Open Source Firmware

Breaking the Boundary: A Way to Create Your Own FSP Binary

Subrata Banik

Feb 24, 2023 • 11 min read

When there's no boundaries, that's when you can get creative.

Author: Subrata Banik (subratabanik@google.com)

Collaboration with: Vincent Zimmer (vincent.zimmer@intel.com)

***TL;DR**: This document demonstrates a path forward that breaks the boundary of using FSP (Firmware Support Package) for firmware development, by defining a way to create your own custom FSP blob(s) that meets and aligns with your target product requirement.*

A previous article written by me sometime last year named **Refining Open-Source Firmware Support for the Intel Platform** initiated the thought about bringing openness in firmware development even while working with closed-source silicon reference code. Although the previous article was more specific to detail the challenges seen with Intel SoC-based platform enablement using open source boot firmware aka coreboot, in practice the situation is not much different with other SoC vendors as well. To understand this problem in more detail, one should additionally refer to our last year OSFC talk named The "Thing" Around Your System Firmware.

The prior article had called for action to improve the overall platform-enabling environment that uses the open-source firmware development model with closed-source silicon reference blobs. Below is the list of the work items that came out from that discussion:

- Improve the platform enablement environment by balancing out the `binary blob model` (only include the *mandatory* closed source blobs

which can't be open-source) and focusing on bringing openness in silicon code by leveraging open-source boot firmware e.g. coreboot.

- Reduce the boundaries of proprietary firmware running on Host CPU Firmware a.k.a Intel Firmware Support Package (FSP).
- Classify the FSP modules as `***mandatory***` and `***good to have***`. Allow dropping of `*good to have*` FSP modules to leverage more open source coreboot libraries/drivers for platform bring-up. This effort would help to reduce FSP boundaries and eventually optimize the SPI flash footprint.
- Optimizing the boot path would eventually help to achieve the ambitious goal of fast booting up the system firmware (in < 1 second boot time).

This document is to capture the progress being made to solve such ambitious challenges and provide flexibility to the boot firmware designer/developer aka users of the FSP to create their own FSP blobs (essentially could be different than what Intel "*officially*" uploaded into the FSP Github for each SoC product). Further sections of this document will provide more technical details about solving this challenge and how it would benefit the open-source community.

"One thing that can be clearly claimed already at the starting of this article is the big shortcoming of the current FSP delivery: the "one-binary-fits-all and eventually bloated" blob is now getting diminish with this approach where the consumers of

the FSP binary have the freedom to create their own FSP binary that fits well in the open source firmware development model with coreboot."

The Planning

***TL;DR**: This section illustrates the design aspect of creating a custom FSP blob solution.*

The "**Alternative Path**" section from the previous article discussed the few possibilities that the open-source firmware development approach needs to consider to optimize the usage of proprietary FSP modules over the coreboot libraries/drivers. This section illustrates the design philosophy being used to identify the mandatory FSP modules and how to eliminate them (as there might be more modules optional). This effort results in achieving the "**Gain Control of Platform Initialization using native coreboot driver**" which was briefly described in the previously written document.

For ease of understanding let's describe the SoC in more firmware-friendly terms: the SoC design incorporates various IPs (Intellectual Property) to provide a great set of capabilities for the different platforms to choose from.

