Automatic BIOS Code Generation

Submitted By

Shalini Somani

13MCEC22

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INSTITUTE OF TECHNOLOGY

NIRMA UNIVERSITY

AHMEDABAD-382481

May 2015

Automatic BIOS Code Generation

Major Project

Submitted in partial fulfillment of the requirements

for the degree of

Master of Technology in Computer Science and Engineering

Submitted By

Shalini Somani

(13MCEC22)

Guided By

Prof. Pooja Shah

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INSTITUTE OF TECHNOLOGY

NIRMA UNIVERSITY

AHMEDABAD-382481

May 2015

Certificate

This is to certify that the major project entitled **"Automatic BIOS Code Generation"** submitted by **Shalini Somani (Roll No: 13MCEC22)**, towards the partial fulfillment of the requirements for the award of degree of Master of Technology in Computer Science and Engineering of Nirma University, Ahmedabad, is the record of work carried out by her under my supervision and guidance. In my opinion, the submitted work has reached a level required for being accepted for examination. The results embodied in this project, to the best of my knowledge, haven't been submitted to any other university or institution for award of any degree or diploma.

Prof. Pooja Shah
Internal Guide & Assistant Professor,
CSE Department,
Institute of Technology,
Nirma University, Ahmedabad.

Prof. Vijay Ukani
Associate Professor,
Coordinator M.Tech - CSE
Institute of Technology,
Nirma University, Ahmedabad

Dr. Sanjay Garg
Professor and Head,
CSE Department,
Institute of Technology,
Nirma University, Ahmedabad.

Dr. Ketan Kotecha
Director,
Institute of Technology,
Nirma University, Ahmedabad

Certificate

This to certify that Miss Shalini Somani (13MCEC22), a student of M.Tech CSE(Computer Science and Engineering), Institute of Technology, Nirma University was working in this organization since 11/06/2014 and carried out her thesis work titled "Automatic BIOS Code Generation". She was working in name of BIOS Engineer intern under supervision of Mrs. Nivedita Aggarwal (Mentor), and Mr. Bimod Narayanan (Manager). She has successfully completed the assigned work and is allowed to submit her dissertation report. The results embodied in this project, to the best of our knowledge, haven't been submitted to any other university or institution for award of any degree or diploma. We wish her all the success in future.

Mrs. Nivedita Aggarwal
External Guide & Client BIOS Architect,
Intel Technology India Pvt.Ltd,
Bengaluru.

Mr. Bimod Narayanan
Engineering Manager,
Intel Technology India Pvt.Ltd
Bengaluru.

Statement of Originality

I, **Shalini Somani**, Roll. No. **13MCEC22**, give undertaking that the Major Project entitled "**Title of the Project**" submitted by me, towards the partial fulfillment of the requirements for the degree of Master of Technology in **Computer Science & Engineering** of Institute of Technology, Nirma University, Ahmedabad, contains no material that has been awarded for any degree or diploma in any university or school in any territory to the best of my knowledge. It is the original work carried out by me and I give assurance that no attempt of plagiarism has been made. It contains no material that is previously published or written, except where reference has been made. I understand that in the event of any similarity found subsequently with any published work or any dissertation work elsewhere; it will result in severe disciplinary action.

_____

Signature of Student

Date:

Place:

Endorsed by

Prof. Pooja Shah

(Signature of Guide)

Acknowledgements

It gives me immense pleasure in expressing thanks and profound gratitude to **Prof. Pooja Shah**, Associate Professor, Computer Science Department, Institute of Technology, Nirma University, Ahmedabad for his valuable guidance and continual encouragement throughout this work. The appreciation and continual support he has imparted has been a great motivation to me in reaching a higher goal. His guidance has triggered and nourished my intellectual maturity that I will benefit from, for a long time to come.

It gives me an immense pleasure to thank **Dr. Sanjay Garg**, Hon'ble Head of Computer Science and Engineering Department, Institute of Technology, Nirma University, Ahmedabad for his kind support and providing basic infrastructure and healthy research environment.

A special thank you is expressed wholeheartedly to **Dr K Kotecha**, Hon'ble Director, Institute of Technology, Nirma University, Ahmedabad for the unmentionable motivation he has extended throughout course of this work.

I would also thank the Institution, all faculty members of Computer Engineering Department, Nirma University, Ahmedabad for their special attention and suggestions towards the project work.

See that you acknowledge each one who have helped you in the project directly or indirectly.

- Shalini Somani

13MCEC22

Abstract

Today, the complexity of the computer systems has grown, with processors and chipsets incorporating millions of the transistors and compatible with dozens of operating system,hundreds of platform components and thousands of hardware devices and software applications. Hence the complexity of BIOS source code is increased so it is hard to develop different BIOS for each type of platform with different favors.

Also every generation of processor (RTL) is accompanied by an XML file that consists of entire register set details for that chip. For each generation of an Silicon(IC chip), there are many flavors of its layouts each accompanied by a specific xml file. As and when the layout changes even slightly, there are few registers among thousands of registers that might change.

Currently, programmers have to manually type in the details of all the registers in their code as part of header files and source files. Therefore, whenever there is a new version of xml file, even though only few register details would have changed, tracing them and logging them to update the code is difficult. Doing it multiple times is time consuming and prone to errors.

This project is aimed at generating the Silicon initialization code from Silicon RTL XML using Silicon and feature specific configuration overrides with minimal manual intervention.

Abbreviations

BIOS	Basic Input Output System.
POC	Proof Of Concept.
EFI	Extensible Firmware Interface.
UEFI	Unified Extensible Firmware Interface.
RTL	Register Transistor Logic .
BDF	BIOS Directives Format.
POST	Power On Self Test.
PEI	Pre-EFI.
DXE	Driver Execution Environment.
BDS	Boot Device Selection.
SOC	System On Chip.

Contents

List of Figures

List of Tables

%

Chapter 1

Introduction

1.1 Overview

In this thesis report,method to automize the BIOS code generation is discussed.BIOS stands for Basic Input Output System[**?**] On a very high level, it initializes the hardware like processor, chipset, peripherals etc. and then it gives control to the OS through boot manager. The BIOS must do its job before computer can load operating system and applications. Currently Intel has migrated from legacy BIOS to EFI BIOS which is based on EFI Specification. It has standard, modular environment and have many advantages over legacy BIOS.EFI is a standard or a specification which has different phases named as SEC (Security), PEI (Pre EFI Initialization)[1], DXE (Driver Execution Environment)[1] and BDS (Boot Device Selection). There is a provision of legacy BIOS in EFI BIOS code because not all the OS are EFI compatible. The focus of this project is on the portion of BIOS that initializes the Silicon and its features.

1.2 Motivation

Here the motivation behind the work done in the thesis to generate the Silicon initialization code from Silicon RTL XML using Silicon and feature specific configuration is to minimize the manual work done and also this automization is less error prone, clearly depicted in (Figures 1.1).

If we automate the generation of source code, it would have the following benefits.

- Organized code format for BIOS development

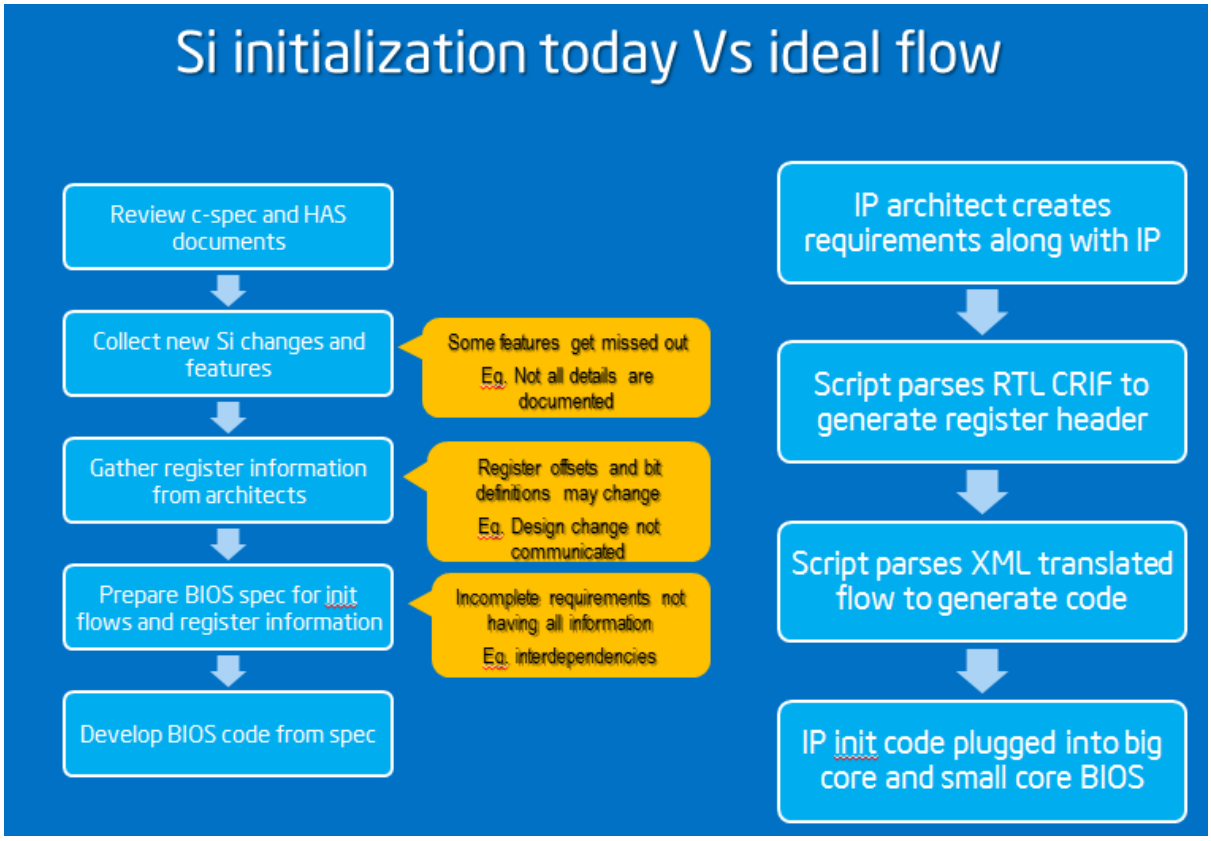

Figure 1.1: Need of project

- Avoid gaps and missing information
- Reduce development effort on Silicon specific code
- Minimize bugs and issues due to incorrect Silicon configurations and offsets
- Leverage on existing standardized XML formats for Silicon configurations

1.3 Problem Definition

Today BIOS engineers write BIOS code based on the RTL. Every generation of processor (RTL) is accompanied by an XML file that consists of entire register set details for that chip. For each generation of a Silicon(IC chip), there are many flavors of it each accompanied by a specific xml file. As and when the stepping changes even slightly,there are some configurations that need to be changed.

