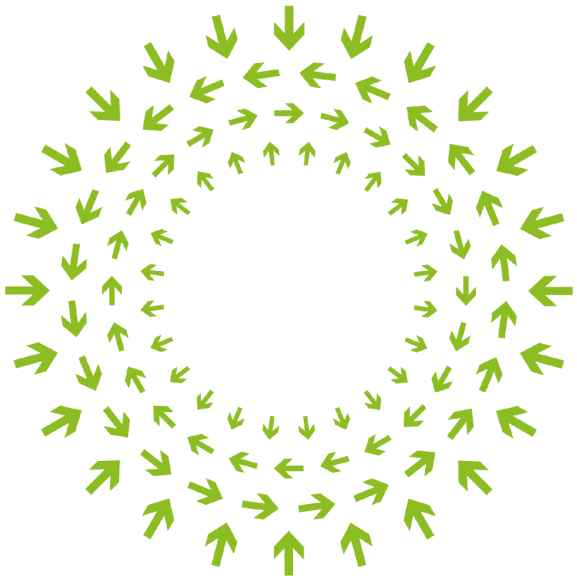


OPEN

Compute Project

Open Platform Firmware

Open Silicon Firmware Interface (openSFI) Firmware Architecture Specification (FAS)

Version 0.3

Authors:
Raj Kapoor (AMD), Albert Rennie (AMD), Srini Narayana (AMI), Tim Lewis (Insyde)

Collaborators:

Allison Goodman (Intel), Michael Kinney (Intel), Felix Polyodov (AMI), Giri Mudusuru (Microsoft), Ian Goegebuer (Google), Paul Grimes (AMD), Pradeep Gopal (AMD), Christian Walter (9 Elements), Senthil Ramakrishnan (ARM), Jose Marinho (ARM), Jean-Marie Verdun (HPE), Nil Ge (ByteDance), Hancock Chang (MiTAC), Piotr Król (3mdeb), Amos Bu (OSFF)

Sept. 2025

CC BY 4.0

This work is licensed under a Creative Commons Attribution 4.0 International License.

Voting Roster for openSFI v0.3 Ratification

Organization	Representative	Vote
3mdeb	Piotr Król	➕ 1
9elements	Christian Walter	➕ 1
Advanced Micro Devices (AMD)	Raj Kapoor	➕ 1
AMI US Holdings Inc.	Srini Narayana	➕ 1
ARM	Senthil Ramakrishnan	➕ 1
ByteDance	Nill Ge	➕ 1
Google	Ian Goegebuer / Amber Huffman	➕ 1
Hewlett Packard Enterprise (HPe)	Jean-Marie Verdun	➕ 1
Insyde Software	Tim Lewis	➕ 1
Intel	Allison Goodman	➕ 1
Microsoft	Giri Mudusuru	➕ 1
MiTAC	Hancock Chang	➕ 1
Open Source Firmware Foundation (OSFF)	Christian Walter	➕ 1

This work is licensed under a Creative Commons Attribution 4.0 International License.

CC BY 4.0

Version History

Date	Version #	Author	Description
27 MAY ‘25	0.0.1	Paul Grimes	Initial Release
02 JUN ‘25	0.3 DRAFT	Albert Rennie Raj Kapoor	- Added Technical details throughout the document. Ready for OPF review. - Added Chapter 10 - Abstraction Layer - Added Chapter 11 - Platform Configuration - Overhauled Chapter 11 - Platform Input Configuration Semantics - Added Appendix B - Clarifying Notes - Added Appendix C - Unified Enumerated Stages List - License changed back to CC BY 4.0 per discussion in Mar 2025 F2F - Added clarifying notes for SOCID & ExecuteSiliconRuntimeOp() at the bottom of Section 8.2 - Added Clarifying Notes for VAL in the Terminology Section
23 SEP’25	0.3	Albert Rennie Raj Kapoor	Voting Roster for OPF Ratification - COMPLETE Ready for OCP Steering Committee WG Review - This document will be frozen, and v0.5 DRAFT will be started based on this version. - **OCP SC Approved - 09 SEP’25 (v0.3 FROZEN)**

This work is licensed under a Creative Commons Attribution 4.0 International License.

