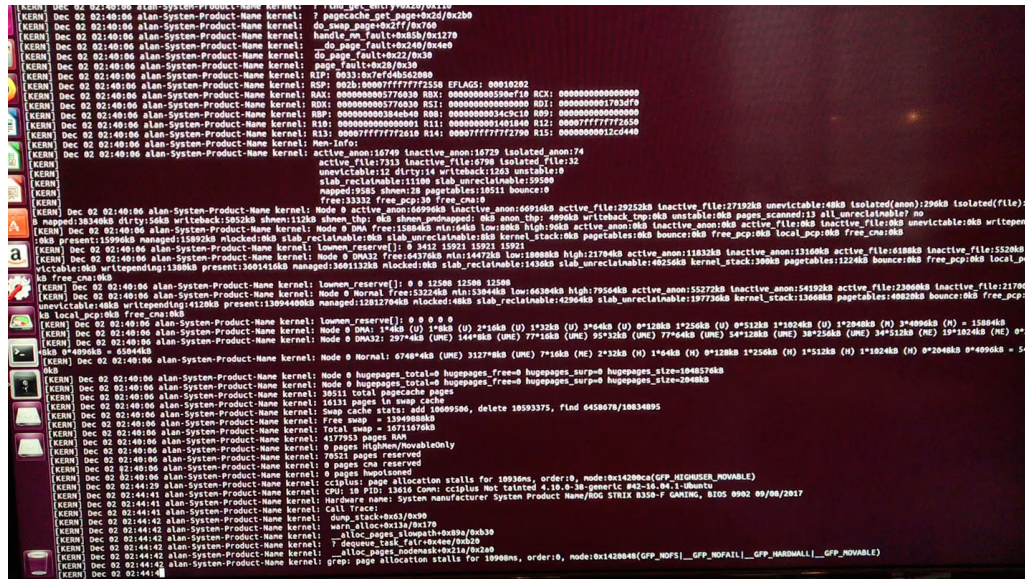


If you look carefully, you'll see that the datestamp is "1737", versus the "1709" from my previous device. From my researches on the web, via for example [New Ryzen Is Running Solid Under Linux, No Compiler Segmentation Fault Issue](#), we know that any CPU with a datestamp after work week 25 should be good. So, I carefully re-installed the new CPU into my home-built PC (which is always a nerve-wracking process, by the way, since this is my money and I don't want to mess anything up), and put it to the test by running the Kill Ryzen script again:



It ran for several hours, and it was getting late, so, I let it run overnight. When I got up in the morning, this is what I saw:

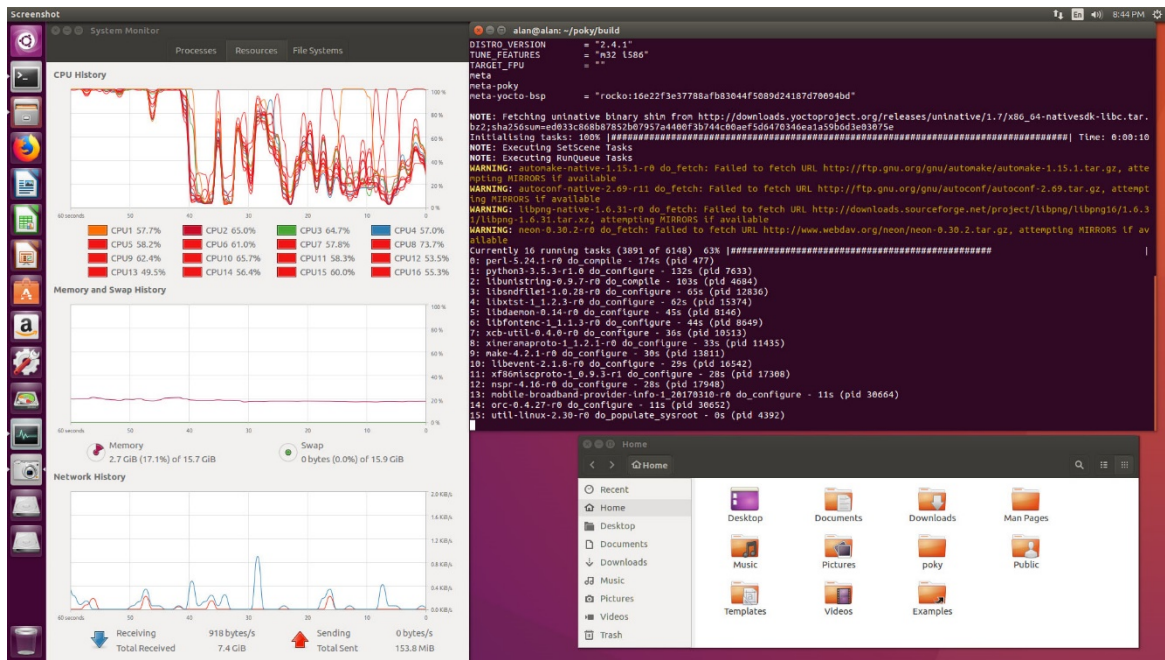
In the middle of the night, it crashed the system, with a kernel dump! This bears further investigation. Is it some new flaw in the new chip? Some incompatibility with Ubuntu? Did I re-install the new CPU correctly, with the right amount of thermal paste? Or maybe a bug in the Kill Ryzen script? So many interesting avenues to explore, so little time...



It's easy to get distracted while on a mission, but I decided to put this issue in the parking lot for now and focus on the main goal: doing a Linux build for the MinnowBoard Turbot. Apparently, something is still suspect with my system, but at least it appears that it can run for several hours without the segmentation faults manifesting; which would lead me to believe that I can probably consistently do a Yocto build without any failures. It was time to put that to the test.