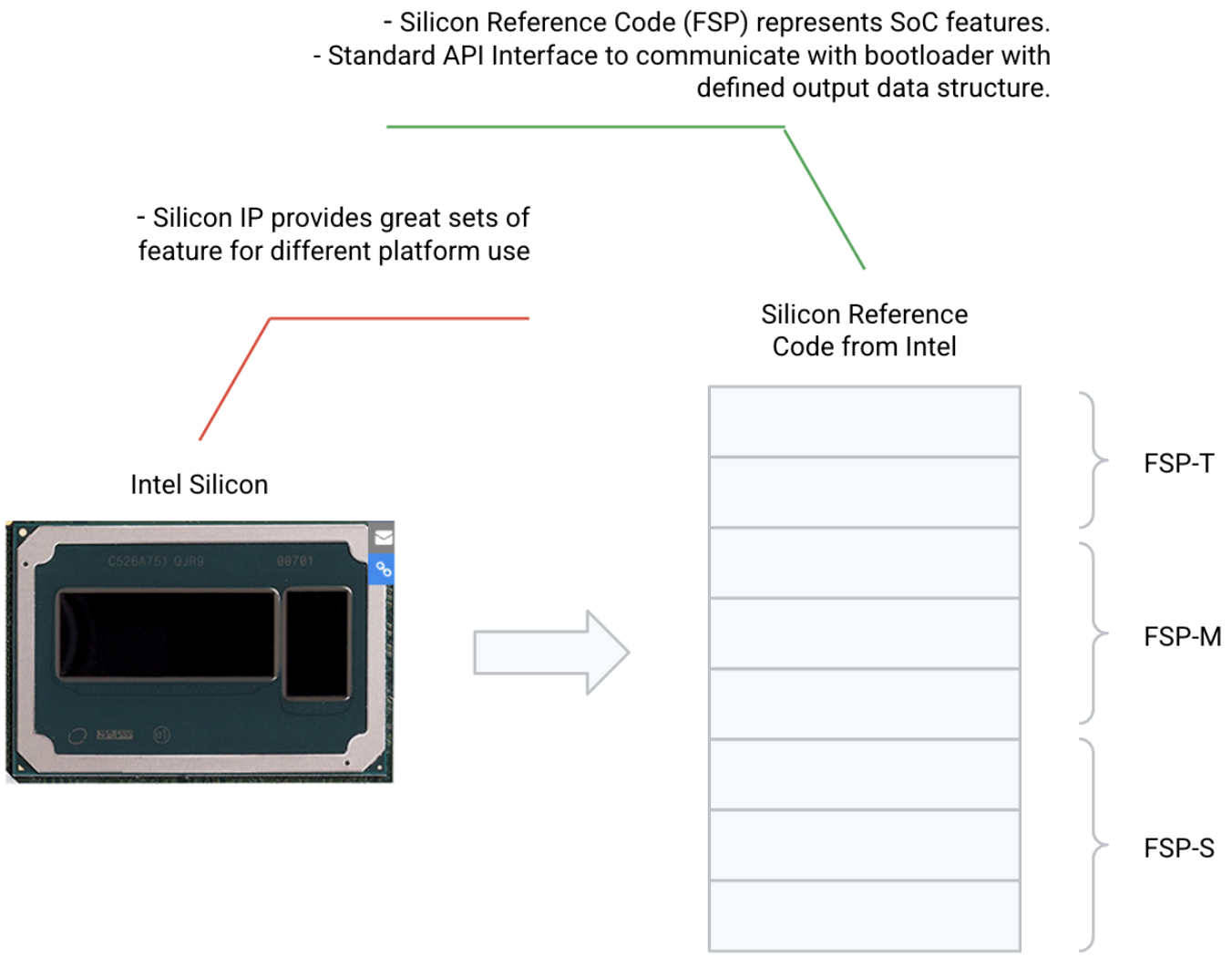

Figure 1.0: Existing Intel SoC Reference Code Design

An IP can provide different hardware interfaces for example audio IP provides interfaces (like High-Definition Audio or HDA, Non-HDA a.k.a. I2S and Soundwire) for the target platform to eventually select and perform the configuration. The firmware being closest to the hardware is responsible for such interface selection, and the Intel SoC platform is no exception to that, where silicon reference code represents the SoC features. Figure 1.0 illustrates the relationship between SoC and silicon reference code.

The missing part in this whole evolution process is the understanding of the end-user platform requirement. Intel Silicon reference code enabling model only focuses on delivering a unified blob irrespective of the different OS-based platform needs. It’s very obvious that silicon vendors can't customize the silicon for every end-user product requirement but the minimum expectation is to allow configurable silicon reference code as per the platform need. The reference code, being designed to show all the capabilities of the new platform, usually does more than the default user actually needs (and of which a big portion is re-done in the firmware then) which in turn leads to an overloaded FSP piece after piece.