CC BY 4.0

Table of Contents

Voting Roster for openSFI v0.3 Ratification 1
Version History 3
License 6
1. Acknowledgements 7
2. Compliance with OCP Tenets 8
2.1. Openness 8
2.2. Efficiency 8
2.3. Impact 8
2.4. Sustainability 8
3. Terminology 9
4. Scope 10
5. Problem Statement 10
6. Introduction 13
6.1. Description 13
6.2. Scope 13
6.3. Goals 13
6.4. Interface Compatibility Testing 14
6.5. Standards Interoperability (Non-Normative) Compliance 14
7. Architecture Overview 16
8. API Definition 17
8.1. InitializeSilicon() 17
8.2. ExecuteSiliconRuntimeOp() 18
9. Initialization Stages 20
9.1. SFI_INIT_STAGE_LIB_INIT - Library Initialization Metadata Query Stage 20
9.2. Stage Execution Order Manifest (Normative) 21
10. Platform Input Configuration Semantics 23
11.1 Purpose and Scope 23
11.2 Vendor Responsibility for Schema Definition 23
11.3 Format of Input and Output Structures 23
11.4 Relationship to UPCI 24
11.5 Error Handling for Memory Allocation 24
11.6 Post DRAM Initialization Memory Availability 24
11. Standards Alignment 25
12. Governance 26
Appendix A PascalCase Naming Convention (with Capitalized Acronyms) 27
1. Naming Convention for Constants 27
Example: 28
Why? 28
Additional Tip: 28

This work is licensed under a Creative Commons Attribution 4.0 International License.
CC BY 4.0

2. Naming Convention for Functions, Variables, and Parameters....28
Rules for Naming....28
Examples....29
Special Notes....29
Appendix B - Clarifying Notes....30
Appendix C - Unified Enumerated Stages List....32
Appendix D - Supporting Documentation & Guidance....33
1. openSFI FAS (openSFI Firmware Architecture Specification)....33
2. openSFI SSImG (openSFI Silicon Specific Implementer's Guide)....33
3. openSFI SSInG UEFI (openSFI SoC-Specific Integrator's Guide for UEFI Host Firmware)....34
4. openSFI SSInG coreboot (openSFI SoC-Specific Integrator's Guide for coreboot Host Firmware)....34
Appendix E - Abstraction Layer (Defined Interface Spec)....36
12.1. Abstraction Layer (Defined Interface Spec)....36
12.2. Vendor Adaptation Layer (VAL / Shim Layer)....36
12.3. Lifecycle of the VAL Library....37
12.4. Design Goals and Benefits....37
12.5. Illustration – Conceptual Mapping....38
Appendix F....39

This work is licensed under a Creative Commons Attribution 4.0 International License.
CC BY 4.0

License

openSFI Firmware Architecture Specification is licensed under **CC BY 4.0**. To view a copy of this license, visit https://creativecommons.org/licenses/by/4.0/

Creative Commons

OCP encourages participants to share their proposals, specifications and designs with the community. This is to promote openness and encourage continuous and open feedback. It is important to remember that by providing feedback for any such documents, whether in written or verbal form, that the contributor or the contributor's organization grants OCP and its members irrevocable right to use this feedback for any purpose without any further obligation.

It is acknowledged that any such documentation and any ancillary materials that are provided to OCP in connection with this document, including without limitation any white papers, articles, photographs, studies, diagrams, contact information (together, “Materials”) are made available under the Creative Commons Attribution 4.0 International License found here: https://creativecommons.org/licenses/by/4.0/, or any later version, and without limiting the foregoing, OCP may make the Materials available under such terms.