Currently, programmers have to manually type in the details of all the registers in their code as part of header files and source files. Therefore, whenever there is a new

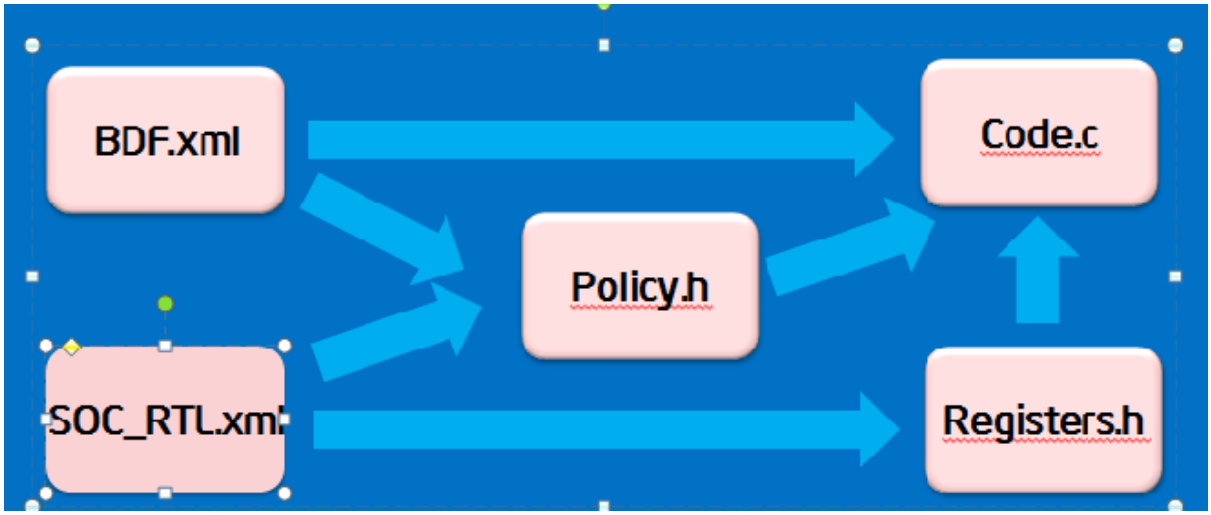

Figure 1.2: Proposed work

version of xml file, even though only few register details would have changed, tracing them and logging them to update the code is difficult. Doing it multiple times is time consuming and prone to errors.

My project is aimed at automating the generation of source code and header files, by taking xml file as input, parsing it based on the information and generating the required source code and header files.

Chapter 2

Literature Survey

2.1 Intel Architecture - Platform Overview

2.1.1 Introduction to platform

Platform encompasses all required ingredients, features, capabilities, initiatives and technologies (Figures 2.1).
The 4 major ingredients:

- Hardware - which includes Processors, boards ,memory, chipsets, etc.[2]
- Software - which includes Operating systems (OSs), compilers and applications.[2]
- Technologies - which includes Intel Virtualization Technology,Hyper-Threading Technology (HT Technology) and Intel Active Management Technology (Intel AMT).[3]
- Standards and Initiatives - WiMAX,Wi-Fi, the Wireless Verification Program, and so on.[2]

The platform is complex with lots of components on it. Every component must work as designed and there shouldn't be any conflicts between the devices on it. The Figure below shows the typical diagram of Intel Client platform 2014. It comes up with single chip solution which means CPU and PCH are in single die.[2]

2.2 Introduction to BIOS as ingredient

BIOS is the first code to be run when our PC is switched on. It acts as a layer between Operating System and Hardware. BIOS initialize the various platform components like

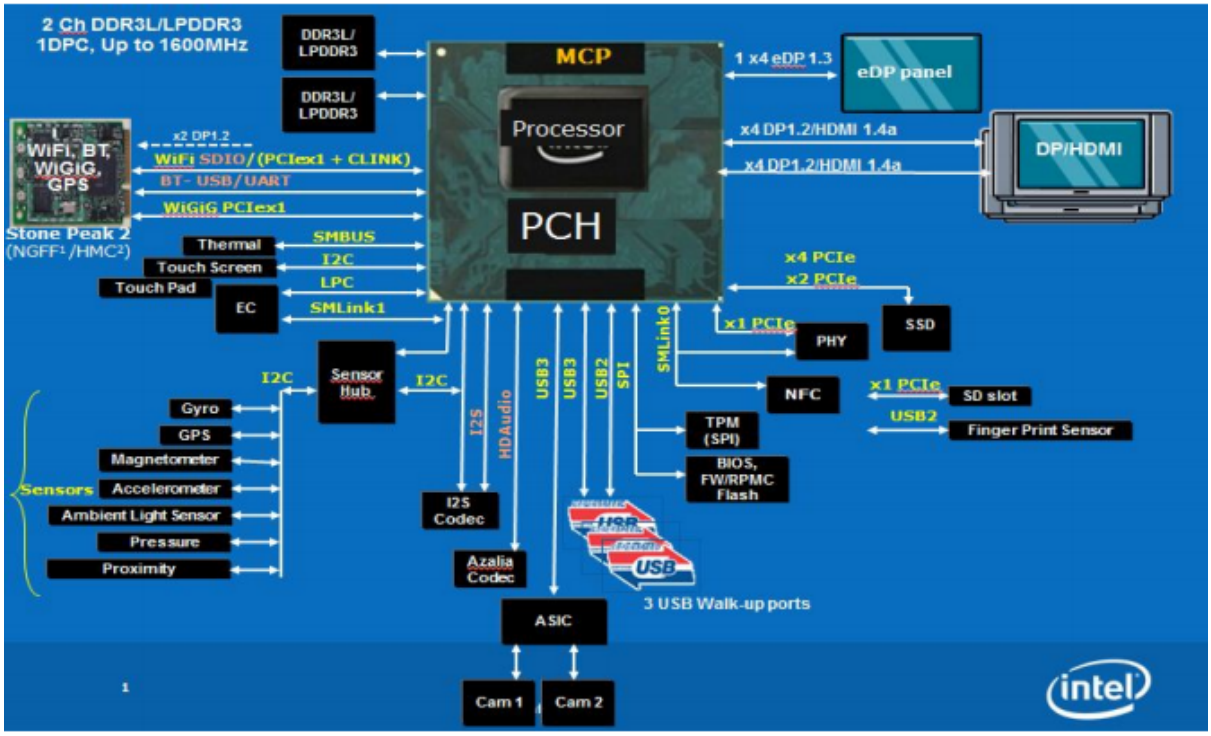

Figure 2.1: Intel platform architecture

CPU initialization, core initialization, memory and chipset initialization etc. [2] Once BIOS does its job properly then only your computer can load its operating system and applications(Figures 2.2).

The essential capacity of the BIOS is to begin a boot loader by setting up and stacking equipment. When the PC begins is controlled on, the first occupation for the BIOS is to identify,enumerate and instate framework gadgets, for example, the feature showcase card, system cards,

console and mouse, hard circle drive, optical plate drive and other hardware.[9]

At that point after the product which is hung on a fringe gadget (assigned as a 'boot gadget, for example, a hard circle or a CD/DVD is situated by BIOS.[4] After this BIOS is mainly responsible for loading and execution of the software, giving it control of the PC. This process of BIOS loading and executing software to make sure that our system is booting to OS is known as booting, which is short for bootstrapping.[4]

BIOS software is stored on a non-volatile ROM (generally flash memory) chip built into the system on the motherboard. BIOS plays a number of different roles and most important among them is loading the Operating System.[4] As soon as our system is turned on,

the microprocessor tries to execute the first instruction. Now for executing any instruction it has to be present somewhere. Also Operating System cannot have that instruction as OS is loacated in hardisk and we have not identified any device so far. So BIOS is responsible for providing those instructions.[4]
BIOS features are as follows:

- It acts as a layer between OS and Hardware.
- It gets your computer up and running.
- Initializes the hardware like Microprocessor, memory, chipset, devices, peripherals etc.[2]
- Provides Power Management functionality through ACPI.
- Loads and hands control over to the OS boot loader.
- Provides a set of standardized routines for the OS to use.
- Abstracts motherboard and silicon specifics from the OS.
- Prepares system to run an OS.
- Provides runtime services to the OS e.g. disk and video.

The BIOS is responsible for providing drivers/libraries of basic input/output functions used to operate and control the peripherals such as the keyboard, text display functions and so forth, and these software library functions are callable by external software.[3] BIOS performs power-on self-test (POST) for various hardware components in the system to ensure everything is working properly. It also provides a set of low level routines which is used by Operating System for interfacing with different hardware devices.[4]

The Advanced Configuration and Power Interface (ACPI) is a detail which

was produced to create industry regular interfaces empowering vigorous working framework (OS)-coordinated motherboard gadget arrangement and force administration of both

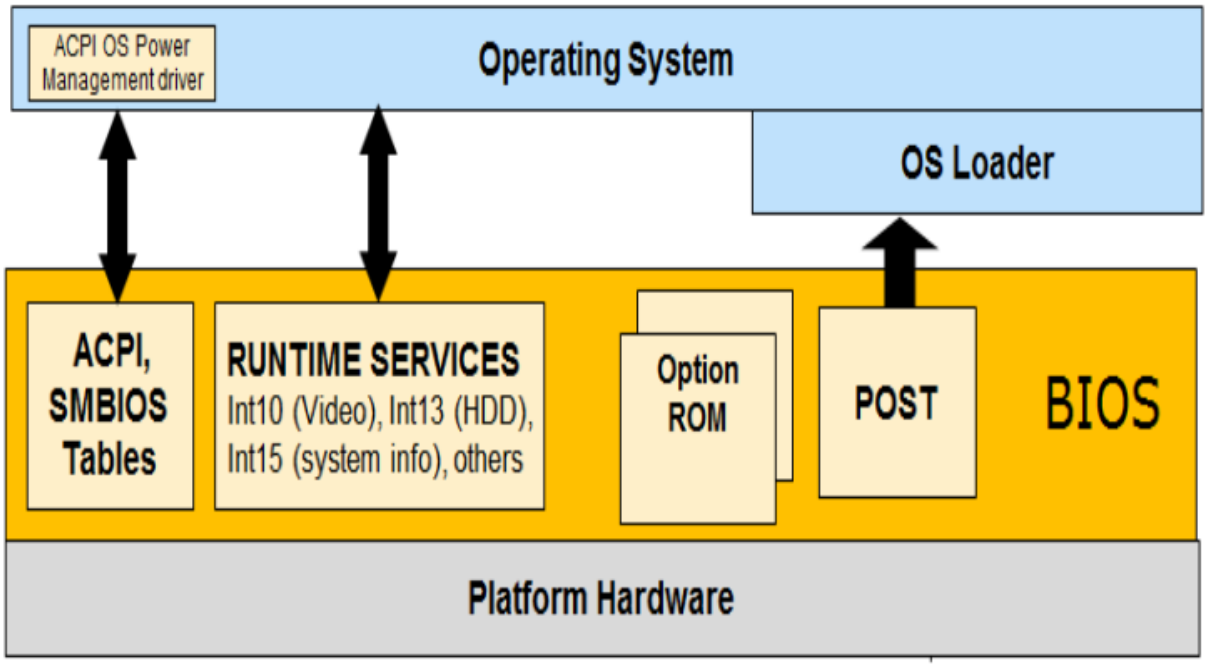

Figure 2.2: BIOS Overview

gadgets and whole systems.[4] ACPI is the key component in Operating System-coordinated

design and Power Management (OSPM).

