OPEN SOURCE FIRMWARE

FOUNDATION

ByteTalks

OPEN SOURCE FIRMWARE FOUNDATION

ByteTalks

2024
MAR. 7TH

FSP Customization

Remove “non-mandatory” components from the Intel FSP

Subrata Banik, Firmware Engineer, Google
Vincent Zimmer, Firmware Engineer, Intel

Speaker Introduction

Subrata Banik is a Firmware Engineer with over 15 years of experience in system firmware design, development, and debugging across various firmware architectures, including UEFI, coreboot for x86 and ARM platforms. Subrata has received multiple US Patents and is the author of two books on firmware.
https://www.linkedin.com/in/subrata-banik/

Vincent Zimmer has been working on embedded firmware for over 32 years. Vincent has contributed to or created firmware spanning various initiatives and standards. Vincent has also co-authored various papers and books, along US and international patents.
https://www.linkedin.com/in/vzimmer/.

Objective

Designing custom silicon for each product is an attractive objective, but it is not a practical solution. Our aim is to be able to **customize the silicon reference code to meet the requirements of the target product**.

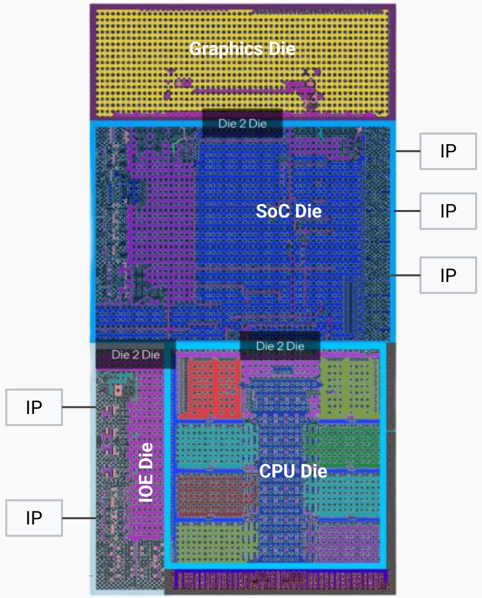
Materials

- Overview of FSP
- FSP <-> Boot Firmware
- Pros and Cons of FSP+Boot Firmware Model
- Stepwise approach to get rid of “non-mandatory” FSP modules
- Demonstration

Firmware Support Package (FSP) Overview

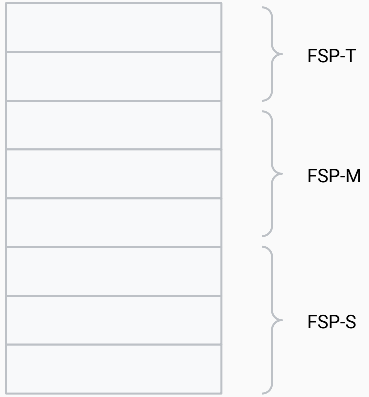
- Silicon IP provides great sets of feature for different platform use

Intel Silicon



- Silicon Reference Code (FSP) represents SoC features.
- Standard API Interface to communicate with boot firmware with defined output data structure.

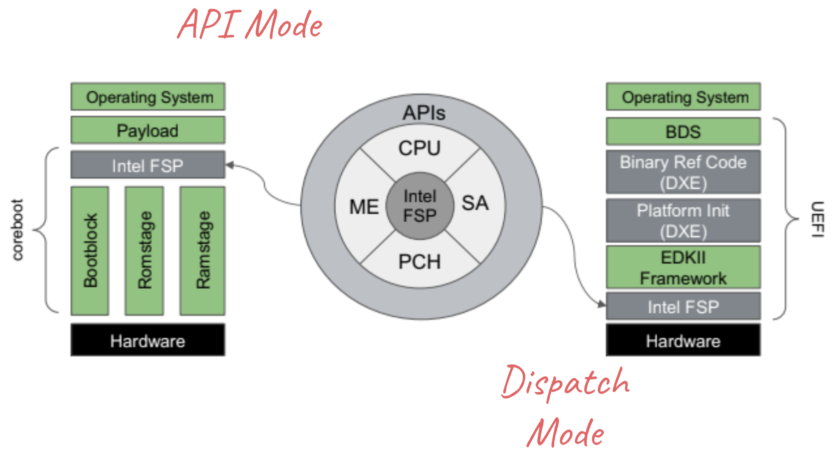
Silicon Reference Code from Intel



*In summary, **FSP** is a binary distribution model based on APIs (i.e., FSP-T/M/S) that includes blobs (responsible for silicon initialization), header files (for easy integration with boot firmware), and documentation (such as an integration guide).*