Since it's been a while since I did a Yocto build, I decided to do a QEMU emulator run from a fresh environment, following the directions in the [Yocto Project Quick Start Guide](#). QEMU images were easier to build, I found, with less chance of user error on my part. I used the same approach as documented in [The MinnowBoard Chronicles Episode 22: Project Yocto success!](#), and it fired up right away and started running. Normally, it takes about 45 minutes (on my new PC, using all 16 threads, assuming it didn't crash due to an AMD segmentation fault on the old CPU) to build a QEMU image, so I stepped away for a coffee, and let it run:





Alas, when I returned, the system was sitting in the Ubuntu login screen! Somehow, something bad was happening during the build, and it would never complete, but rather would reboot and put me into the login screen. I tried this numerous times, and always got the same result. I looked into the logs in `~poky/build/tmp/log/cooker/qemux86`, and saw that it got to task 4283 of 6148, but then the log ended, with no failure information. The last lines looked like:

```

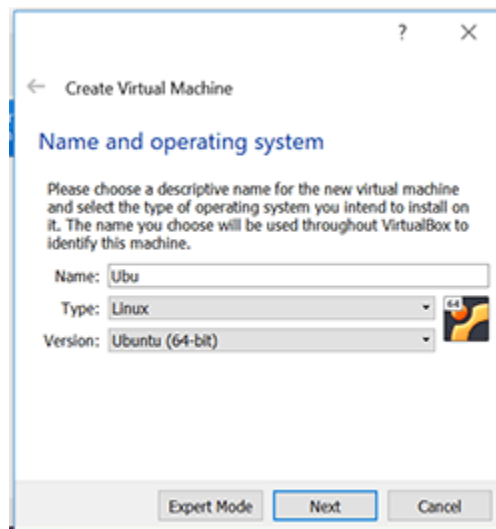
NOTE: Running task 4281 of 6148 (/home/alan/poky/meta/recipes-core/dbus/dbus-glib_0.108.bb:do_package)
NOTE: recipe libxt-1.1.5-r0: task do_package: Started
NOTE: recipe dbus-glib-0.108-r0: task do_package: Started
NOTE: recipe eudev-3.2.2-r0: task do_compile: Succeeded
NOTE: Running task 4282 of 6148 (/home/alan/poky/meta/recipes-core/udev/eudev_3.2.2.bb:do_install)
NOTE: recipe eudev-3.2.2-r0: task do_install: Started
NOTE: recipe libtirpc-1.0.2-r0: task do_package: Started
NOTE: recipe cairo-1.14.10-r0: task do_configure: Succeeded
NOTE: Running task 4283 of 6148 (/home/alan/poky/meta/recipes-graphics/cairo/cairo_1.14.10.bb:do_compile)
NOTE: recipe cairo-1.14.10-r0: task do_compile: Started

```

Somewhat frustrated at this point, I elected to try a different approach, with a completely fresh environment. I had had some earlier success with Yocto using Virtualbox on my old machine, so I installed Virtualbox on my new machine and clicked on “New” to begin the new installation. What I found was that only 32-bit operating were supported! It took a little digging around on Google, but I finally found out that the default setting on my AMI BIOS did not support virtualization. I had to boot into UEFI and enable this setting first. It was really hard to find: in

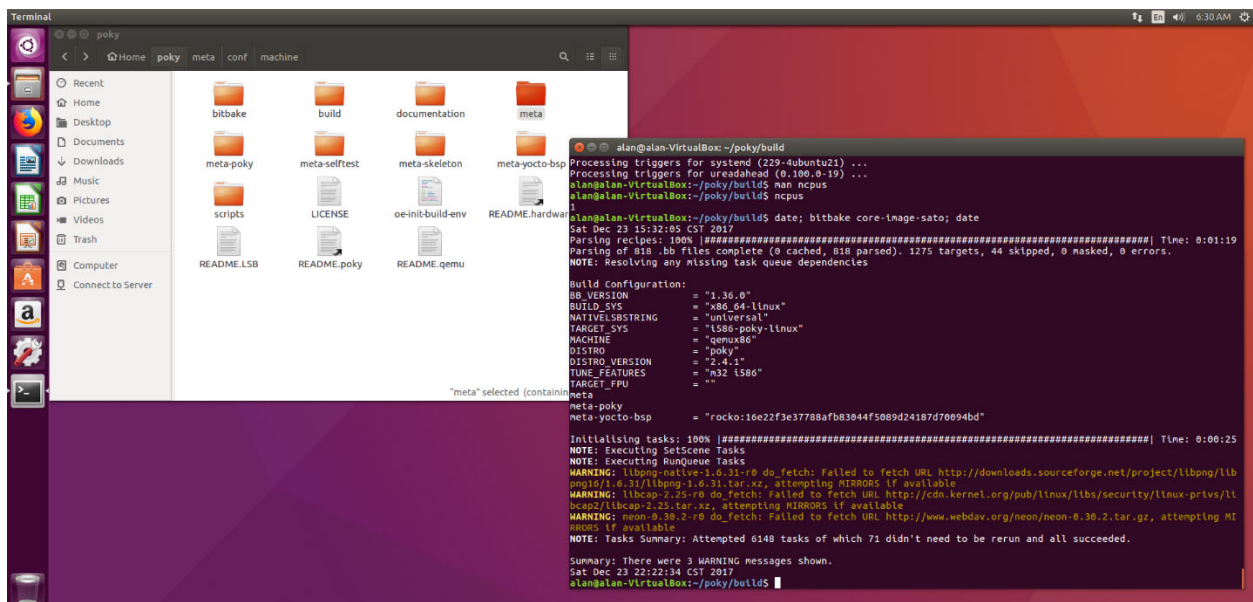


the AMI UEFI BIOS Utility, it's buried under the Advanced Menu, and labeled "SVM Mode". After this was done, I was finally able to create a new 64-bit virtual machine.



I decided to go ahead and install Ubuntu 16.04.3 LTS desktop, the same as I had on my separate Linux partition, to see if running it in a VM made any difference. I could always install Debian or any of the other distributions later.

