Generic IP independent BIOS Signing and Parsing

Gahan Saraiya

Institute of Technology
Nirma University

March 17, 2020

Outline

1. About the Project
2. Motivation
3. BIOS: Basics
4. Requirement Specification
5. My Contribution
6. Future Scope
7. Study of existing architecture for hotspot
8. Analysis and Gathering precise Problem Information
9. Requirement Specification
10. Why JSON?
11. Implementation of Parsing
12. Future Scope

About the Project

In general to generate BIOS image (*.rom file), compilation of XYZ.c (source code) has to be done, this compilation not only involves compilation of DXE driver, PEI driver, EFI Application but also includes pre-processing checks, compression of raw files which takes huge amount of time depending on the system configuration. Implementation of this project aids in reduction of this compilation time.

Motivation: Stakeholders

- BIOS development Team
- Automation Team
- Validation Team
- Other Development Team

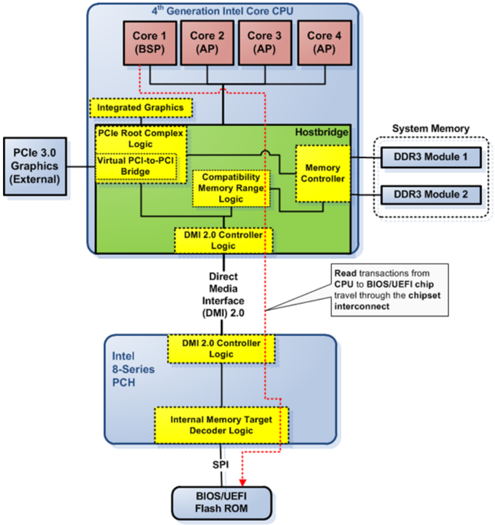
Motivation: Issues/Challenges to the industry (Towards my contribution)

- BIOS image generation: Compilation of whole source code
- More Time complexity: Compilation of source code to generate BIOS image

BIOS: Basics

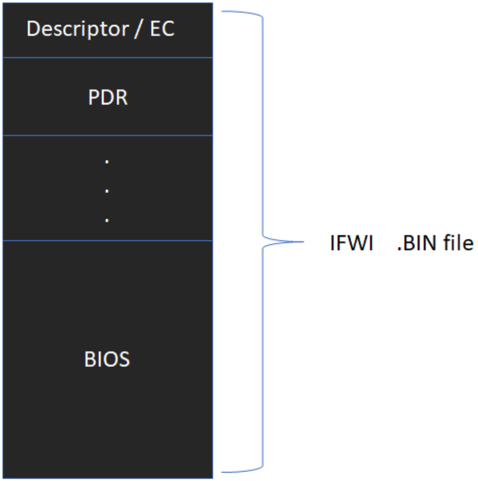
- Set of Software Routines
  - Initialize and test hardware on start
  - Provides the OS with a generic hardware abstraction
- the BIOS must do its job before your computer can load its operating system and applications

BIOS: Architecture



4th Generation Intel Core CPU
Core 1 (BSP)
Core 2 (AP)
Core 3 (AP)
Core 4 (AP)
Integrated Graphics
PCIe Root Complex Logic
Virtual PCI-to-PCI Bridge
Hostbridge
Compatibility Memory Range Logic
Memory Controller
DMI 2.0 Controller Logic
PCIe 3.0 Graphics (External)
System Memory
DDR3 Module 1
DDR3 Module 2
Direct Media Interface (DMI) 2.0
Read transactions from CPU to BIOS/UEFI chip travel through the chipset interconnect
Intel 8-Series PCH
DMI 2.0 Controller Logic
Internal Memory Target Decoder Logic
SPI
BIOS/UEFI Flash ROM