Source: https://images.anandtech.com/doci/20046/Meteor_Lake_Hotchips%20_%20Wilfred%20_%20final_submit%20%281%29-24.png

FSP fits into Boot Firmware's world



FSP integration with different boot firmwares

FSP	**coreboot**	**UEFI**
FSP-T (Temporary Memory Init)	bootblock	SEC
FSP-M (DRAM Init)	romstage	PEI
FSP-S (Silicon Init)	ramstage	PEI
FSP-Notify (Security and Lock-down)	ramstage	DXE

- At high level FSP can be viewed as three big firmware volume (FVs). i.e., FSP-T/M and S.
- The calling convention differs based on the integration of the boot-firmware
  - **API Mode**: The boot firmware interacts with the FSP through a series of well-defined API calls to have the higher control of the boot process and order of the Silicon Init tasks.
  - **Dispatch Mode**: Designed for better integration into UEFI-based boot firmware where FSP exposes firmware volumes directly to the boot firmware w/o any API calls.

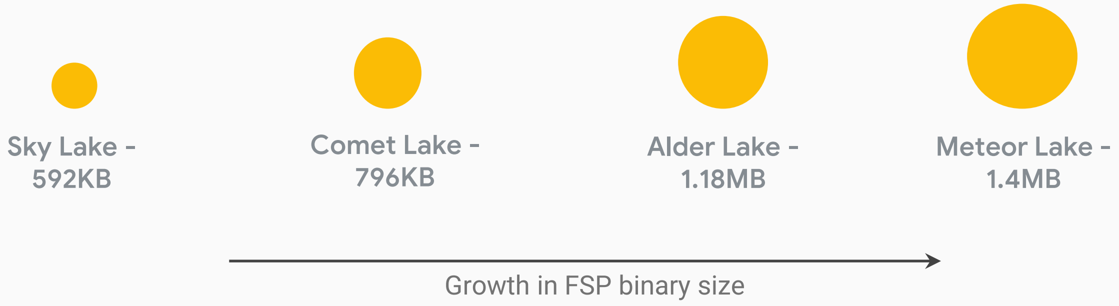
Pros and Cons of FSP

Pros:

1. Helped to speed up the new SoC bring up.
2. Getting tested silicon reference code which can be used as is with any boot firmware (open source and/or closed source).
3. Silicon vendors are able to protect their confidential code but still able to support OSF (Open Source Firmware) development.

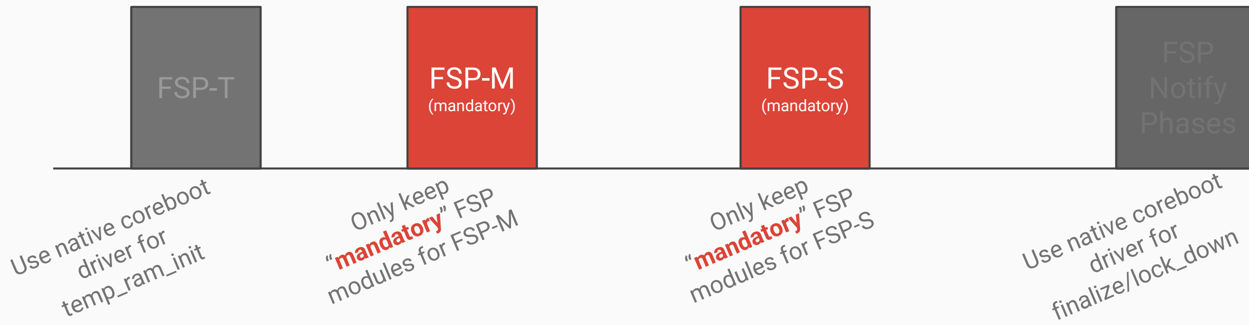
Cons:

1. FSP is a mammoth binary blob model that compliments all SoC features.
2. FSP configuration is limited to the runtime execution flow.
3. Modern Boot Firmwares are too competent enough to take ownership of platform and SoC initialization.
4. Definition of "**mandatory**" and "**non-mandatory**" components are missing in FSP binary model.
5. FSP has lacked in defining KPIs for boot time and SPI size.
6. FSP's desire to do more and own more which makes it bloated.