Currently, Industry has migrated from Legacy BIOS to a standard and modular EFI. BIOS EFI BIOS offers new and improved features and edibility for code developers.[1] The difference between Legacy BIOS and EFI BIOS is shown in (Figures 2.3)

2.2.1 Power-On Self Test (POST)

Power-On Self-Test (POST)[2] refers to routines which runs immediately as soon as the PC is powered on, by almost all electronic devices.[2] Conceivably the most well known usage involves computing devices (personal computers, PDAs, networking devices, switches, intrusion detection systems and other monitoring devices), kitchen appliances, avionics, laboratory test equipments, medical equipment,etc. The routines are part of a device's pre-boot sequence.[4] On the successfull completion of POST, bootstrapping code is invoked.[4]

POST contains schedules as controlled by the gadget maker. These schedules are mailny used to situated an introductory worth for information and yield signals and to

Legacy BIOS	UEFI BIOS
This is the traditional BIOS	New architecture based on EFI spec
Written in assembly code; initially designed for IBM PC-AT	C based; initially designed for Itanium server systems
Interface is per-BIOS "spaghetti" code, not modular	Well defined module environment and interface based on EFI specification
Lives within the first 1MB of system memory	Can live anywhere in the 4GB system memory space
Uses 16bit memory access, requires hacks to access above 1MB memory	Allows direct access of all memory via (32-bit and/or 64 bit) pointers
Supports 3rd party modules in the form of 16 bit Option ROMS	Supports 3rd party 32/64 bit drivers
No built in boot/test environment	Built-in boot/test via EFI - Shell
Only supports 16-bit runtime services such as INT10, INT13, etc	New runtime interfaces and supports legacy OSs and 16-bit legacy devices
Example of Legacy BIOS: AMI core 8, Phoenix legacy BIOS	Standardized implementations: Aptio (AMI), H2O (Insyde), Tiano (Intel)

Figure 2.3: Legacy BIOS vs EFI BIOS
[4]

execute inner tests. These introductory conditions are additionally referred to as the gadget's state.[4] They might either be put away in firmware or included

as equipment, either as a major aspect of the configuration itself.[3]

2.3 UEFI Specification

2.3.1 Overview

This Unified Extensible Firmware Interface[1] (hereafter known as UEFI) Specification describes an interface between the operating system (OS) and the platform firmware.[5]Infact its a way for the OS and platform software to communicate only information required to support the OS boot process.UEFI comes prior to the Extensible Firmware Interface Specification (EFI)(Figures 2.4).

The interface resembles the data tables that contain information related to platform, and boot and runtime service calls which are already there with the OS loader and the OS. Both these together provide a standard environment for booting an OS.[5] This specification is designed as a pure interface specification. As such, the specification gives information about what all interfaces and structures the platform firmware must imple-

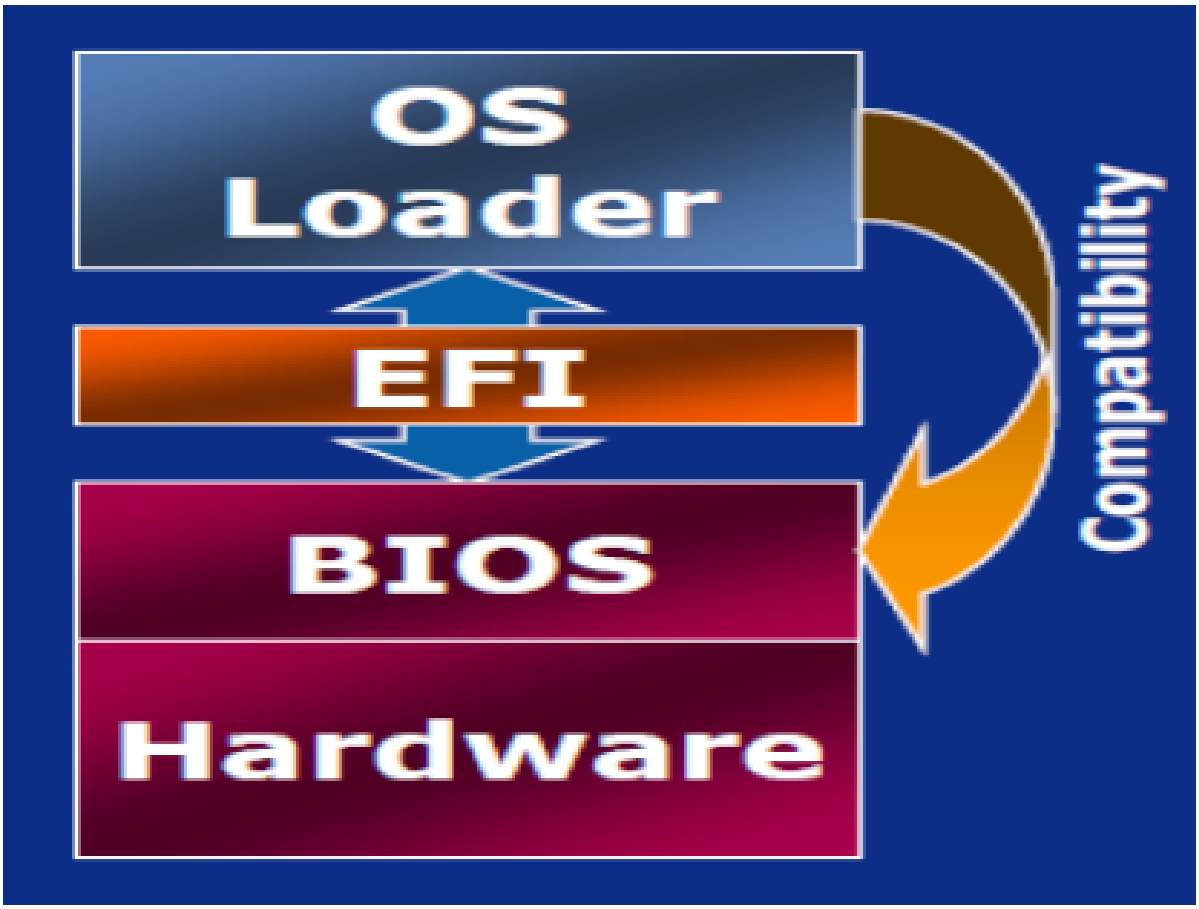

Figure 2.4: EFI interface
[1]

ment and the OS may use while booting.[1]

Complete range of hardware platforms from mobile systems to servers uses this specification. The specification is flexible enought to provide core set of services along with a selection of protocol interfaces.[5] The selection of protocol interfaces can evolve over time to be optimized for various platform market segments. At the same time, the specification allows maximum extensibility and customization abilities for OEMs to allow differentiation.[1]

2.3.2 UEFI Boot Phases

EFI BIOS is a modular code and it boots in a manner shown in (Figures 2.5). EFI Boot process is dived into four main phases which are:[1]

- Security Phase

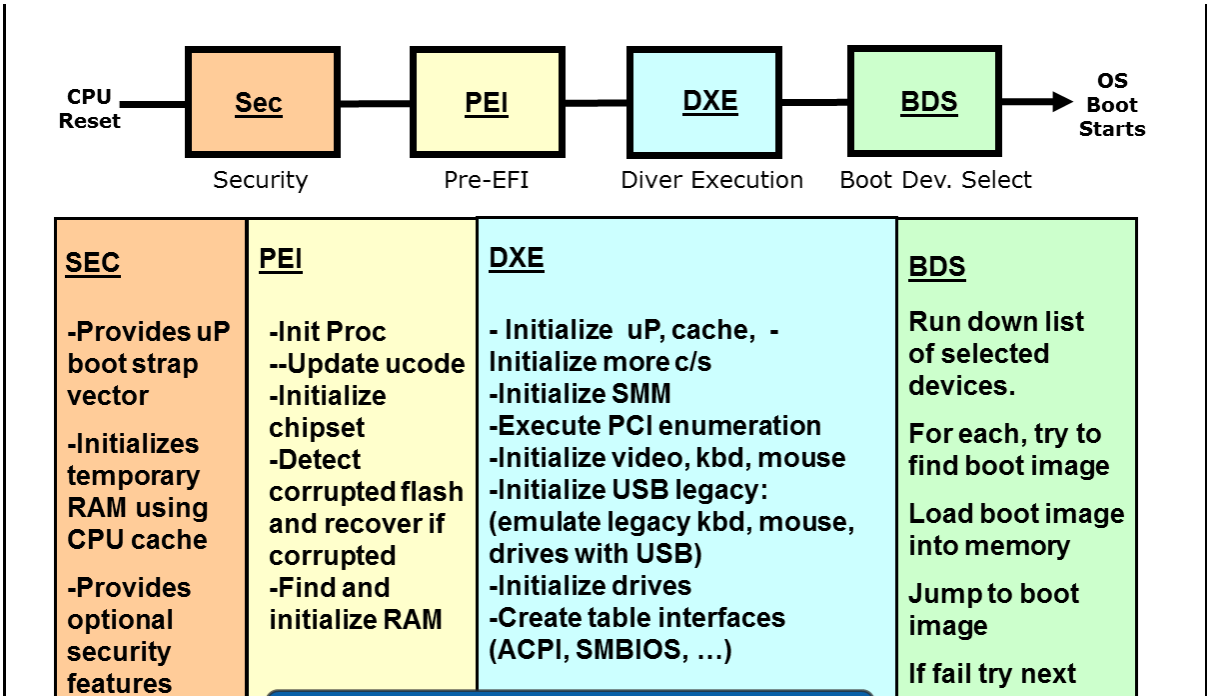

Figure 2.5: EFI boot phases

- Pre EFI Initialization Phase
- Driver Execution Environment Phase
- Boot Device Selection Phase

Each phases with its services are shown in (Figures 2.5)

Security phase:

The Security (SEC) phase is the first phase in the PI architecture and is responsible for taking care of all the event related to platform restart and it also serves as root of trust in the system.[5]

Pre-EFI Initialization Phase:

The Pre-EFI Initialization (PEI) period of the PI Architecture details (henceforth alluded to as the PI Architecture) is summoned bit ahead of schedule in the boot stream process.Once the fundamental transforming of SEC stage is done,[4] PEI stage will be conjured by any machine restart event.The PEI stage will at first work with the stage in an incipient state, utilizing just on processor assets, for example, the processor reserve

as a call stack, to dispatch Pre-EFI Initialization Modules (PEIMs).[1]

Driver execution Environment (DXE) Phase:

After PEI come the Driver Execution Environment (DXE) phase is wherein most of the system initialization is performed. Pre-EFI Initialization (PEI), the phase prior to DXE, helps in loading and execution of DXE phase by initializing perpetual memory in the platform.[2] The condition of the framework toward the end of the PEI stage is

gone to the DXE stage through a rundown of position-free information structures called

Hand-O Blocks (HOBs).[3]

Boot Device Selection (BDS) Phase:

The Boot Manager in DXE executes after all the DXE drivers whose conditions

have been slaked, have been dispatched by the DXE Dispatcher.[4] Its is the time when control is transfered to the Boot Device Selection (BDS) period of execution. The stage boot approach is actualized in this phase.This boot arrangement gives

exibility that authorizes framework sellers to tweak the utilizer experience amid this stage

of execution.[1]

If the BDS phase cannot make any further progress, it will reinvoke the DXE Dispatcher to check if the dependencies of any additional DXE drivers have been satisfied since the last time the DXE Dispatcher was invoked.[5]

Chapter 3

Implementation Details

3.1 BDF Introduction

The BIOS Directive Format (BDF) is an attempt to provide a (simple) machine parsable format to define necessary BIOS configuration flows. [4] The BIOS Directive Format is an XML schema. The hierarchy of a BDF file consists of a top-level ***bdf*** node, with flow ***child*** nodes, followed by **registerFile, register,** and **field** nodes as shown in (Figures 3.1).