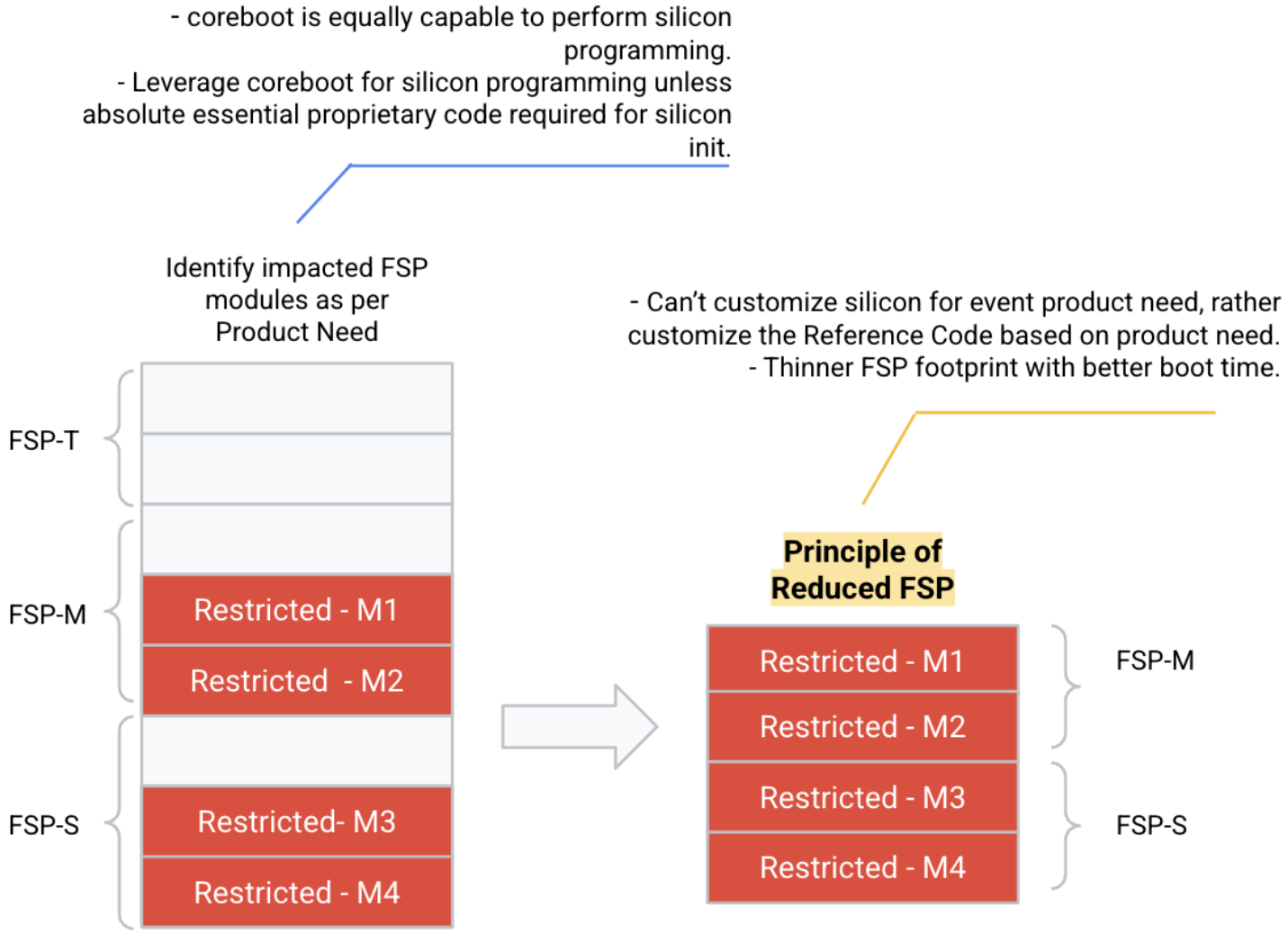

Figure 1.1: Proposed Intel SoC Reference Code Design

Based on this proposed design principle we shall provide the platform owners the opportunity to create an essential silicon reference code for the target platform. In fact, silicon reference code is not only meant to pass the board configuration parameter but also to remove unused or redundant IP initialization that resonates with the target board design. Figure 1.1 illustrates the design principle that results in optimized silicon reference code based on the target platform.

To summarise, the work items based on the design phases are:

1. Enhance the coreboot/open source boot firmware capability so that it can be used as a replacement for any closed source FSP module (preferably where register sets are well explained as part of the datasheet and OEM programming section ask to implement the mandatory silicon programming in the boot firmware). For example, coreboot implemented CAR using a native driver/library and helped to make the FSP-T optional starting with FSP 2.0 specification.
2. In the open source development model, the highest preference is to execute the native code of the open source firmware instead of calling into closed source FSP modules but at the same time needs to have a way to eliminate the unused FSP modules from the SPI Flash. This effort

will help to reduce the maintenance towards incorporating FSP fixes and additionally reduce the SPI flash space occupied by the FSP binary.

3. Need better tooling support at the FSP side to able to achieve the #2 above.

The Execution

***TL;DR**: This section illustrates the stepwise approach that leads into meeting the goal.*

In practice, meeting the ambitious goal of designing the custom FSP binary won't be possible without a detailed and gradual approach to it. These approaches might be interconnected or may appear independent but eventually required altogether to meet the desired goal (for the interest of the readers and easy understanding, the numeric ordering being used here to describe the stepwise approach).