Goal

Create an "**Alternative Path**" towards Open Source Firmware Development - starts with reduced FSP


FSP-T
FSP-M
(mandatory)
FSP-S
(mandatory)
FSP Notify Phases
Use native coreboot driver for temp_ram_init
Only keep "mandatory" FSP modules for FSP-M
Only keep "mandatory" FSP modules for FSP-S
Use native coreboot driver for finalize/lock_down

Stepwise approach to get rid of “non-mandatory” FSP modules

Step 1

Follow the rules to start identifying the “non-mandatory” FSP Modules

- Enhance the coreboot/open source boot firmware capability so that it can be used as a replacement for any closed source FSP module.
- Give more importance towards SoC programming documentation to reflect the register programming using OSF.
- Execute the native OSF code instead of calling into closed source FSP modules.
- Need better tooling support at the FSP side to able to eliminate those “**non-mandatory**” aka unused modules.

Application of Step 1

Eliminate the FSP-T (aka TempRamInit) blob

- Upstream coreboot has support for Cache-as-RAM using different methodology
  - **INTEL_CAR_NEM** - Use basic NEM (non-evictionm mode)
  - **INTEL_CAR_NEM_ENHANCED** - Use advanced form of NEM

- SplitFspBin.py is a python script to split the FSP 2.x binary into FSP-T/M/S blobs.

```
python SplitFspBin.py split [-h] -f FSPBINARY [-o
OUTPUTDIR] [-n NAMETEMPLATE]
```

- For example:
  - **python SplitFspBin.py split -f FSP.bin**

It will create FSP_T.bin, FSP_M.bin and FSP_S.bin in current directory.

- Don’t stitch FSP-T blob into the CBFS (aka part of final AP FW image).

Application of Step 1

Eliminate the FSP-Notify Phase APIs (part of FSP-S blob)

- Upstream coreboot has support to skip calling into FSP-Notify Phase APIs
  - **USE_FSP_NOTIFY_PHASE_POST_PCI_ENUM**
  - **USE_FSP_NOTIFY_PHASE_READY_TO_BOOT**
  - **USE_FSP_NOTIFY_PHASE_END_OF_FIRMWARE**

- Relying on the `.final` hooks in coreboot to perform required SoC programming instead calling into FSP Notify Phase APIs

Sample code from cse.c file

```
static void cse_final(struct device *dev)
{
if (!CONFIG(USE_FSP_NOTIFY_PHASE_READY_TO_BOOT))
                              cse_final_ready_to_boot();

if (!CONFIG(USE_FSP_NOTIFY_PHASE_END_OF_FIRMWARE))
                              cse_final_end_of_firmware();
}
```

Application of Step 1

Use native coreboot drivers over FSP-M and FSP-S modules

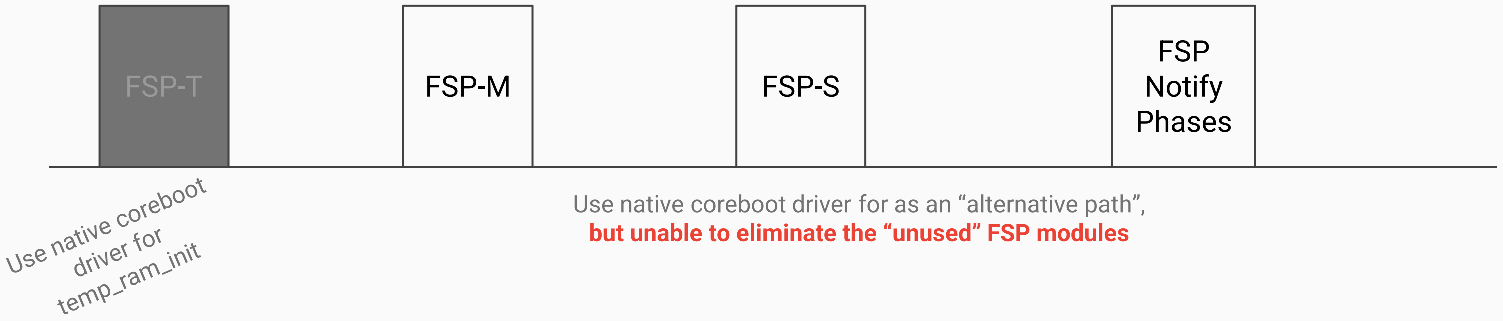
- coreboot relies on native GPIO driver to program the platform and SoC GPIOs rather executing GPIO driver/library part of FSP-M APIs

```
/* Skip GPIO configuration from FSP */
m_cfg->GpioOverride = 0x1;
```

- Allow to run coreboot native code for CPU (BSP and APs) feature programing rather than running FSP-S modules
    - Platform to select “**DROP_CPU_FEATURE_PROGRAM_IN_FSP**” config to bring CPU feature programming ownership to coreboot.

- Relying on open-source libgfxinit can further help to remove the dependency over GFX PEIM and/or uGOP for display initialization (part of FSP-S API)

Relook at what we have achieved so far...