So, once again following the instructions in the [Yocto Project Quick Start 2.4](#) document, I kicked off another QEMU bitbake. It only used one thread as a default within the VM, not the 16 that I have on my AMD CPU, but I decided to let it run anyway to see what happened. And it ran to completion!



Now, that is a real clue. On my separate Ubuntu partition, it was blasting away with all 16 threads, and never finishing. Cut it back to one thread and run in a VM, and it finished (albeit taking almost 7 hours, compared to the 45 minutes it was taking when it managed to run to completion using the older AMD CPU and 16 threads). There is a `BB_NUMBER_THREADS` variable that I can set in my project's `local.conf` configuration file that might be able to adjust this? Maybe I should adjust this to a higher number and see when it starts to crash? Stay tuned!

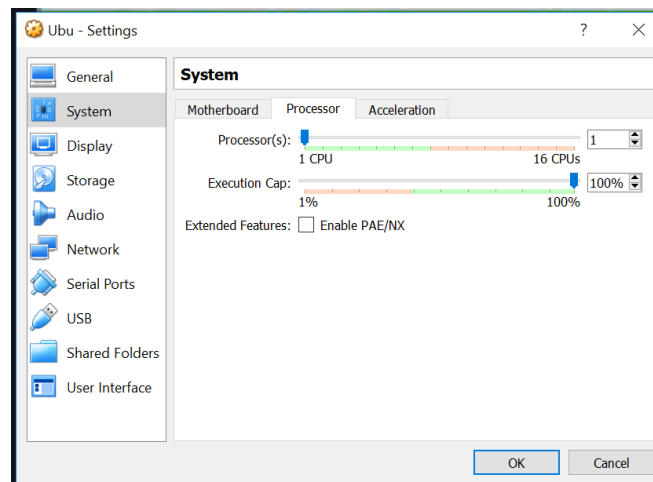
## Episode 30: Using all 16 threads on my Ryzen?

*January 14, 2018*

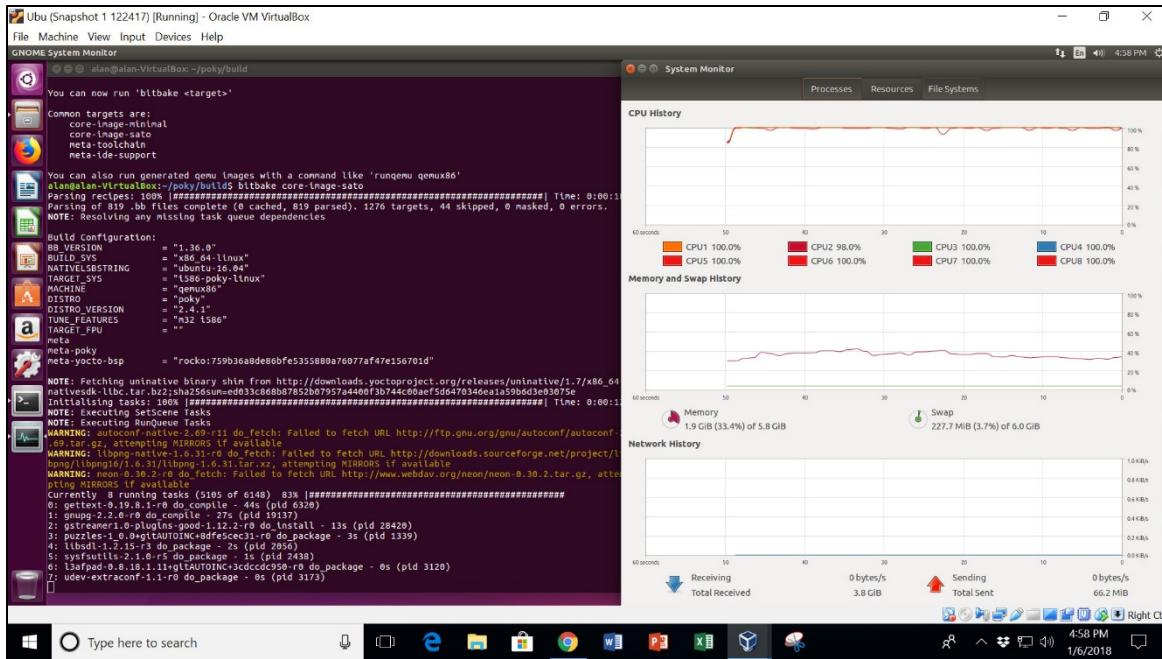
In my last blog, I observed that running Yocto Linux builds with all 16 threads of my new AMD Ryzen 7 1700X machine would always crash. Running under VirtualBox and only using one thread always worked; but it took seven hours. Could I achieve a compromise?

In [Episode 29](#) of the MinnowBoard Chronicles, I described how I RMA'd my original 2017 work week 9 Ryzen 7 1700X CPU, and got a new one with a production date stamp of work week 37. Luckily, this made the segmentation faults entirely go away during my Yocto image builds. I could play Gears of War 4 for hours without troubles. But, consistently, the new system would kick me out to the login screen after about 4,000 tasks into the 6,148 tasks executed to do a Yocto QEMU build on my Linux partition using all 16 threads. So, was I back where I started?