Step 1: Enhancing the Open Source Firmware capability to suppress the need for Closed Source Firmware Blobs

Since the evolution of the Intel FSP, the main goal is to reduce the effort at the boot firmware side and use the FSP blobs as a drop-in solution to create final

production AP firmware. It serves to meet the goal of platform enablement taking place with pre-validated SoC code.

Unfortunately, due to several reasons (for example FSP being bloated with more than required functionalities and lack of roles and responsibility) the entire FSP binary drop-in model has been unable to prove its inevitability. Over the period of time, the boot firmware has evolved and become more powerful in terms of taking more responsibility for the platform enablement without relying on the 3rd party blob solution.

Table 1.0 shows the comparison between coreboot and FSP. coreboot is one of the powerful boot firmware, which is capable of doing more than what FSP is actually expecting from the bootloader.

Table 1.0: Comparison between coreboot and FSP phases

PURPOSE	COREBOOT	FSP
Temporary Memory Init	bootblock	FSP-T
DRAM Init	romstage	FSP-M
Silicon Init	ramstage	FSP-S

For example, the primary responsibility of the FSP-Notify Phase APIs is to meet the SoC programming requirements (described in the Firmware Architecture

Specification (FAS) Chapter 11, known as the `Security Guideline`). The intent of this section is to ensure all OEM designs are meeting the SoC vendor-provided security guideline requirements. This section of the FAS is purely meant for the OEM/ODM design to comply with using underlying boot firmware (in the case of ChromeOS, it applies to the coreboot) but FSP still prefers to perform those recommended chipset programming on its own which often at an execution time contradicts with the open source firmware flow.

> **Fact**: Unfortunately, this guideline is not even true for FSP executing in dispatch mode (alternative to the API mode) used by UEFI aware bootloaders. In dispatch mode, the UEFI bootloader is not abide by the fact to use FSP-Notify Phase APIs.

The `**Gain Control**` proposal described in the previously written article helped to enhance the open-source firmware capabilities. Now with this approach, coreboot can eliminate the need to have multiple FSP modules due to redundant functionality. A total of 8 FSP modules from FSP-S Firmware Volume can be now dropped.

Additionally, another work item that intended to replace the FSP modules for multi-processor initialization with open-source coreboot drivers has shown that yet two more FSP modules can become optional.

Another work item that currently is in progress is to adopt open source *libgfxinit* to replace FSP GFX PEIM performing Pre-Boot display Initialization.

In summary, out of roughly 13 PEIM modules the FSP-S consists of, 10 modules can be eliminated. This leads to a reduction of ~80%.

Step 2: A way to eliminate the FSP modules using better tooling

The design principle of the silicon programming blob is to dynamically configure the IP interface based on the boot firmware provided inputs (as per the target board schematics). But the underrated part is the capability inside FSP to be able to statically decouple the `***good to have FSP modules***` from the `***mandatory ones***` at the source. Hence, in the present scenario, FSP doesn't provide any option to eliminate the unused modules from the FSP blob and to reduce the SPI Flash usage. This limitation leads to keeping unused/dead binaries in the SPI Flash and eventually bearing the additional BoM cost.

The earlier section (as Step 1) illustrates the path to make a number of FSP modules possibly **unused** while integrating with open-source coreboot as boot firmware. But the actual benefit of boot time improvement or the SPI size reduction won't have been achieved unless there is a scalable way to be able to drop those *claimed unused FSP modules* (without modifying the FSP source code). The fact is that most likely consumers of the FSP are directly consuming FSP binaries for their platform enablement. Hence, the most specific need is to be able to remove FSP Firmware Files (FFS) from the Firmware Volume using simple command line arguments (without modifying the source code).

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. This utility belongs to the open-source EDK2 github repository (link here https://github.com/tianocore/edk2/blob/a64b944942d828fe98e4843929662aad7f47bcca/BaseTools/Source/Python/FMMT/README.md).

Our target was to use the FMMT utility to be able to drop the unused modules from the given FSP blob. But the limitation of FMMT utility is that after removing an FFS file, the utility is unable to shrink the firmware volume space, which results in having the unused memory space being occupied by the firmware volume and eventually increasing the cost of the OEM devices (due to maintaining the higher SPI Flash footprint).