BIOS: Firmware Image

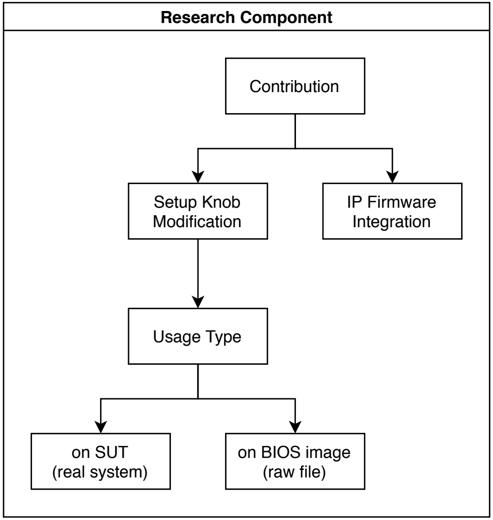


Requirement Specification

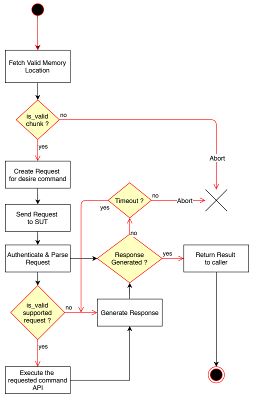
- Visual Studio C/C++ IDE
- Python 3
- Visual Studio Code
- Memory Access Interfaces[1]

[1] itp, itp2, windows, linux, ltb, dci, efi shell, etc.

My Contribution towards issues/challenges

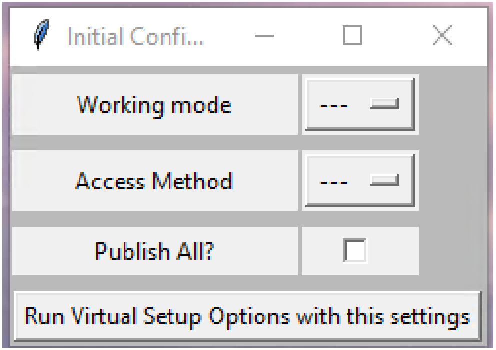


Setup Knobs Modification: Process Flow I


Fetch Valid Memory Location
is_valid chunk ?
no
yes
Create Request for desire command
Send Request to SUT
Authenticate & Parse Request
is_valid supported request ?
no
yes
Execute the requested command API
Generate Response
Response Generated ?
yes
no
Timeout ?
yes
no
Abort
Abort
Return Result to caller

Setup Knobs Modification Flow on SUT

Setup Knobs Modification: Implementation Snaps I



Working mode: ---

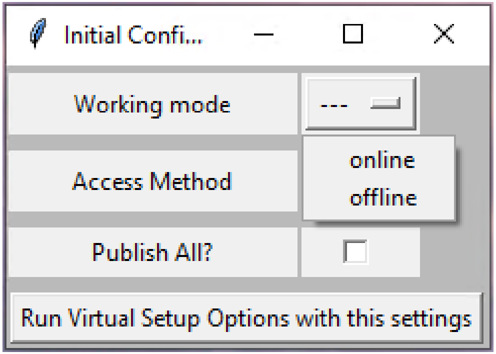
Access Method: ---

Publish All? ☐

Run Virtual Setup Options with this settings

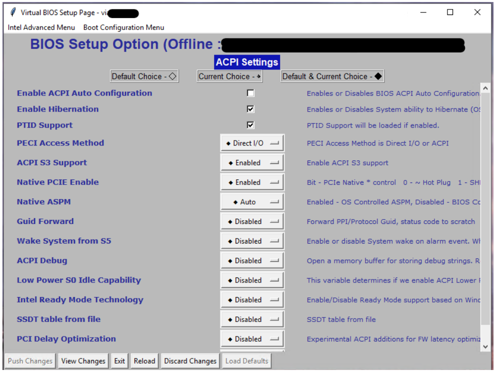
Menu to Select initial configuration for work

Setup Knobs Modification: Implementation Snaps II



Available work mode for the system: Online and Offline

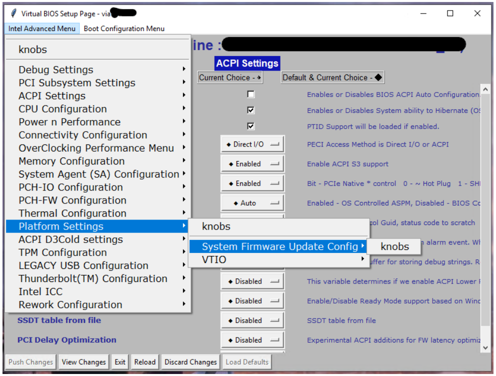
Setup Knobs Modification: Implementation Snaps III