Use native coreboot driver for as an “alternative path”,
but unable to eliminate the “unused” FSP modules

Step 2

Rules to eliminate the FSP "non-mandatory" modules using better tooling

- Divided the FSP PEIM modules into two categories
  - **Must Have/Mandatory**: Mostly boot critical modules which are Intel's IP code block. Example: Memory Training algorithms.
  - **Good-to-have/Non-mandatory**: FSP modules those are not boot critical and easy to implement using alternative boot firmware as well.
- If OSF boot firmware has an alternative implementation then such modules inside FSP can be considered "**non-mandatory**". For example: TempRamInit as part of FSP-T.
- Intel Firmware Module Management Tool (Intel® FMMT) is a utility that is capable of removal, addition, and replacement of FFS files in FV image binaries.
- Use the FMMT utility to be able to drop the "**non-mandatory**" FSP modules from the given FSP blob.

Application of Step 2

Eliminate the “non-mandatory” **FSP modules**

- Drop the FSP PEIM modules using FMMT

```
-d < Inputfile > < TargetFvName/TargetFvGuid > < TargetFfsName > < Outputfile >

    -   Delete the Ffs from Inputfile. TargetFfsName (Guid) is
        the TargetFfs which will be deleted.

Ex: python FMMT.py -d Fsp.fd 6938079b-b503-4e3d-9d24-b28337a25806 FspS3Notify Fsp.fd
```

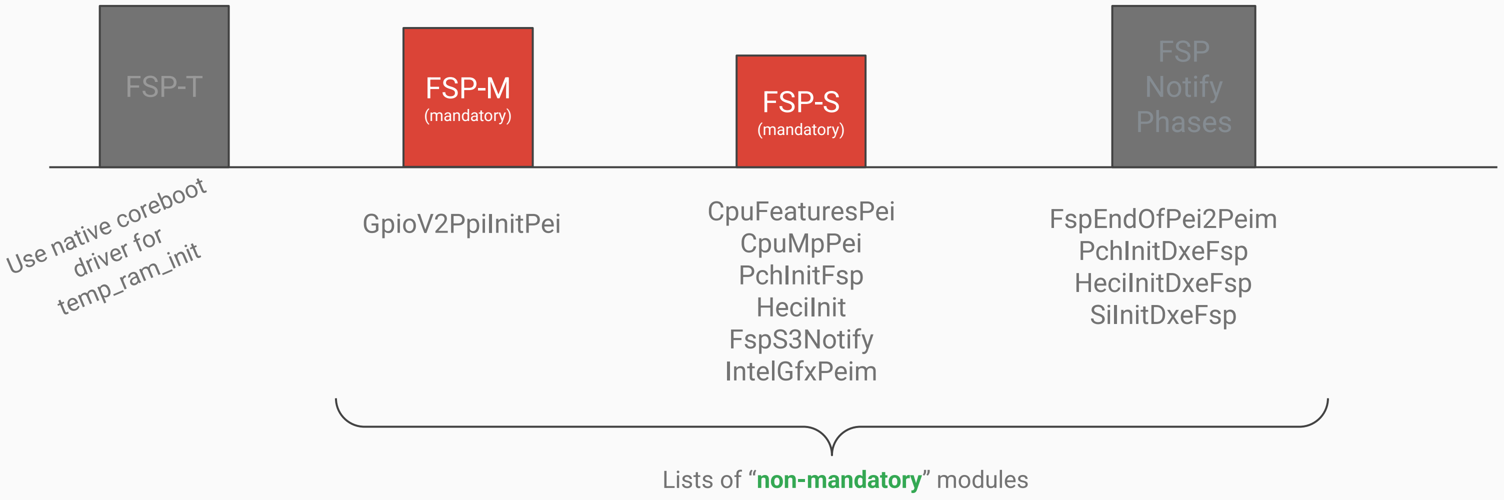
- Shrink the FSP Firmware Volume FMMT (after dropping the FSP modules)