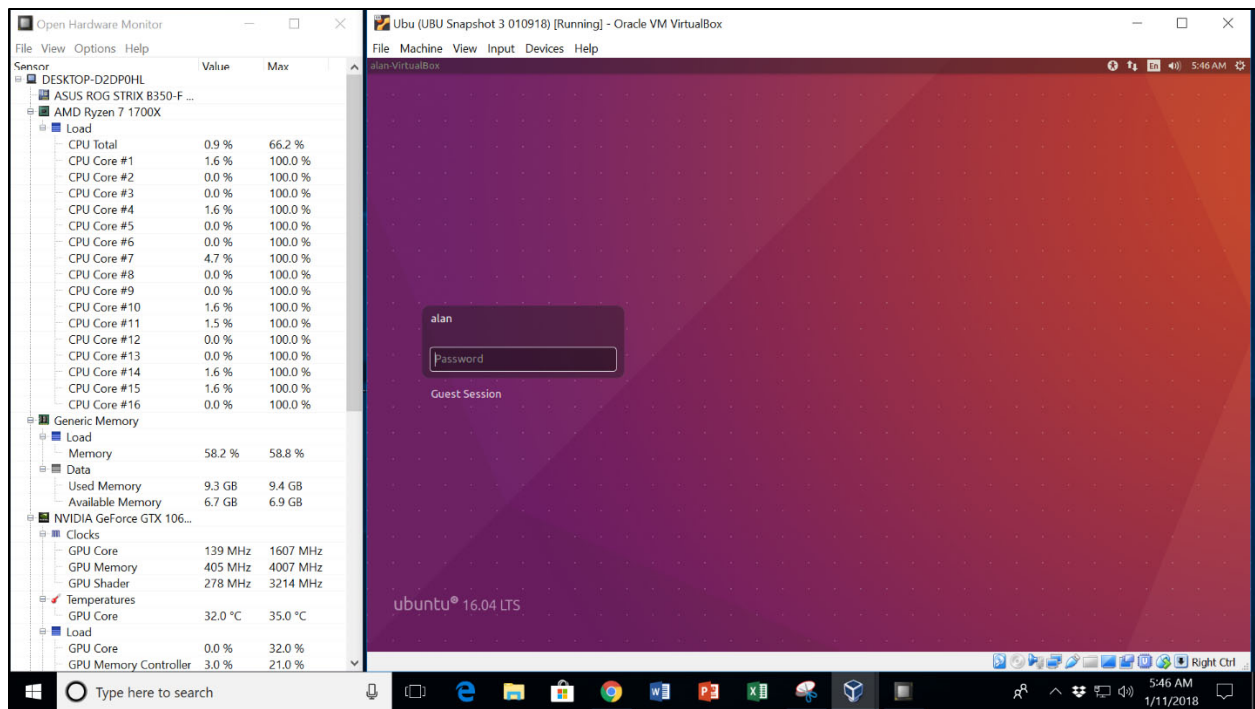
Googling this online and finding nothing, I decided to go back and use VirtualBox like I had done in earlier episodes of the Chronicles. And this worked like a charm; but by default VirtualBox used only one thread of my 16-thread AMD CPU. Luckily, it was easy enough to adjust the number of cores up in VirtualBox's settings. I decided to respect its recommendations to leave 8 cores available to Windows and have 8 decided to my Ubuntu VM for the Yocto builds.



And then, firing up all eight cores for the build:



Everything looked good so far, until I stepped away for coffee. When I came back – bang! – it



was back to the logon screen again!

The good news is, if I go down one core, down to seven cores, the build will complete successfully 100% of the time.

There seems to be some strange “num\_cores – 1” thing going on here: if I max out the core utilization, it always fails; if I go down by one, it always succeeds. If anyone knows why this might be, please do feel free to drop me a note – it’s very strange.

Now that I’m up and running again, I can’t wait to finish a MinnowBoard Turbot Yocto image build, and begin using [SourcePoint](#) for some serious debugging.

## Episode 31: First attempts to debug the Linux kernel

*March 11, 2018*

In Episode 30, I finally succeeded in building a Yocto Linux image for the MinnowBoard. But, it won't boot! Is it time to drag out a copy of SourcePoint to help?

In the [Chronicles Episode 30, Using all 16 Threads on my Ryzen?](#), I finally had some success in building a Yocto Linux image for QEMU. I can't seem to take advantage of all 16 threads, because the build crashes consistently when the thread count is maxed out. But, in the grand scheme of things, I've decided not to spend too much time debugging that. The build just takes a little longer than it normally would running at full tilt: normally it completes in about 45 minutes or so. I have my eye on the prize of building a real Linux embedded image for the MinnowBoard and running it successfully.

As a warm-up and a refresher, since a lot of things have changed since I last tried this (RMA on my AMD CPU, moving from Debian to Ubuntu, new version of YP Core – Rocko 2.4.1, etc.), I wanted to first install a copy of Ubuntu Linux on the MinnowBoard. It is easy enough to just follow the instructions at the MinnowBoard.org tutorial page, [Installing Ubuntu 16.04.3 LTS](#). This worked like a charm, just like it did way back in [Episode 24, New MinnowBoard, New PC, and a nod to Netgate](#). It's worth noting that a full install of Ubuntu Desktop runs very slowly on the MinnowBoard, but for me this is just a proof-of-concept and a learning experience, so that's fine.

Feeling confident, it was time to build a fresh image for the MinnowBoard using the Yocto Project. Things have changed a bit since I last tried this, not the least of which is that we are now on the “Rocko” release of the YP. I followed the instructions in the [Yocto Quick Start Guide](#), that describes clearly how to build an image for the MinnowBoard Turbot. And it took multiple runs before the image would build; but finally it came out.