3.1.1 BDF attributes : Details

- ***bdf*** ...
  - ***version*** : which may be specified if there are future versions of the BIOS Directive Format that introduce schema variations.
- ***flow*** ...
  - ***name*** : which specifies the name of the flow.

```
<?xml version= "1.0" encoding= "ISO-8859-1" ?>
- <bdf version= "1.0">
- <flow name= "" owner= "" version= "" ordered= "" before= "" after= "">
-    <registerFile name= "" flags= "" else= "" comment= "" stepping= "">
-       <register name= "" value= "" comment= "">
            <field name= "" value= "" pollfor= "" timeout= "" />
        </register>
    </registerFile>
  </flow>
  </bdf>
```

Figure 3.1: BDF format

- ***owner*** : which specifies the owner/contact of the flow.
- ***version*** : which specifies the version/release/revision of the flow. This is especially useful when handling revision control of updated/changed flows.
- ***before and after*** : which are used to optionally specify the events that temporally bound when the flow should be executed.
- ***ordered*** : which specifies whether the register child accesses must be performed in-order.

- ***registerFile*** ...
  - ***name*** : which specifies the path of the registerFile.
  - ***flags*** : which optionally specifies (as semi-colon separated "key=value" pairs) the parameters that control the applicability of the registerFile programming.
  - ***else*** : which helps managing both if-else conditions.
  - ***comment*** : which optionally specifies additional descriptive language the user would like to provide.
  - ***stepping*** : which specifies which stepping of the silicon chip we are refering to.

- ***register*** ...
  - ***name*** : which specifies the name of the register.
  - ***value*** : which specifies value to be written into that register.
  - ***comment*** : which optionally specifies additional descriptive language the user would like to provide.

- ***field*** ...
  - ***name*** : which specifies name of the field.
  - ***value*** : which specifies value to be written into the field.
  - ***pollfor*** : which optionally specifies that the field should be polled until the specified value is returned. This is especially useful if there is a section of a flow that requires hardware semaphore before continuing with programmings.

– ***timeout*** : which optionally specifies a timeout to use when using the pollfor attribute

3.2 Script for generating Header files

This is the Python script which is parsing SOC-RTL.xml and my BDF.xml and generating all the required header files.

```
import xml.etree.ElementTree as etree import time
StartTime = time.time() print time.ctime() FileList = [ regFile1, regFile2, regFile3,....regFilen]
File1 = [0 for i in FileList]

print 'parsing xml file...'
stim=time.time()
root = etree.parse('rtl.xml').getroot()
etim=time.time()
print 'time elapsed: %d secs' % (etim-stim)

print 'generating header files...'
for i in range(len(FileList)):

stim=time.time()
regfilelist = []
for keyname in FileList[i][1]:
regfile.findtext('name').split('/')[0] == keyname]
regfilelist.extend(tempList)

OffsetList = []
for registerFile in regfilelist:
for register in registerFile.findall('register'):
string = int(register.findtext('addressOffset').split('h')[1], 16)
OffsetList.append(string)

OffsetSet = list(set(OffsetList))
```

```
OffsetSet.sort()

    HeaderFileName = FileList[i][0]

    File1[i] = open(HeaderFileName + ".h", "w+")
File1[i].write(" This file was automatically generated. Modify at your own risk.")
File1[i].write("@copyright")
File1[i].write(" Copyright (c) 2010 - 2013 Intel Corporation. All rights reserved.")
File1[i].write(" This software and associated documentation (if any) is furnished")
File1[i].write(" under a license and may only be used or copied in accordance")
File1[i].write(" with the terms of the license. Except as permitted by such")
File1[i].write(" license, no part of this software or documentation may be")
File1[i].write(" reproduced, stored in a retrieval system, or transmitted in any")
File1[i].write(" form or by any means without the express written consent of")
File1[i].write(" Intel Corporation.")
File1[i].write(" This file contains an 'Intel Peripheral Driver' and uniquely")
File1[i].write(" identified as 'Intel Reference Module' and is")
File1[i].write(" licensed for Intel CPUs and chipsets under the terms of your")
File1[i].write(" license agreement with Intel or your vendor. This file may")
File1[i].write(" be modified by the user, subject to additional terms of the")
File1[i].write(" license agreement.")
File1[i].write("ifndef " + HeaderFileName + "h")
File1[i].write("define " + HeaderFileName + "h")
File1[i].write("pragma pack(push, 1)")
File1[i].write("include "MrcTypes.h")

    for offset in OffsetSet:
flag = False

    for registerFile in regfilelist:
for register in registerFile.findall('register'):
if offset == int(register.findtext('addressOffset').split('h')[1], 16):
```

```
flag = True
regName = register.findtext('designName')
regSize = int (register.findtext('size'))
regStart = int(register.findtext('addressOffset').split('h')[1], 16)

    NextBitFieldLsb = 0
if regSize == 8:
for field in register.findall('field'):
bitfieldName = field.findtext('name')
Width = int(field.findtext('bitWidth'))
bitfieldLsb = int(field.findtext('bitOffset'))
bitfieldMsb = bitfieldLsb + Width - 1

    if NextBitFieldLsb == 0:
File1[i].write("typedef union  struct ")

    if bitfieldLsb ¡¿ NextBitFieldLsb:
rsvd = bitfieldLsb - NextBitFieldLsb
bitFieldMinus = bitfieldLsb - 1
File1[i].write(" U8 :% 2u; // Bits%u:%u "% (rsvd, bitFieldMinus, NextBitFieldLsb))

    File1[i].write(" U8 % -40s:% 2u; // Bits%u:%u "% (bitfieldName, Width, bitfieldMsb,
bitfieldLsb))
NextBitFieldLsb = bitfieldMsb + 1

    rsvd = 7 - bitfieldMsb
if bitfieldMsb ¡ 7:
File1[i].write(" U8 :% 2u; // Bits 7:%u "% (rsvd, NextBitFieldLsb))
File1[i].write("  Bits;")
File1[i].write(" U8 Data;")
File1[i].write("%sSTRUCT;"% regName)
```

```
    elif regSize == 16:
for field in register.findall('field'):
bitfieldName = field.findtext('name')
Width = int(field.findtext('bitWidth'))
bitfieldLsb = int(field.findtext('bitOffset'))
bitfieldMsb = bitfieldLsb + Width - 1

    if NextBitFieldLsb == 0:
File1[i].write("typedef union  struct ")

    if bitfieldLsb ¡¿ NextBitFieldLsb:
rsvd = bitfieldLsb - NextBitFieldLsb
bitFieldMinus = bitfieldLsb - 1
File1[i].write(" U16 :% 2u; // Bits%u:"% (rsvd, bitFieldMinus, NextBitFieldLsb))

    File1[i].write(" U16% -40s:% 2u; // Bits%u:"% (bitfieldName, Width, bitfieldMsb,
bitfieldLsb))
NextBitFieldLsb = bitfieldMsb + 1

    rsvd = 15 - bitfieldMsb
if bitfieldMsb ¡ 15:
File1[i].write(" U16 :% 2u; // Bits 15%u"% (rsvd, NextBitFieldLsb))
File1[i].write("  Bits;")
File1[i].write(" U16 Data;")
File1[i].write(" U8 Data8[2];")
File1[i].write("%sSTRUCT;"% regName)

    elif regSize == 32:
for field in register.findall('field'):
bitfieldName = field.findtext('name')
Width = int(field.findtext('bitWidth'))
bitfieldLsb = int(field.findtext('bitOffset'))
```

```
bitfieldMsb = bitfieldLsb + Width - 1

    if NextBitFieldLsb == 0:
File1[i].write("typedef union  struct ")

    if bitfieldLsb ¡¿ NextBitFieldLsb:
rsvd = bitfieldLsb - NextBitFieldLsb
bitFieldMinus = bitfieldLsb - 1
File1[i].write(" U32 :% 2u; // Bits%u%u%(rsvd, bitFieldMinus, NextBitFieldLsb))

    File1[i].write(" U32% -40s:% 2u; // Bits%u:%u"% (bitfieldName, Width, bitfieldMsb,
bitfieldLsb))
NextBitFieldLsb = bitfieldMsb + 1


    rsvd = 31 - bitfieldMsb
if bitfieldMsb ¡ 31:
File1[i].write(" U32 :% 2u; // Bits 31:%u"% (rsvd, NextBitFieldLsb))
File1[i].write("  Bits; ")
File1[i].write(" U32 Data; ")
File1[i].write(" U16 Data16[2]; ")
File1[i].write(" U8 Data8[4]; ")
File1[i].write("%sSTRUCT; "% regName)

    elif regSize == 64:
for field in register.findall('field'):
bitfieldName = field.findtext('name')
Width = int(field.findtext('bitWidth'))
bitfieldLsb = int(field.findtext('bitOffset'))
bitfieldMsb = bitfieldLsb + Width - 1

    if NextBitFieldLsb == 0:
```

```
File1[i].write("typedef union  struct  ")

    if bitfieldLsb ¡¿ NextBitFieldLsb:
rsvd = bitfieldLsb - NextBitFieldLsb
bitFieldMinus = bitfieldLsb - 1
File1[i].write(" U64 :% 2u; // Bits%u:%u "% (rsvd, bitFieldMinus, NextBitFieldLsb))

    File1[i].write(" U64% -40s:% 2u; // Bits%u:%u "% (bitfieldName, Width, bitfieldMsb,
bitfieldLsb))
NextBitFieldLsb = bitfieldMsb + 1

    rsvd = 63 - bitfieldMsb
if bitfieldMsb ¡ 63:
File1[i].write(" U64 :% 2u; // Bits 31:%u "% (rsvd, NextBitFieldLsb))
File1[i].write("  Bits; ")
File1[i].write(" U64 Data; ")
File1[i].write(" U32 Data32[2]; ")
File1[i].write(" U16 Data16[4]; ")
File1[i].write(" U8 Data8[8]; ")
File1[i].write("%s STRUCT; "% regName)

    else:
File1[i].write("//WARNING: REGISTER DOES NOT HAVE A SUPPORTED BIT WIDTH,
Name:%s, Width:% 2u "% (regName, regSize))

    Name = register.findtext('name')
File1[i].write("define% -40s (0x%X) "% (Name + "REG", regStart))
if (regSize == 8) or (regSize == 16) or (regSize == 32) or (regSize == 64):
for field in register.findall('field'):

bitfieldName = field.findtext('name')
Width = int(field.findtext('bitWidth'))
```

bitfieldLsb = int(field.findtext('bitOffset'))

bitfieldMsb = bitfieldLsb + Width - 1

MaxValue = (1 ¡¡ Width) - 1

Mask = MaxValue ¡¡ bitfieldLsb

fieldName = Name + "-" + bitfieldName + "OFF"