Enable ACPI Auto Configuration	☐	Enables or Disables BIOS ACPI Auto Configuration.
Enable Hibernation	☑	Enables or Disables System ability to Hibernate (OS
PTID Support	☑	PTID Support will be loaded if enabled.
PECI Access Method	⬩ Direct I/O	PECI Access Method is Direct I/O or ACPI
ACPI S3 Support	⬩ Enabled	Enable ACPI S3 support
Native PCIE Enable	⬩ Enabled	Bit - PCIe Native * control 0 - ~ Hot Plug 1 - SHI
Native ASPM	⬩ Auto	Enabled - OS Controlled ASPM, Disabled - BIOS Co
Guid Forward	⬩ Disabled	Forward PPI/Protocol Guid, status code to scratch
Wake System from S5	⬩ Disabled	Enable or disable System wake on alarm event. Wh
ACPI Debug	⬩ Disabled	Open a memory buffer for storing debug strings. R
Low Power S0 Idle Capability	⬩ Disabled	This variable determines if we enable ACPI Lower P
Intel Ready Mode Technology	⬩ Disabled	Enable/Disable Ready Mode support based on Winc
SSDT table from file	⬩ Disabled	SSDT table from file
PCI Delay Optimization	⬩ Disabled	Experimental ACPI additions for FW latency optimiz

Setup Options listed under ACPI Configurations

Setup Knobs Modification: Implementation Snaps IV



Virtual BIOS Setup Page - via [redacted]

Intel Advanced Menu | Boot Configuration Menu

knobs

- Debug Settings ▸
- PCI Subsystem Settings ▸
- ACPI Settings ▸
- CPU Configuration ▸
- Power n Performance ▸
- Connectivity Configuration ▸
- OverClocking Performance Menu ▸
- Memory Configuration ▸
- System Agent (SA) Configuration ▸
- PCH-IO Configuration ▸
- PCH-FW Configuration ▸
- Thermal Configuration ▸
- Platform Settings ▸
- ACPI D3Cold settings ▸
- TPM Configuration ▸
- LEGACY USB Configuration ▸
- Thunderbolt(TM) Configuration ▸
- Intel ICC ▸
- Rework Configuration ▸

Platform Settings submenu: knobs; System Firmware Update Config ▸ (knobs); VTIO ▸

ine : [redacted]

ACPI Settings

Current Choice - ◆ | Default & Current Choice - ◆

	☐	Enables or Disables BIOS ACPI Auto Configuration.
	☑	Enables or Disables System ability to Hibernate (O
	☑	PTID Support will be loaded if enabled.
	◆ Direct I/O	PECI Access Method is Direct I/O or ACPI
	◆ Enabled	Enable ACPI S3 support
	◆ Enabled	Bit - PCIe Native * control 0 - ~ Hot Plug 1 - SH
	◆ Auto	Enabled - OS Controlled ASPM, Disabled - BIOS C
		col Guid, status code to scratch
		alarm event. W
		uffer for storing debug strings. R
	◆ Disabled	This variable determines if we enable ACPI Lower
	◆ Disabled	Enable/Disable Ready Mode support based on Win
SSDT table from file	◆ Disabled	SSDT table from file
PCI Delay Optimization	◆ Disabled	Experimental ACPI additions for FW latency optimi

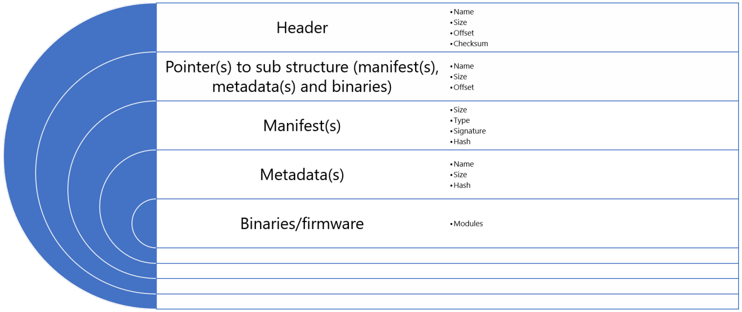
Push Changes | View Changes | Exit | Reload | Discard Changes | Load Defaults