Having had success in building a Yocto Linux image, it was time to try to install it on my MinnowBoard. Just as before, this is accomplished by inserting the USB stick with the image files into the board, and then hitting F2 while powering up to go into the UEFI menu. Selecting “Boot Manager” followed by “EFI USB Device” starts the boot process:



## Boot Option Menu

```

EFI Internal Shell
EFI Network 0008A209B492 IPv4
EFI Network 0008A209B492 IPv6
EFI Misc Device
EFI USB Device

```

↑ and ↓ to change option, ENTER to select an option, ESC to exit

Device Path :  
 PciRoot(0x0)/Pci(0x14,0x0)/US  
 B(0x0,0x0)

Alas, I got the same issue as I did way back in [Episode 25, Yocto builds for the MinnowBoard and the Portwell Neptune Alpha](#); the boot process runs up a point and then just hangs:

```

3.3723321 sst: IPv6, IPv4 and MPLS over IPv4 tunneling driver
3.3723361 NET: Registered protocol family 17
3.3723391 NET: Registered protocol family 35
3.3723411 Key type dns_resolver registered
3.3723421 microcode: sig=0x30679, pf=0x1, revision=0x0
3.3723431 microcode: Microcode Update Driver: v2.01 (CiprianBalasoiu@redhat.com)
3.3723441 sst: version of gen_mmc/dmcc engaged
3.3723451 Mmc loaded, crc32=0xc302-generic
3.3723461 Key type encrypted registered
3.3723471 random: fast init done
3.3723481 rng2: new high speed SMHC card at address 0007
3.3723491 rnc1k2: rnc2:0007 20320 29.8 GIB
3.3723501 rnc1k2: p1 p2 p3
3.3723511 console [netcon0] enabled
3.3723521 netconsole: network logging started
3.3723531 rtc_cmos 00:00: setting system clock to 2016-01-28 00:00:23 UTC (
453939223)
3.3723541 ALSA device list:
3.3723551 No soundcards found.
3.3723561 usb 1-2: new high-speed USB device number 3 using xhci_hcd
3.3723571 clocksource: Switched to clocksource tsc
3.3723581 usb 1-2:1.0: USB hub found
3.3723591 usb 1-2:1.0: 4 ports detected
3.3723601 f100m: intel801f's (f10) is primary device
3.3723611 usb 1-2:1.1: new low-speed USB device number 4 using xhci_hcd
3.3723621 Console: switching to colour frame buffer device 240x67
3.3723631 f101: 0000:00:02:0: f10: intel801f's frame buffer device
3.3723641 rd: Waiting for all devices to be available before autodetect
3.3723651 rd: If you don't use raid, use raid=mountdetect
3.3723661 rd: Autodetecting RAID arrays.
3.3723671 rd: Scanned 0 and added 0 devices.
3.3723681 rd: autoren ...
3.3723691 rd: ... autoren DONE.
3.3723701 Waiting for root device PRRUJID-Safec305-24af-4e50-604f-69c17f3e277...
3.3723711 input: Logitech USB Keyboard as /devices/pci0000:00/0000:00:14.0/usb1/1-2/1-2.1.1.0/0003:0469:0001:0000:0000:0000:0000:0000/input/input4
3.3723721 hid-generic 0003:0469:0001:0000:0000:0000:0000:0000: input: USB HID v1.10 Keyboard (Logitech USB Keyboard) on usb-0000:00:14.0-2.1.1.1.0-0003:0469:0001:0000:0000:0000:0000:0000
3.3723731 input: Logitech USB Keyboard as /devices/pci0000:00/0000:00:14.0/usb1/1-2/1-2.1.1.0/0003:0469:0002:0000:0000:0000:0000:0000/input/input5
3.3723741 hid-generic 0003:0469:0002:0000:0000:0000:0000:0000: input: USB HID v1.10 Device (Logitech USB Keyboard) on usb-0000:00:14.0-2.1.1.1.0-0003:0469:0002:0000:0000:0000:0000:0000
3.3723751 sst: 0:0:0:0: Direct-access Generic USB 1.0a PQ: 0 PPS: 2
3.3723761 sd 0:0:0:0: Attached scsi generic sg0 type 0
3.3723771 usb 1-2.2: new low-speed USB device number 5 using xhci_hcd
3.3723781 sd 0:0:0:0: [sda] 3991200 512-byte logical blocks: (1.99 GB/1.86 GiB)
3.3723791 sd 0:0:0:0: [sda] Write Protect is off
3.3723801 sd 0:0:0:0: [sda] Write cache: disabled, read cache: enabled, doesn't support WPO or RW
3.3723811 sda: sda1
3.3723821 sd 0:0:0:0: [sda] Attached SCSI removable disk
3.3723831 input: Logitech USB-PS/2 Optical Mouse as /devices/pci0000:00/0000:00:14.0/usb1/1-2/1-2.2/1-2.2.1.0/0003:0469:0002:0000:0000:0000:0000:0000/input/input6
3.3723841 hid-generic 0003:0469:0002:0000:0000:0000:0000:0000: input: USB HID v1.10 Mouse (Logitech USB-PS/2 Optical Mouse) on usb-0000:00:14.0-2.2.1.0-0003:0469:0002:0000:0000:0000:0000:0000

```

The boot process stalls right after it seems to be enumerating the USB keyboard and then mouse. I tried a lot of different things to get past this: get rid of the USB hub that I'm using, ditching the mouse, swapping ports, pulling the keyboard USB port out and putting it back in again, etc.

So, with all this time going by, I still haven't managed to get my own Linux image onto the MinnowBoard. It was time to drag out the "big guns": a tool that would help me identify root cause in the code as to why the image would not build. It was time to use our hardware-assisted debugger, [SourcePoint](#). With its capabilities of viewing the offending code, setting breakpoints, single-stepping through the code, and finally trace capabilities, I should be able to see what's going on.