-s < Inputfile > < Outputfile >

- Shrink the *Inputfile* Firmware Volume and create the newer *Outputfile*

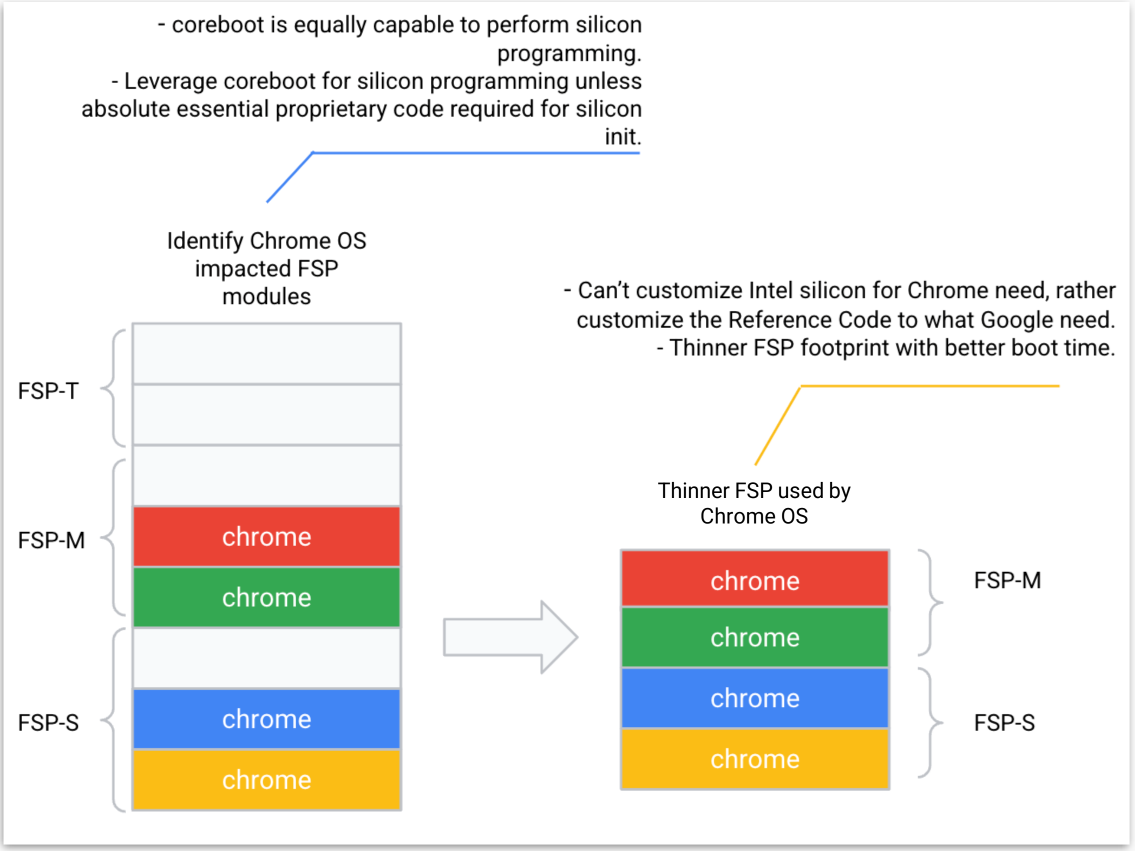
```
Ex: python FMMT.py -s Fsp.fd Fsp.fd
```

Relook at what we have achieved so far…



Demonstration

- Bring more open-source code in Intel Meteor Lake development using coreboot over FSP.
- FMMT helped to drop total **11 PEIMs across FSP-M and FSP-S.**
- Able to reduce the Intel Meteor Lake FSP blob size from **1.4MB to <1MB in this approach.**
- A thinner FSP is also boot efficient while integrating with coreboot.



- coreboot is equally capable to perform silicon programming.
- Leverage coreboot for silicon programming unless absolute essential proprietary code required for silicon init.
Identify Chrome OS impacted FSP modules
- Can't customize Intel silicon for Chrome need, rather customize the Reference Code to what Google need.
- Thinner FSP footprint with better boot time.
Thinner FSP used by Chrome OS
FSP-T
FSP-M
FSP-S
chrome
chrome
chrome
chrome
chrome
chrome
chrome
chrome
FSP-M
FSP-S

Q&A

More information

- System Firmware https://link.springer.com/book/10.1007/978-1-4842-7939-7
  Firmware Development https://link.springer.com/book/10.1007/978-1-4842-7974-8
- Creating your own FSP
  https://blog.osfw.foundation/breaking-the-boundary-a-way-to-create-your-own-fsp-binary/
- FSP https://github.com/intel/fsp/wiki repositories
  https://github.com/intel/FSP https://github.com/coreboot/fsp
- coreboot FSP https://doc.coreboot.org/soc/intel/fsp/index.html
- Tools
  https://github.com/tianocore/edk2/blob/a64b944942d828fe98e4843929662aad7f47bcca/BaseTools/Source/Python/FMMT/README.md
  https://microsoft.github.io/m u/dyn/mu_basecore/BaseTools/Source/Python/FMMT/
  https://github.com/LongSoft/UEFITool

OPEN
SOURCE
FIRMWARE
— FOUNDATION

Thank you!