Good News is that with the bug being filled on the EDK2 to add the `shrink` support into the FMMT utility and the EDK2 community has responded positively by adding the shrinking support into the latest FMMT release (part of the EDK2 latest repository).

With the latest FMMT tool, we are now able to drop any "*good to have*" FSP file system (FFS) from the FD (Firmware Device) after following the few simple steps below:

1. Remove an FSP module FFS by providing the module GUID

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

- Delete the Ffs from *Inputfile*. TargetFfsName (Guid) is the TargetFfs which will be deleted. *TargetFvName/TargetFvGuid* is optional, which is the parent of TargetFfs*.*
- Ex: py -3 FMMT.py -d Ovmf.fd 6938079b-b503-4e3d-9d24-b28337a25806 S3Resume2Pei output.fd

2. Shrink the FSP Firmware Volume using the newly added feature of FMMT.

-s < Inputfile > < Outputfile >

- Shrink the *Inputfile* Firmware Volume and create the newer *Outputfile*
- Ex: py -3 FMMT.py -s Ovmf.fd output.fd

Step 3: FSP Integration Guide to list out "*good to have*" FSP Modules

Since the origin of the FSP specification the entire FSP Firmware Device (FD) is considered mandatory and boot firmware is supposed to make calls into each and every entry point to let the FSP modules get a chance to execute and perform the silicon initialization. For the first time with FSP 2.0 specification, Intel has made FSP-T (aka for performing temporary memory initialization) an optional API. With this evolution, we are asking to even do more deep down inside each FSP firmware volume and classify modules inside each FSP Firmware Volume as

"**Mandatory**" (aka Must Have, which shouldn't be dropped as it might have an adverse effect on the platform) or "**Good To Have**" (aka Optional, which can be dropped as per the decision made by boot firmware).

Based on recent communication with the concerned team on the FSP side, the FSP team has now agreed to capture such a "**Good To Have"** module list per SoC inside the FSP integration guide. This would help the boot firmware aka consumer of the FSP, with the ability to actually decide what goes inside the FSP Firmware Volume aka FSP blob.

This **Good To Have** FSP module list can be used now along with the FMMT tool to be able to achieve our goal of creating our own custom FSP blob as per the target product requirements.

The Outcome

The final section of this document shows the benefit of this approach in terms of the data and other improvements that this evolution brings to the users of open-source firmware (working along with the FSP-like binary PI (platform initialization) model).

1. Having the flexibility to choose Open Source Firmware over a closed-source binary model even for silicon initialization is great freedom from the product

designer side. The users of the open-source firmware would be able to appreciate it much better.

2. This evolution would help to bring more adoption towards the Hybrid Work Model (a combination of closed and open source firmware used for platform enablement) as now the users of the closed source blob have the flexibility to chop down redundant/unrequired pieces.

3. OEM/ODM firmware engineers can create their own customized FSP binary which meets their product requirements.

4. Overcoming the shortcomings of the FSP model is often described as a "*single FSP binary meeting all customer need problems and very much bloated*".

5. Optimise the boot time significantly with the limited FSP modules and lesser infrastructure overheads due to UEFI.

6. Able to meet lower SPI Flash which is a product distinguishing feature. Figure 1.2 illustrates the amount of reduction that one could achieve in the SPI Flash size with the flexibility of being able to drop unused FSP modules as per the target platform's need.