The first thing I did was to power up the MinnowBoard, and have it start booting off the USB stick. Powering up the emulator, I used JTAG to halt the boot process somewhere close to where the USB mouse enumeration is failing:

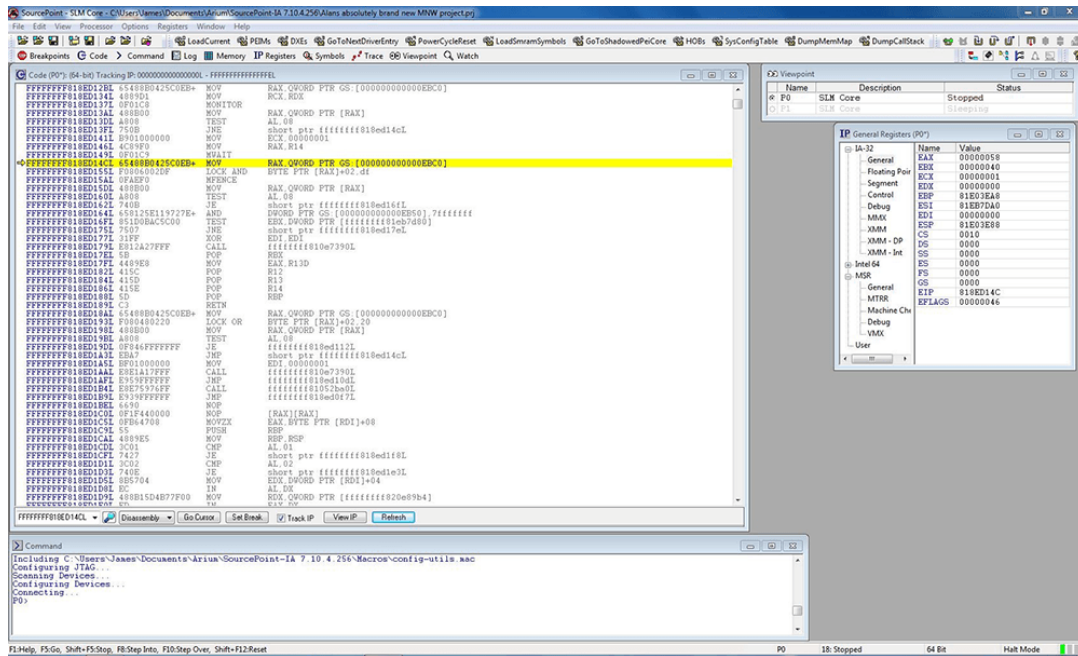
```

0.7253681 microcode: sig=0x30679, pf=0x1, revision=0x0
0.7298981 microcode: Microcode Update Driver: v2.01 <tigran@aivazian.fsnet.co.u
0.7299181 SSE version of gcm_enc/dec engaged.
0.7371551 Btrfs loaded, crc32c=crc32c-generic
0.7418631 Key type encrypted registered
0.7940131 console [netcon0] enabled
0.8030471 clocksource: Switched to clocksource tsc
0.8128461 netconsole: network logging started
0.8201031 rtc_cmos 00:00: setting system clock to 2016-01-28 00:00:31 UTC (14539
0.8360381 ALSA device list:
0.8416271   No soundcards found.
0.8477301 md: Waiting for all devices to be available before autodetect
0.8596931 md: If you don't use raid, use raid=noautodetect
0.8749051 md: Autodetecting RAID arrays.
0.8822841 md: Scanned 0 and added 0 devices.
0.8896611 md: autorun ...
0.8951311 md: ... autorun DONE.
0.9010471 Waiting for root device PARTUUID=204d4bdd-e1b9-4eb5-9c67-c06d33b62845.
0.9341211 mmc2: new high speed SDHC card at address 0007
0.9474731 mmcblk2: mmc2:0007 SD32G 28.8 GiB
0.9582961 usb 1-2: new high-speed USB device number 3 using xhci_hcd
0.9741971 random: fast init done
0.9879831 mmcblk2: p1 p2 p3
4.1449541 hub 1-2:1.0: USB hub found
4.1550281 hub 1-2:1.0: 4 ports detected
4.5004761 usb 1-2.1: new low-speed USB device number 4 using xhci_hcd
4.6657311 input: Logitech USB Keyboard as /devices/pci0000:00/0000:00:14.0/usb1/1
0003:046D:C31C.0001/input/input3
4.7437821 hid-generic 0003:046D:C31C.0001: input: USB HID v1.10 Keyboard [Logitec
usb-0000:00:14.0-2.1/input0
4.7788871 input: Logitech USB Keyboard as /devices/pci0000:00/0000:00:14.0/usb1/1
0003:046D:C31C.0002/input/input4
4.8092381 scsi 0:0:0:0: Direct-Access Generic UDISK 1.0a PQ: 0 ANSI
4.8245371 sd 0:0:0:0: Attached scsi generic sg0 type 0
4.8248091 sd 0:0:0:0: [sda] 3891200 512-byte logical blocks: (1.99 GB/1.86 GiB)
4.8250131 sd 0:0:0:0: [sda] Write Protect is off

```

If you look carefully at the two outputs, you can see that SourcePoint halted the code flow right after the message “Write Protect is off”, which is about six lines of output above where the system hung in the prior screenshot.

The SourcePoint screen shows that only one of the cores is running (the second core is sleeping from the Viewpoints window); the General Purpose Registers (GPRs) are displayed; and the Code window shows where we are in the boot code:



The information in the Code window isn't particularly edifying to me. I do see a couple of instructions I haven't tripped across before in my UEFI travels, such as "LOCK AND" and "MFENCE"; but without source code, it's hard to see what's going on.

Just for reference's sake, here is what the [Intel Software Developer's Manual, Volume 2B](#) says:

## LOCK

*Causes the processor's LOCK# signal to be asserted during execution of the accompanying instruction (turns the instruction into an atomic instruction). In a multiprocessor environment, the LOCK# signal ensures that the processor has exclusive use of any shared memory while the signal is asserted.*

*In most IA-32 and all Intel 64 processors, locking may occur without the LOCK# signal being asserted. See the "IA-32 Architecture Compatibility" section below for more details.*

## MFENCE

*Performs a serializing operation on all load-from-memory and store-to-memory instructions that were issued prior the MFENCE instruction. This serializing operation guarantees that every load and store instruction that precedes the MFENCE instruction in program order becomes*

*globally visible before any load or store instruction that follows the MFENCE instruction. The MFENCE instruction is ordered with respect to all load and store instructions, other*