File1[i].write(" define% -40s(%u) "% (fieldName, bitfieldLsb))

fieldName = Name + "-" + bitfieldName + " WID"

File1[i].write(" define% -40s(%u) "% (fieldName, Width))

fieldName = Name + "-" + bitfieldName + " MSK"

File1[i].write(" define% -40s (0x%X) "% (fieldName, Mask))

fieldName = Name + "-" + bitfieldName + " MAX"

File1[i].write(" define% -40s (0x%X)"% (fieldName, MaxValue))

defaultValueName = [reset for reset in field.findall('reset') if reset.get('type') == 'default'][0].text

defaultValueName = defaultValueName.replace('-', '')

defType = defaultValueName.find("'b")

if (defType ¿= 0):

defaultValue = int(defaultValueName.split('b')[1], 2)

else:

defType = defaultValueName.find("'h")

if (defType ¿= 0):

defaultValue = int(defaultValueName.split('h')[1], 16)

if (defType ¿=0):

fieldName = Name + "-" + bitfieldName + " DEF"

```
File1[i].write(" define% -40s (0x%X)"% (fieldName, defaultValue))

    if flag : break
breaks the regfile loop
    File1[i].write(" pragma pack(pop) ")
File1[i].write(" endif // - + HeaderFileName + " h- ")

    File1[i].close()
etim=time.time()
print%d file done, time taken%d sec'% (i, etim-stim)

    endTime=time.time()
print 'Done Time elspsed:%d seconds'% (endTime-StartTime)
```

3.2.1 Explanation of Script

The above script basically parse the Silicon RTL and generate all the required header files automatically based on the information given in BDF.xml. Along with the generation of header files this script also standarize the way all register offsets, register mask, register width, etc. are declared throughout the BIOS code which is useful in further optimization of code.

3.2.2 Output: Sample Header File

(Figures 3.3),(Figures 3.4),(Figures 3.5),(Figures 3.6),(Figures 3.7),(Figures 3.8)shows view of a header file generated from the above script. The images are intentionally blurred as they contain Intel confidential stuff.

3.3 Source Code Generation

A python script parses silicon RTL.xml, BDF.xml and takes necessary header files as input and generate source code(.c) files automatically for various modules in a proper format.

It was observed that the manual writing of source code (for say 10 .c files) takes