Navigating through BIOS setup page

Setup Knobs Modification: Outcome

- Cross platform usage
- API as a driver in BIOS Firmware
- Generic solution for usage types - on **SUT**, on **BIOS image**
- Information parsing and simulation
- Realtime sync for simulation changes
- Seamless Integration

IP firmware Integration: Structure of Module



Proposed Structure for firmware signing

IP firmware Integration: Outcome

- Removal of IP dependency during firmware loading
- IP Subsystem :
  - Loader and Verifier
  - IP is always consumer
- Signature verification using SHA hash algorithm
- Seamless Integration of any other hash algorithm for verification
- Hardware based and Software based verification support
- Prevent common security threats
- Allow easier OEM adoption and modification based on the respective design
- Reusability/Portability of design across many IPs
- Generic design which supports any new IP integration

Future Scope

- Study of existing hotspots for automation
- Analyzing and gathering requirements of automation to hotspot
- Implementing and managing platform to keep up-to-date the user base of the framework

Phase 1: Study of existing architecture for hotspot

- Identifying changes of two different BIOS Image of different check-ins
- Lookup of module by GUID and vice-versa
- Exposed source code support for OEM information
- Runtime BIOS Support for temporary UEFI variable creation

Phase 2: Analysis and Gathering precise Problem Information

- Debugging via comparing Setup Knobs
- Lookup of order of the module in BIOS Image as file system
- Verification of module integration via GUID
- OEM needs to fill information which need not to compulsorily expose the source code
- Automation and Testing for **non-BIOS driver** require BIOS Support for creation of temporary UEFI variables

Requirement Specification

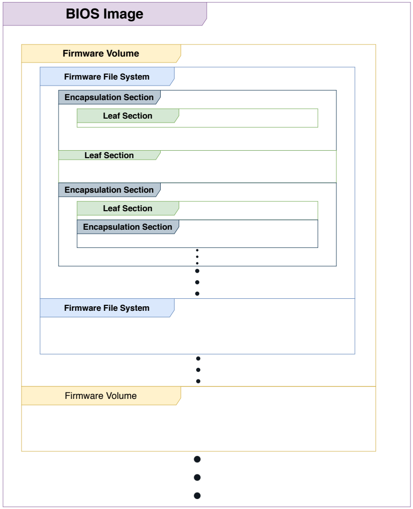
- Visual Studio C/C++
- Visual Studio Code
- Python
- JSON
- Memory Access Interfaces[2]

[2] itp, itp2, windows, linux, ltb, dci, efi shell, etc.

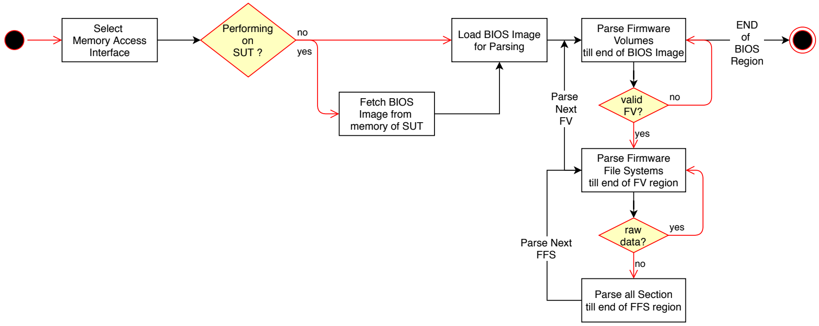
Why JSON?

- Light-weight structured-database!
- Minimal in terms of space complexity
- Easier to parse and interpret
- Portable

Implementation of Parsing: Format of BIOS Image



Implementation of Parsing: Work Flow



Select Memory Access Interface
Performing on SUT ?
no
yes
Fetch BIOS Image from memory of SUT
Load BIOS Image for Parsing
Parse Firmware Volumes till end of BIOS Image
END of BIOS Region
valid FV?
no
yes
Parse Next FV
Parse Firmware File Systems till end of FV region
raw data?
yes
no
Parse Next FFS
Parse all Section till end of FFS region

Outcome

- Human Readable interpretation of BIOS Image
- GUIDs Lookup
- Verification of existence of module by GUID
- Storing the image file system content by GUID
- Summarizing changes of two BIOS image

Future Scope

- Exposed source code support for OEM information
- Runtime BIOS Support for temporary UEFI variable creation

Thank You