*MFENCE instructions, any LFENCE and SFENCE instructions, and any serializing instructions (such as the CPUID instruction). MFENCE does not serialize the instruction stream.*

*Weakly ordered memory types can be used to achieve higher processor performance through such techniques as out-of-order issue, speculative reads, write-combining, and write-collapsing. The degree to which a consumer of data recognizes or knows that the data is weakly ordered varies among applications and may be unknown to the producer of this data. The MFENCE instruction provides a performance-efficient way of ensuring load and store ordering between routines that produce weakly-ordered results and routines that consume that data.*

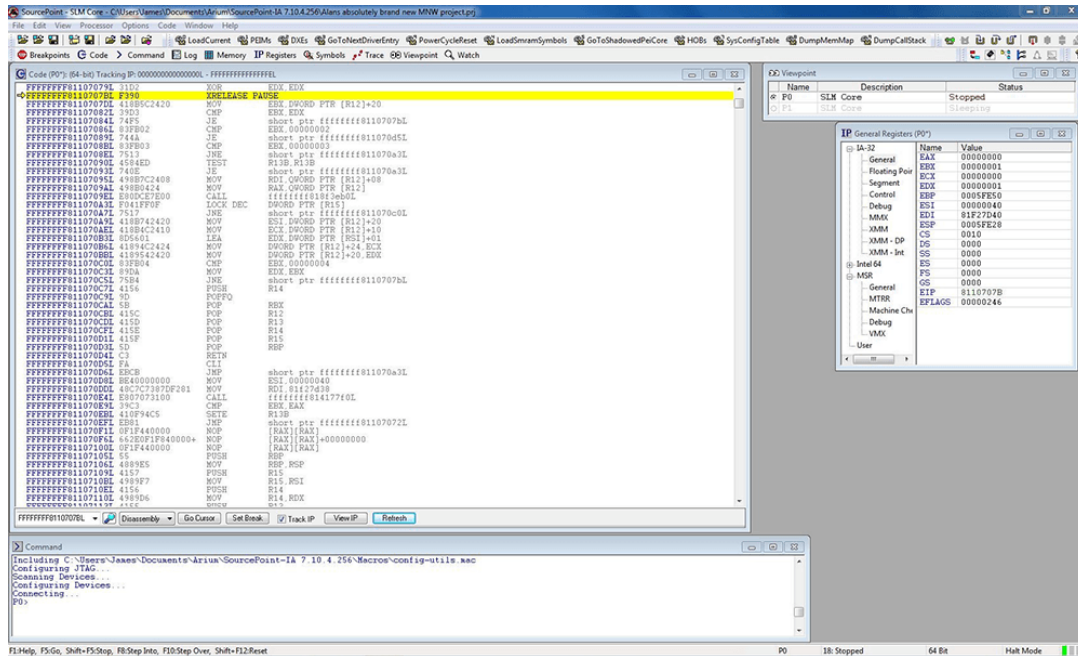
*Processors are free to fetch and cache data speculatively from regions of system memory that use the WB, WC, and WT memory types. This speculative fetching can occur at any time and is not tied to instruction execution. Thus, it is not ordered with respect to executions of the MFENCE instruction; data can be brought into the caches speculatively just before, during, or after the execution of an MFENCE instruction.*

Rather than continue any investigations into the use of these instructions at this time, I decided to then let the boot process continue until it hangs, by using the Run button; and then halt it again and see what I could learn.

Interestingly, nothing further comes out on the screen over the HDMI connection. I realize that I should have had the serial output capture on my Mac's CoolTerm application as a backup, but that's for later.

But we're at a different point in the code now:





Again, we are halted at an instruction “XRELEASE PAUSE” that I am not familiar with. The [SDM](#) reveals XACQUIRE and XRELEASE as “prefix hints”:

*The XRELEASE prefix hint can only be used with the following instructions (also referred to as XRELEASE-enabled when used with the XRELEASE prefix):*

- *Instructions with an explicit LOCK prefix (F0H) prepended to forms of the instruction where the destination operand is a memory operand: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCHG8B, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, and XCHG.*
- *The XCHG instruction either with or without the presence of the LOCK prefix.*
- *The “MOV mem, reg” (Opcode 88H/89H) and “MOV mem, imm” (Opcode C6H/C7H) instructions. In these cases, the XRELEASE is recognized without the presence of the LOCK prefix.*

*The lock variables must satisfy the guidelines described in Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, Section 16.3.3, for elision to be successful, otherwise an HLE abort may be signaled.*

This is a little obscure. I don't see reference to an "XRELEASE PAUSE" anywhere in the SDM, or just about anywhere in Google. But looking at the definition of the PAUSE instruction might be educational:

*Improves the performance of spin-wait loops. When executing a "spin-wait loop," processors will suffer a severe performance penalty when exiting the loop because it detects a possible memory order violation. The PAUSE instruction provides a hint to the processor that the code sequence is a spin-wait loop. The processor uses this hint to avoid the memory order violation in most situations, which greatly improves processor performance. For this reason, it is recommended that a PAUSE instruction be placed in all spin-wait loops.*

*An additional function of the PAUSE instruction is to reduce the power consumed by a processor while executing a spin loop. A processor can execute a spin-wait loop extremely quickly, causing the processor to consume a lot of power while it waits for the resource it is spinning on to become available. Inserting a pause instruction in a spinwait loop greatly reduces the processor's power consumption.*

*This instruction was introduced in the Pentium 4 processors, but is backward compatible with all IA-32 processors. In earlier IA-32 processors, the PAUSE instruction operates like a NOP instruction. The Pentium 4 and Intel Xeon processors implement the PAUSE instruction as a delay. The delay is finite and can be zero for some processors. This instruction does not change the architectural state of the processor (that is, it performs essentially a delaying no-op operation).*