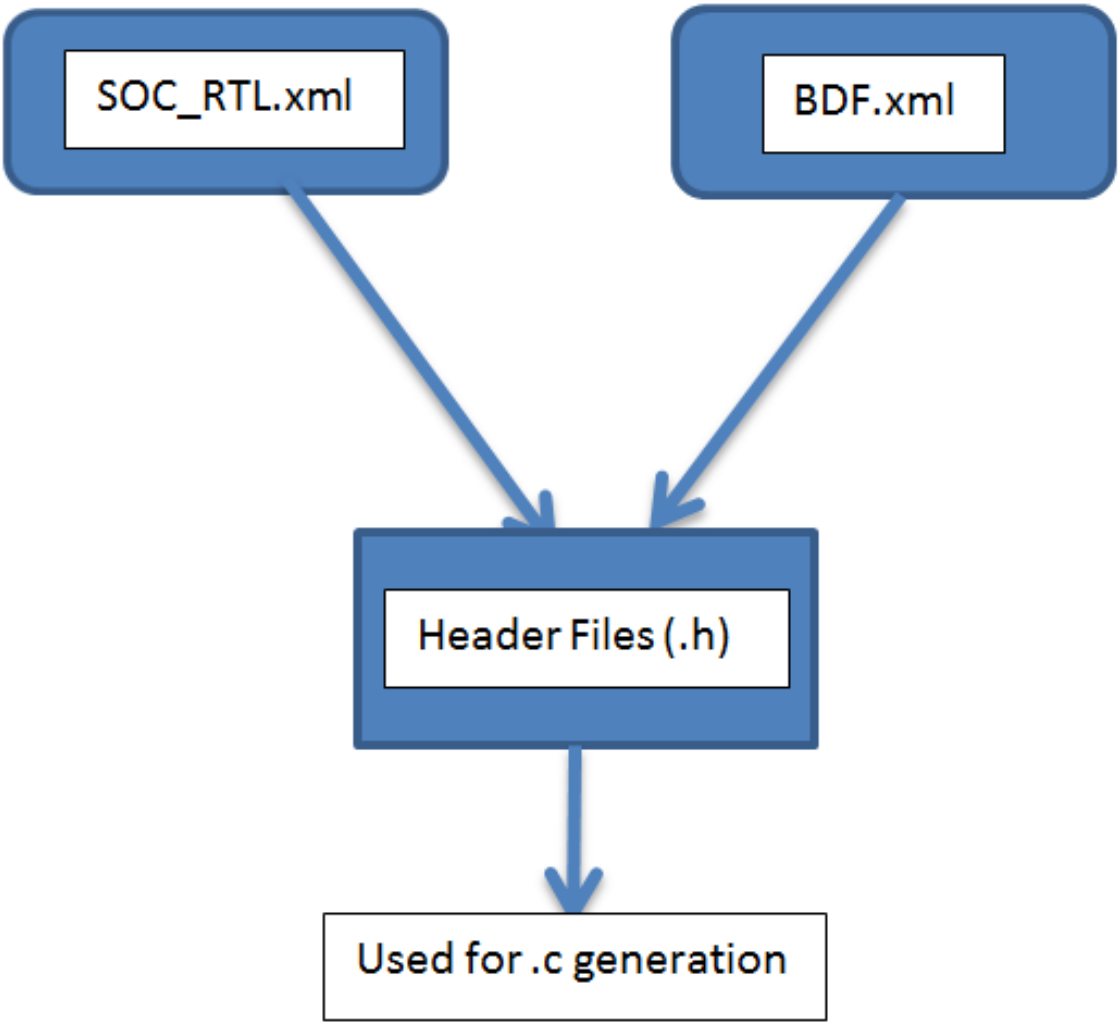

Figure 3.2: Header file generation

Figure 3.3: Sample header file

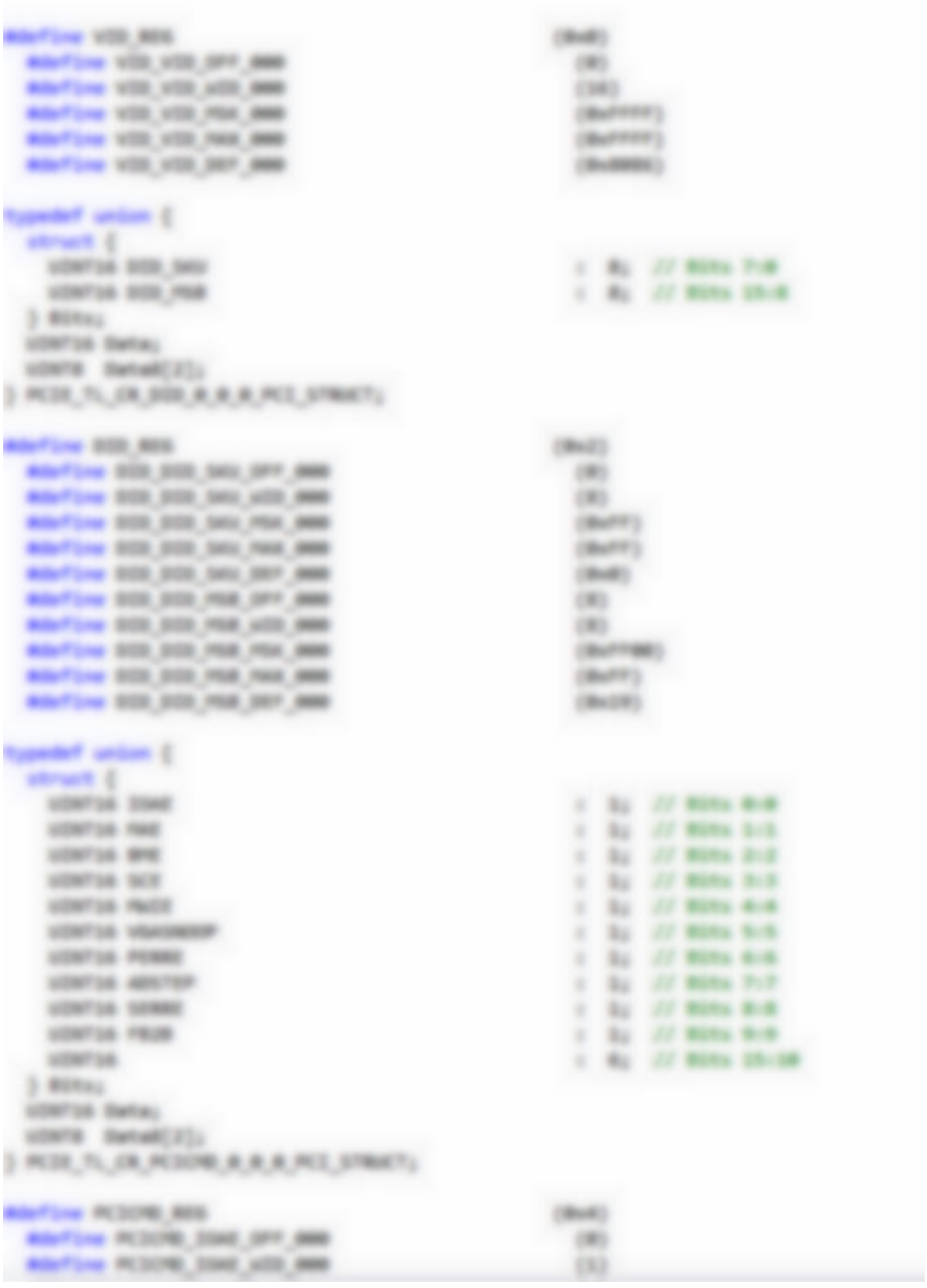

Figure 3.4: Sample header file

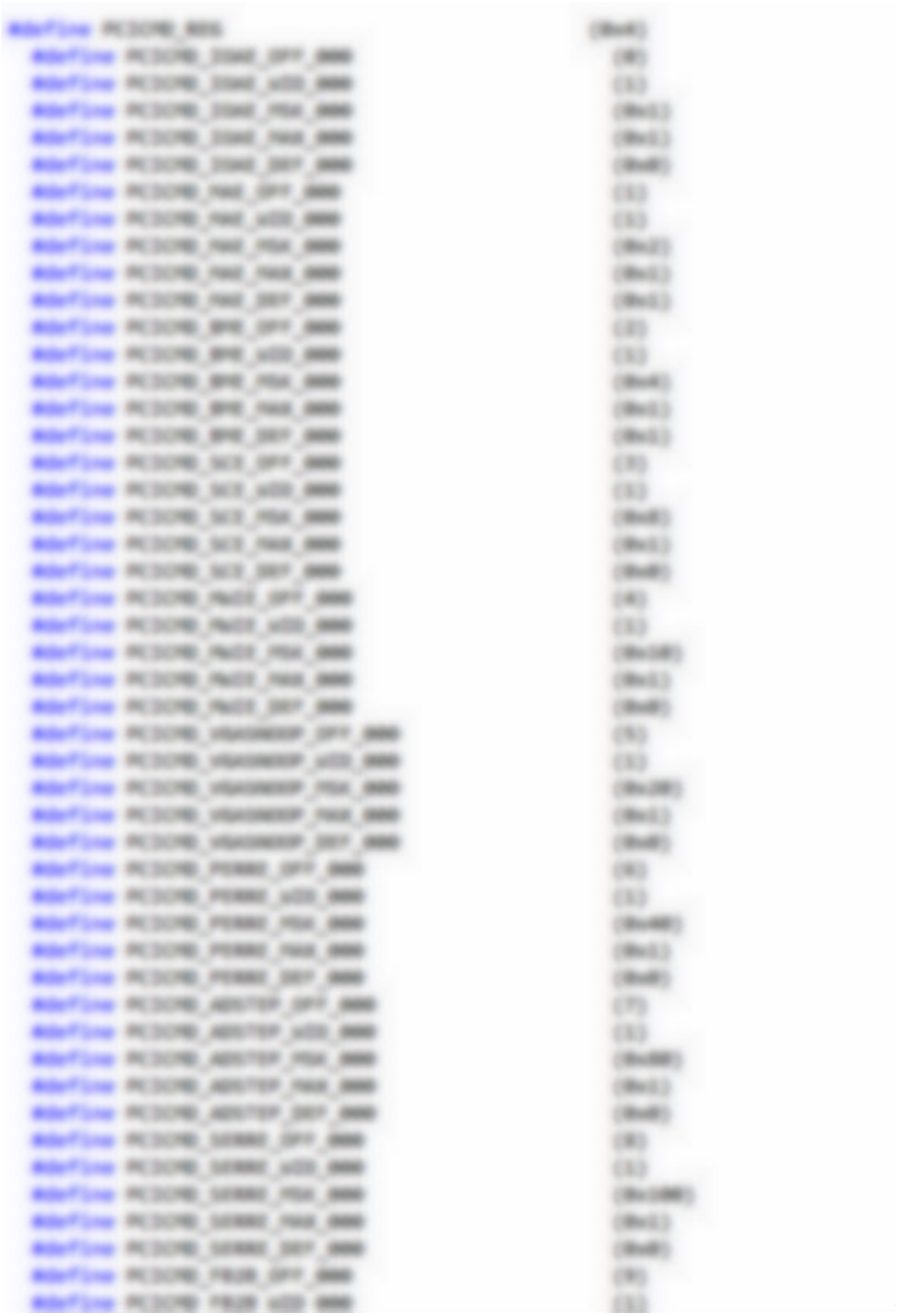

Figure 3.5: Sample header file

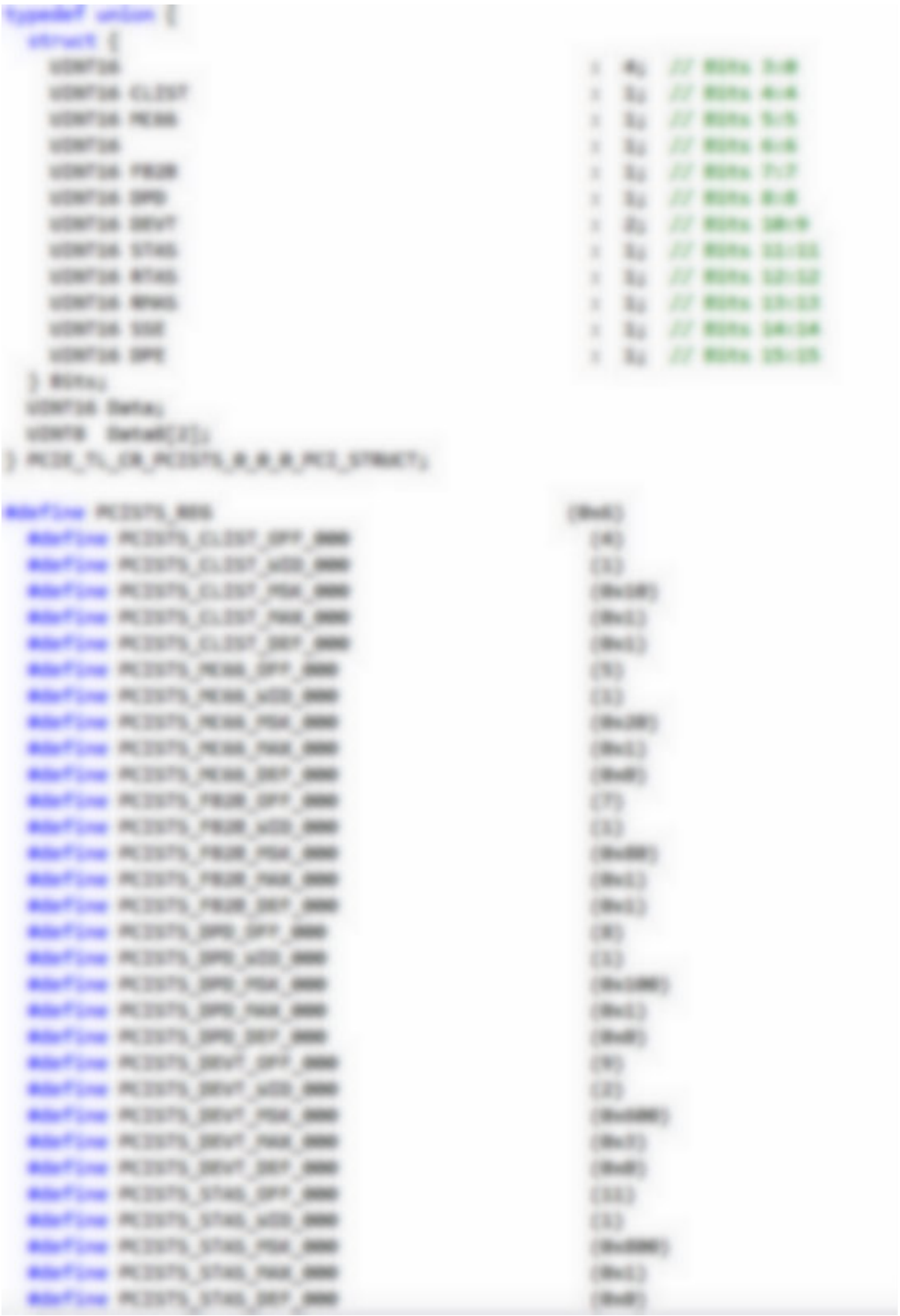

Figure 3.6: Sample header file

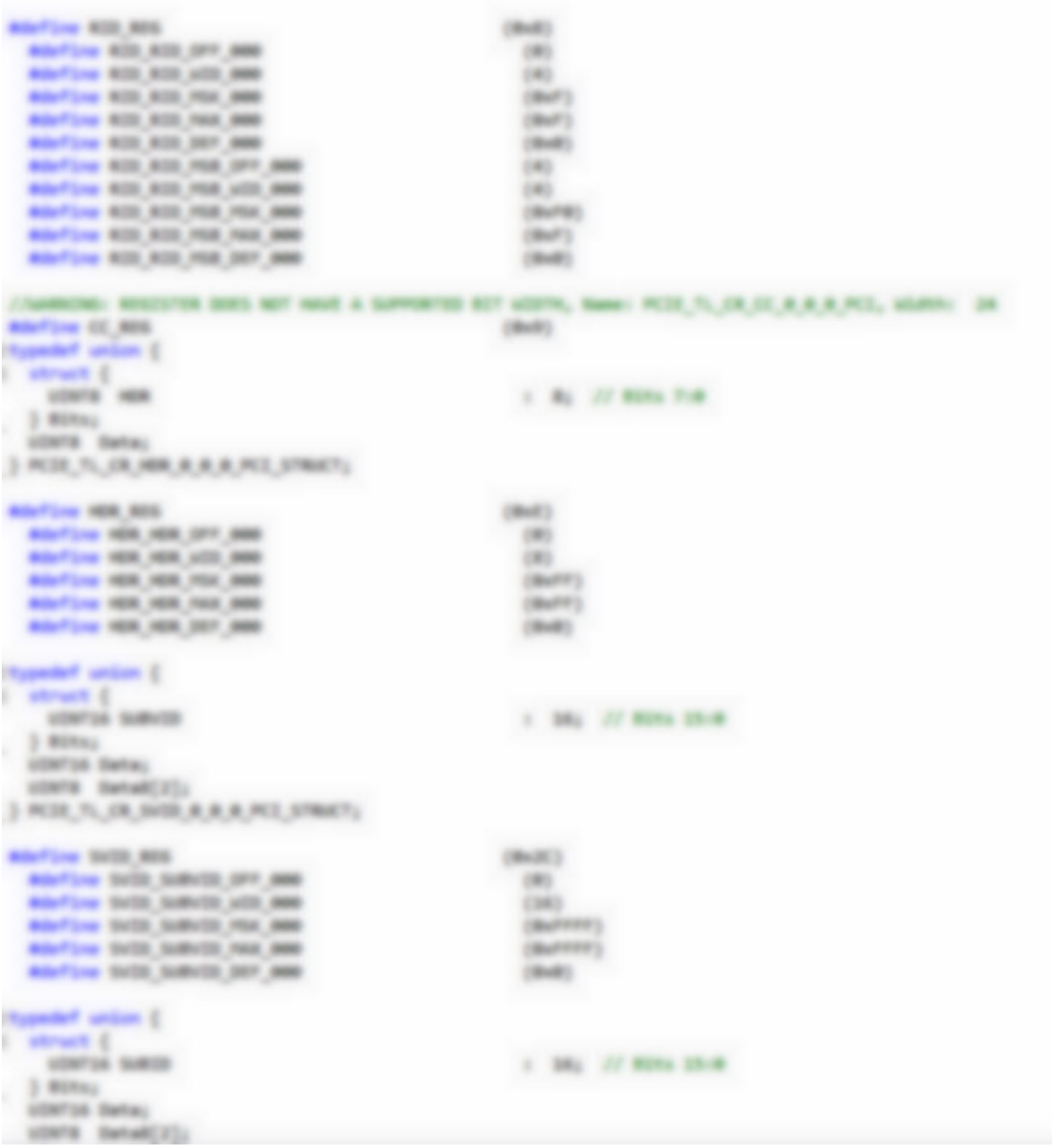

Figure 3.7: Sample header file

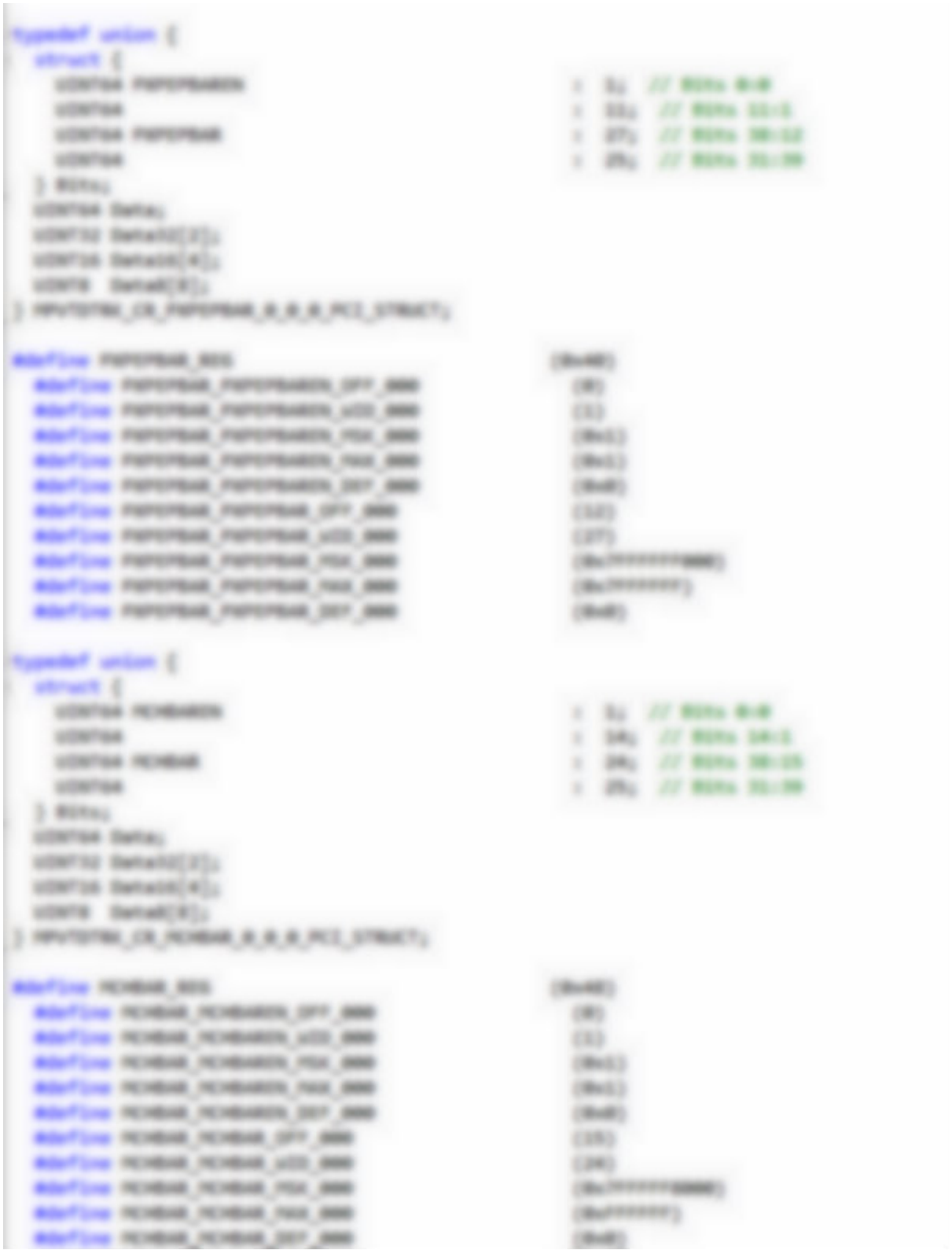

Figure 3.8: Sample header file

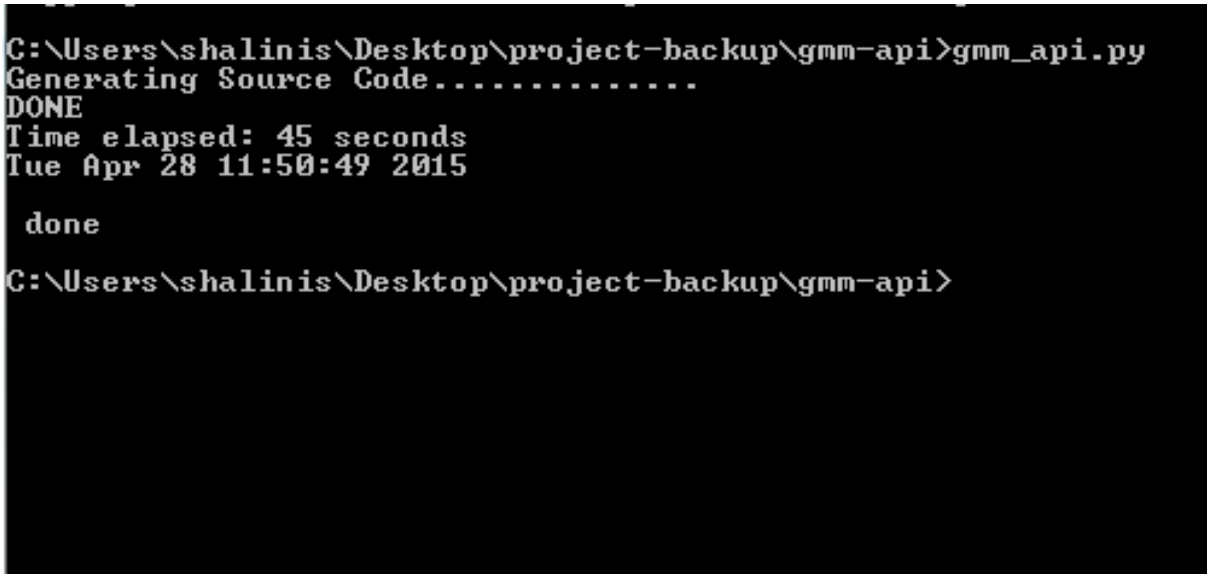

Figure 3.9: Source code generation

approximately a month or even more depending on the time taken for collecting all the requirements.As we have hundreds of such .c files

But with this automation its hardly takes a minute or even less. Here is a screen-shot (Figures 3.9) which shows time taken to generate a source code file using python script.

3.3.1 Scrpit for Source Code generation

Following is the python script which parsed silicon RTL.xml and BDF.xml and generates source code(.c) file as output.

import xml.etree.ElementTree as etree

import re

import sys

import time

StartTime = time.time()

count-step = 1

var = count= 0

count-var = 0

var-1 = 0

root = etree.parse('cregs-mod.xml').getroot()

step1 =[]

headerfile =[]

```
a =[]
no-of-flags = []
check = []
check1 = []
b = []
c = []
e = 0

    File = open("Gmm-API.c","w+")

    ### Function to check all the conditions based on information provided in BDF
before updating any register value ###