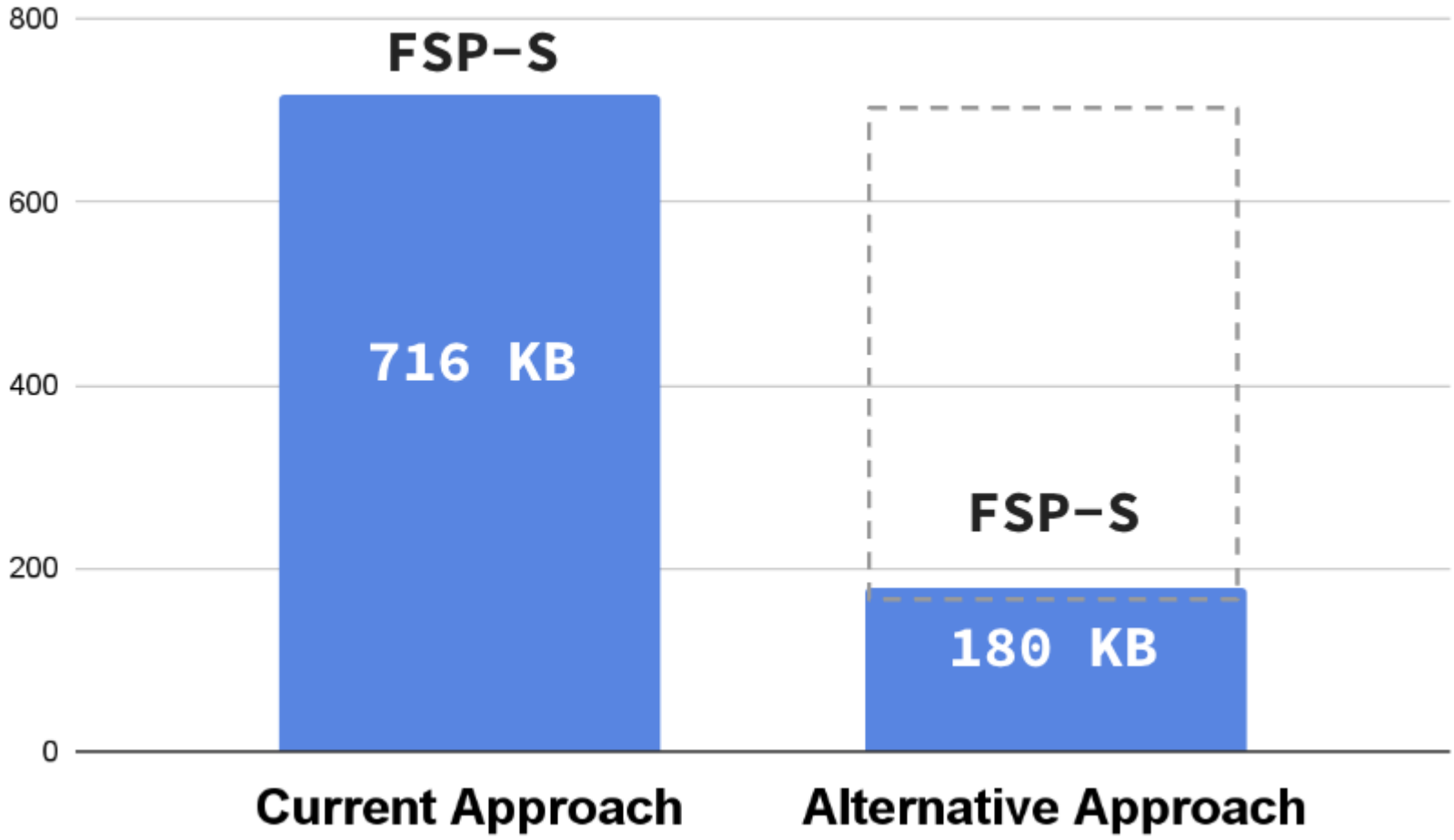

Figure 1.2: Able to reduce FSP-S usage by ~500KB after dropping 10 modules

This proposal is readily available for all the latest Intel platforms starting with Alder Lake while working with coreboot in API mode.

Summary

Refining Open-Source Firmware Support for the Intel Platform had initiated the thought about defining a more open-source-friendly firmware development environment and with the “Gain Control '' proposal, the idea was to provide more flexibility while designing AP firmware using open-source boot firmware. This document has been drafted to capture the progress being made in the paths since the origination of this idea.

Additionally, so far we have only outlined the scope for improvement in FSP-S, which is the tiniest blob in the whole Open Source AP firmware FSP integration process. An actual saving in boot time and SPI Flash size could have been achieved if we were able to bring modularity in FSP-M design where platform owners are able to customize the MRC block as per their need, for example, most of Alder Lake based reference designs have shipped with LPDDR4x DIMM alone but the growth in ADL-MRC is multiple times compared to the previous generation (Tiger Lake SoC platform). We need to understand how much additional opportunity we have to be able to customize FSP-M blobs based on the final product memory type. Because remember our goal is to be ***able to customize the silicon reference code although we might not be able to customize the actual underlying silicon as per the target product***.

Join our Newsletter

Enter your email Subscribe

post-pandemic programming party!

"It's been two years of maybe ..." but ... we did it! We had a hackathon at TU Darmstadt. I had mentioned I'd be in the area to Felix Singer in early June, and Felix proposed a hackathon ...

Jun 25, 2022 — 3 min read

Refining Open-Source Firmware Support for the Intel Platform

This proposal is intended to discuss a path forward towards getting openness in system firmware development with Intel...

Jun 3, 2022 — 8 min read