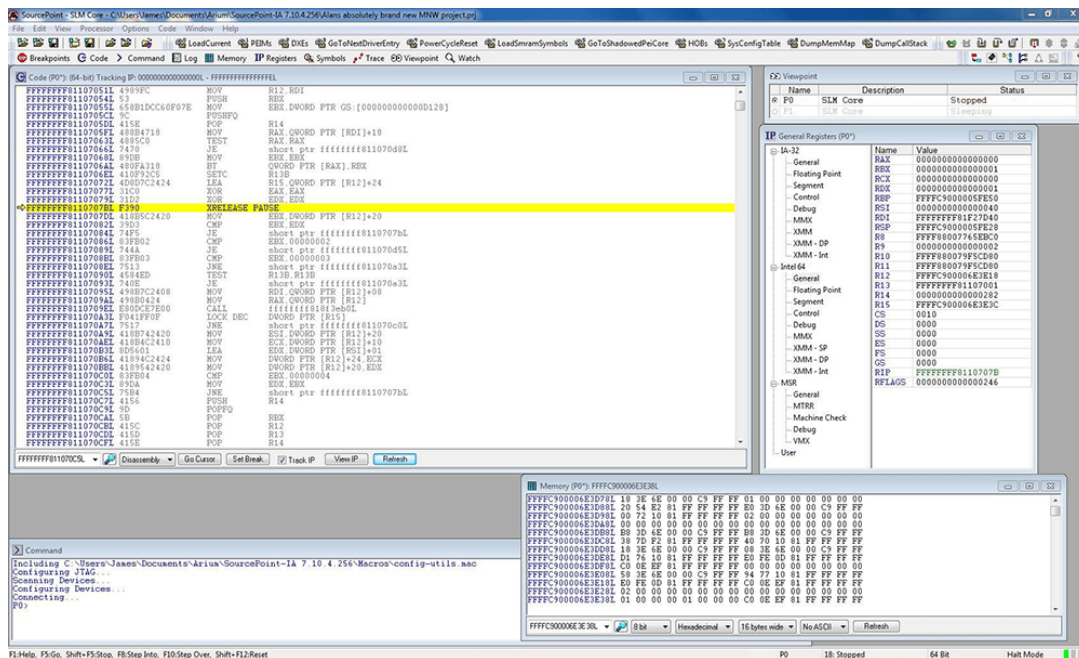
Presuming that the disassembled code is correct, I can only guess that we are in some sort of critical time loop, maybe a CPU deadlock. Will it ever exit? There's only one way to find out: keep stepping through the code and use SourcePoint to provide some insight into the loop.

There are four instructions that are part of the loop:

FFFFFFFF8110707BL	F390	XRELEASE PAUSE
FFFFFFFF8110707DL	418B5C2420	MOV EBX, DWORD PTR [R12]+20
FFFFFFFF81107082L	39D3	CMP EBX, EDX
FFFFFFFF81107084L	74F5	JE SHORT PTR FFFFFFFFF8110707B



You can see from the screenshot that both EBX and EDX are set to 1 currently, so the loop will keep executing until EBX gets changed by the MOV instruction, that sets EBX to the value contained at the address 20 bytes offset from the contents of R12. We need to see the Intel 64 GPRs in order to determine what address is contained within R12, and then to peek at the address offset 20 bytes from that:



R12 contains FFFFC900006E3E18, and the x'20' offset yields FFFFC900006E3E38, and looking at the memory display windows shows that address containing 0000000100000001 (remember that x86 is little-endian). Taking the DWORD value always yields 1 being put into EBX. This is a deadlock; unless some other process changes the value at address FFFFC900006E3E38, it will never exit the loop. And that is just what I found.

I did try to tinker with the contents of the EDX register, and also the value at FFFFC900006E3E38, and did manage to get the code to temporarily exit the four-instruction loop. But, it always came back to the deadlock, sooner or later.

There are quite a few different directions I could go at this point, including using some of the x86 Trace features like Branch Trace & Store (BTS) to follow the code flow leading up to this problem. But, realistically, admitting that, one way or the other, I'm lost without source code, that's become the next step: creating my Yocto Linux build with source and symbols, and

loading them into SourcePoint so I can see exactly what is happening in this area of the code. There are no guarantees that seeing the source will help me debug this problem, but it's a start.

This looks like a big challenge. Even though I've read [Robert Love's Linux Kernel Development](#) from cover to cover, I am by no means a Linux expert (which should be obvious to anyone following these MinnowBoard Chronicles episodes), let alone understanding the operations of the kernel well enough to figure out why it won't boot. There is an entire document at [Yocto Project Linux Kernel Development Manual](#) that should help me, though. We'll see how it goes!

With source code, I should be able to see what code is accessing FFFFC900006E3E18, and set a Data Write breakpoint at that point to see what is putting data in there. I'll also be able to use SourcePoint's support for the powerful x86 Trace features (check out the [eBook](#); requires registration) to see backwards in time and maybe get some insight as to why I'm stuck in the deadlock. Should be fun!

## Afterword

Phew! After about 15 months of exploring and writing about the MinnowBoard, I'm ready to take a break. I learned a huge amount about Intel Architecture, UEFI, Linux, JTAG, and a plethora of other technical topics during this journey. I've always absorbed information better when I've subsequently written about it. I can see myself referring back to this book in the future, recalling some of the more obscure technology that I've dabbled with.

I'm hoping that this book makes for a good, albeit long, story. Maybe others will use it as a reference for some of the technical topics therein. It's easy enough to search through the contents, looking for tips. And perhaps someone has a problem with the design or debug of a particular piece of hardware, firmware, or software that I ran into and solved, and will learn in this book how to move forward. I'd be happy about that.

So, I'm ready to move on to write about other topics. So much to write about, so little time.

On the other hand, there are a few nagging MinnowBoard issues that I never did manage to solve in the last year or so. I'm already starting to feel an itch to take another look at them. Will there be an update to this book in the future, or perhaps a Volume 2? We'll see!