    ### From Source code point of view it takes care of all the if-else conditions ###
    def func-flag(str1,str2):
global count-var
flag-count = str1.count(';')
flag = str1
if flag-count== 0:

    flag1 = flag.split('-')[0]

    flag2 = flag.split('-')[1]

    flag3 = flag.split('-')[2]
flag4 = flag3.split('=')[0]
flag5 = flag3.split('=')[1]
comment = str2

    for regfile in root.findall('registerFile'):
name1 = regfile.findtext('name')
if regfile.findtext('name') != flag1:
continue
```

else:

for reg in regfile.findall('register'):
if reg.findtext('name')!=flag2:
continue
else:

for field in reg.findall('field'):
if field.findtext('name')!= flag4:
continue
else:
RegToRead = flag2
Width = int(field.findtext('bitWidth'))
bitfieldLsb = int(field.findtext('bitOffset'))
bitfieldMsb = bitfieldLsb + Width - 1
MaxValue = (1 ¡¡ Width) - 1
Mask = MaxValue ¡¡ bitfieldLsb
Bus = str(regfile.findtext('bus'))
Device = str(regfile.findtext('device'))
Function = str(regfile.findtext('function'))
fieldName = flag2 + "-" + flag4 + "-MSK" + "-" + Bus + Device + Function
Bus = regfile.findtext('bus')
Device = regfile.findtext('device')
Function = regfile.findtext('function')
BaseAddressToRead = 'McD' + Device + 'BaseAddress'
File.write(" Data32-%s = MmioRead32 ("+ BaseAddressToRead + " + " + RegToRead
+ "-REG); " % count-var)

File.write(" if (%s & Data32-%s) "% (fieldName,count-var))
as mask name should be used for condition checking. File.write(" ")

File.write(" DEBUG ((EFI-D-INFO,¨%10s¨)); "%comment)

```
    count-var = count-var+1

File.write("   ")

    else:

    ### if there are more than one condition check
    for i in range(0,flag-count+1):
n=flag.split(';')[i]
no-of-flags.append(n)
for i in range(0,len(no-of-flags)):
flag = no-of-flags[i]
flag1 = flag.split('-')[0]

    flag2 = flag.split('-')[1]

    flag3 = flag.split('-')[2]
flag4 = flag3.split('=')[0]
flag5 = flag3.split('=')[1]
comment = str2

    for regfile in root.findall('registerFile'):
name1 = regfile.findtext('name')
if regfile.findtext('name') != flag1:
continue
else:

    for reg in regfile.findall('register'):
if reg.findtext('name')!=flag2:
continue
else:
```

```
for field in reg.findall('field'):
if field.findtext('name')!= flag4:
continue
else:
RegToRead = flag2
Width = int(field.findtext('bitWidth'))
bitfieldLsb = int(field.findtext('bitOffset'))
bitfieldMsb = bitfieldLsb + Width - 1
MaxValue = (1 ¡¡ Width) - 1
Mask = MaxValue ¡¡ bitfieldLsb
Bus = str(regfile.findtext('bus'))
Device = str(regfile.findtext('device'))
Function = str(regfile.findtext('function'))
fieldName = flag2 + "-" + flag4 + "-MSK" + "-" + Bus + Device + Function
Bus = regfile.findtext('bus')
Device = regfile.findtext('device')
Function = regfile.findtext('function')
BaseAddressToRead = 'McD' + Device + 'BaseAddress'
File.write(" Data32-"+str(count-var)+"= MmioRead32 ("+ BaseAddressToRead + " +
" + RegToRead + "-REG); ")
File.write(" if (%s & Data32-%s) "%(fieldName,count-var))
File.write("  ")
count-var = count-var+1

    File.write(" DEBUG ((EFI-D-INFO,%10s)); "%comment)

    for x in range(1,len(no-of-flags)):
File.write("  ")

    File.write(" } ")

    return
```

```
    ### Function which writes under CPUID condition ###
    def func-policy(str1,str2,str3,value):

    for regfile1 in root.findall('registerFile'):
name1 = regfile1.findtext('name')
if regfile1.findtext('name') != str1:
continue
else:

    for reg1 in regfile1.findall('register'):

    if reg1.findtext('name')!=str2:
continue
else:
for field1 in reg1.findall('field'):
if field1.findtext('name')!= str3:
continue
else:
size = reg1.findtext('size')
RegToRead = str2
Width = int(field1.findtext('bitWidth'))
bitfieldLsb = int(field1.findtext('bitOffset'))
bitfieldMsb = bitfieldLsb + Width - 1
MaxValue = (1 ¡¡ Width) - 1
Mask = MaxValue ¡¡ bitfieldLsb
Bus = str(regfile1.findtext('bus'))
Device = str(regfile1.findtext('device'))
Function = str(regfile1.findtext('function'))
maskName = str2 + "-" + str3 + "-MSK" + "-" + Bus + Device + Function
word = 'MCHBAR'
check = [e for e in word if e in str1.split('-')]
```

```
if check==[]:
BaseAddressToRead = 'McD' + Device + 'BaseAddress'
else:
BaseAddressToRead = 'MchBarBase'
if bitfieldLsb==0:
File.write(" MmioWrite%s( %s + %s ,0x%s) ; "%(size,BaseAddressToRead,str2,value))
else:
File.write(" MmioWrite%s( %s + %s ,MmioAnd%s((UINTN) %s ,(UINT%s)(0x%s ¡¡
%s)) ; "%(size,BaseAddressToRead,str2,size,maskName,size,value,bitfieldLsb))

    return
### MAIN FUNCTION ###
    ### counting the number of flow steps ### File.write("/** @file ")
File.write(" This file was automatically generated. Modify at your own risk. ")
print "Generating Source Code....."

    for regfile in root.findall('registerFile'):
name = regfile.findtext('name')
for reg in regfile.findall('register'):

    for field in reg.findall('field'):

    for flow in field.findall('flow'):
if flow!="":
s = flow.get('step')
step1.append(s)
var = var+1
hname = name.split('/')[0]
### identifying all the required header files which are also autogenerated header-
file.append(hname)

    ### list containing all the header files to be included in C codea=list(set(headerfile))
```

```
    for i in range(0,len(a)):
File.write((”%include ¡%s.h¿ ” % a[i]))

    platform = ’Skylake’ : ’EnumSkl’

    File.write(”VOID ”)
File.write(”gmminitapi ( ”)
File.write(”IN CPU-STEPPING CpusteppingId”)
File.write(” )  ”)

    for regfile in root.findall(’registerFile’):
name = regfile.findtext(’name’)
for reg in regfile.findall(’register’):
regname = reg.findtext(’name’)
for field in reg.findall(’field’):
fname = field.findtext(’name’)
for flow in field.findall(’flow’):
if flow!=””:
Bus = regfile.findtext(’bus’)
###here, BDF are strings and not integers Device = regfile.findtext(’device’)

    Function = regfile.findtext(’function’)
word = ’MCHBAR’ check1 = [e for e in word if e in name.split(’-’)]
BaseAddressToRead = ’McD’ + Device + ’BaseAddress’
b.append(BaseAddressToRead)

    flag = flow.findtext(’flag’)
flag-count = flag.count(’;’)
e = flag-count
if flag-count !=0:
for i in range(0,flag-count+1):
```

```
var-1=var-1+1
else:
var-1=var-1+1

    File.write(" %-25s MchBarBase; " % ('UINTN'))
c = list(set(b))
for i in range(0,len(c)):
File.write(" %-25s %s; " % ('UINTN',c[i]))
if check1 !=[]:
File.write(" %-25s MchBarBase; " % ('UINTN'))

    for i in range(0,2):
File.write(" %-25s Data32-%s; " %('UINT32',i))

    for address in c:
for key,value in baseadd1.iteritems():

    if address==key:
File.write(" %s = %s ;" %(key,value))

    for key,value in baseadd2.iteritems():
if address==key:
File.write(" %s = %s ;" %(key,value))

    for i in range(1,len(step1)):

    for regfile in root.findall('registerFile'):
name = regfile.findtext('name')
for reg in regfile.findall('register'):
regname = reg.findtext('name')
for field in reg.findall('field'):
fname = field.findtext('name')
```

```
for flow in field.findall('flow'):
if ((flow!="") and (int(flow.get('step'))==i)):

    stepping = flow.findtext('stepping')
comment = flow.findtext('comment')
value = flow.findtext('value')
val = value.split('b')[1]

    if stepping!="":
File.write("// For SKL Stepping %s "%(stepping))
File.write(" if ( CpuSteppingId == EnumSkl%s)  " %(stepping))

    condition = flow.findtext('flag')
if condition != "":
### checking no of conditions in flag func-flag(condition,comment)

    func-policy(name,regname,fname,val)

    if stepping !="":
File.write("  ")

    File.write(" return; ")
File.write("  ")
EndTime = time.time() - StartTime
print "DONE"
print 'Time elapsed: %d seconds' % EndTime
print time.ctime()
File.close()
print " done"
```

Figure 3.10: Sample source code file

3.3.2 Output : Sample Source code file

Figures (Figures 3.10)(Figures 3.11) show view of source code file generated from the above script. The images are intentionally blurred as they contain Intel confidential stuff. A a part of generating POC for this research based project, one of the source code(.c) file was integrated into the existing BIOS code base and it was observed that BIOS code was building successfully.

Figure 3.11: Sample source code file

Chapter 4

Research Scope

Now from research point of view I would like to highlight the point that by this project I am trying to generate a POC (Proof Of Concept). Hence research is involved in demonstrating the feasibility if this idea which includes-

- Understanding the minutes of processor family viz. PCI Express configuration, System Agent devices, Microcode update, ACPI, System Management Mode, etc.
- Understanding the BIOS code and analyzing what all components/devices of processor can be automated by writing script in generalized way.
- Once done with the script for any one module of any processor, I have to be sure if it is generalized enough to generate code for other generations of Silicon.

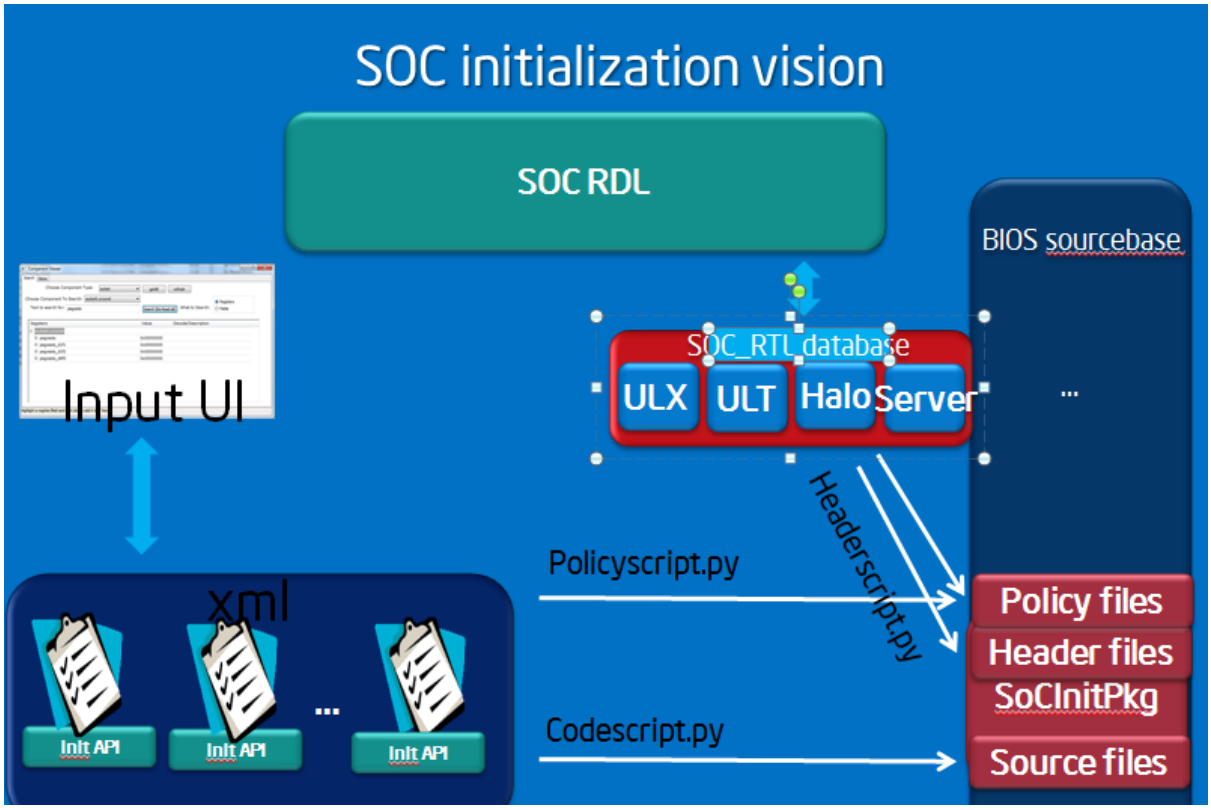

Figure 4.1: Research scope

Chapter 5

Requirements

5.1 Data Requirements

Two major data requirements for my project would be-:

- BIOS specs. - to decide the flow for my code generation. As all the programming detalis are mentioned in BIOS specs. only, so its the kind of entry point for doing my research work.
- RTL.xml - it is the file wherein all the register details are there and i would be parsing this file only based on the information present in BIOS spec. to generate the code.

5.2 Technical Requirements

I would be parsing all the '.xml' and '.xlsx' files using Python. Code is based on C language and developed using Microsoft Visual Studio as an IDE along with Python.

Chapter 6

Conclusion

This thesis shows that the main objective of this project is-

- Dehumanize silicon initialization code.
- Develop a saclable solution that can shift left software rediness to improve efficiency.

Also this project will imporve developement efficiency with respect to-

- Reduced man hours on developing Si initialization code.
- Reduced number of issues due to incorrect or incomplete requirements.
- Accelerate SW readiness in pre-Si timeframe.
- Leverage early RDL availability for early code readiness.

References

[1] "Boot phases," *Unified Extensible Firmware Interface Specication*, vol. 2.3.1, pp. 993–1022, 2011.

[2] http://saba.intel.com/Saba/Web/Main//Platform Basics for CQR, "Platform basics."

[3] Intel, "Basic architecture," *Intel64 and IA-32 Architectures Software Developer's Manual*, vol. 1-3, 2013.

[4] http://saba.intel.com/Saba/Web/Main//Bios Basics for CQR, "Bios."

[5] www.uefi.org, "Uefi."