As a contributor to this document, all members represent that they have the authority to grant the rights and licenses herein. They further represent and warrant that the Materials do not and will not violate the copyrights or misappropriate the trade secret rights of any third party, including without limitation rights in intellectual property. The contributor(s) also represent that, to the extent the Materials include materials protected by copyright or trade secret rights that are owned or created by any third-party, they have obtained permission for its use consistent with the foregoing. They will provide OCP evidence of such permission upon OCP’s request. This document and any "Materials" are published on the respective project's wiki page and are open to the public in accordance with OCP's Bylaws and IP Policy. This can be found at http://www.opencompute.org/participate/legal-documents/ . If you have any questions, please contact OCP.

This work is licensed under a Creative Commons Attribution 4.0 International License.

CC BY 4.0

1. Acknowledgements

The Authors of this Specification would like to acknowledge the following for their feedback:
3mdeb, 9elements, AMD, AMI, ByteDance, Google, Hewlett Packard Enterprise (HPE), Intel, Insyde, Microsoft, MiTAC

This work is licensed under a Creative Commons Attribution 4.0 International License.

CC BY 4.0

2. Compliance with OCP Tenets

2.1. Openness

Promotes openness and efficiency by enabling third parties to build, modify, and personalize a unified firmware interface that is compatible across multiple platforms and silicon architectures.

2.2. Efficiency

Reduce operational expenses, capital expenses and carbon emissions through a unified firmware interface. This enables efficient resource utilization, streamlines development processes, and eliminates redundant engineering efforts.

2.3. Impact

Has transformative impacts on the industry by accelerating time-to-market with standardized interfaces.

2.4. Sustainability

Enables optimization of code to reduce resource consumption.

This work is licensed under a Creative Commons Attribution 4.0 International License.

CC BY 4.0

3. Terminology

- **openSFI**: **O**pen-Source **S**ilicon **F**irmware **I**nterface. An unified interface between the Host Firmware and the Silicon Initialization firmware
- **UPCI**: **U**nified **P**latform **C**onfiguration **I**nterface. A spec to define various interpretable binary formats between data producers and consumers.
- **VAL**: **V**endor **A**daptation **L**ayer Library - An implementation of this openSFI Specification. Note: The VAL chapter has been relocated to the Appendix E in v0.3 of this specification. This is a transitional measure, intended to provide reference context while the community calibrates to the scope and boundaries of the main openSFI specification. The VAL content is currently non-normative, and the long-term plan is to gradually externalize this content, potentially into a separate standalone document by v0.8. All references to VAL within this specification will be reviewed and updated accordingly in future revisions.
- **Host Firmware**: Bootloader Firmware executing on a platform host processor (aka application processor) such as x86 CPUs based on AMD & Intel, or other ISAs such as an ARM SoC. Examples include coreboot (Dasharo and Libreboot etc.) and UEFI (EDK2, Aptio, H2O, etc).
- **SoC**: System on a Chip, including the CPU (Central Processing Unit).
- **API**: **A**pplication **P**rogramming **I**nterface.
- **SiP**: **Si**licon **P**rovider, aka Silicon Vendor
- **Factory Design Pattern**: A creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created (Reference - https://refactoring.guru/design-patterns/factory-method)

This work is licensed under a Creative Commons Attribution 4.0 International License.

CC BY 4.0

4. Scope

The specification addresses the lack of common firmware interfaces for host firmware initialization, leading to fragmented host firmware and increased costs.

The openSFI specification defines a unified interface between host firmware and silicon initialization components. It targets x86 SoCs (AMD and Intel) in its initial release but is designed to be ISA-neutral and extensible.

The scope includes:

- API definition for calling silicon initialization routine
- Runtime or post-boot API services
- Platform configuration integration (UPCI) if required
- Support for bootloader integration (UEFI, coreboot, etc.)

Out of Scope:

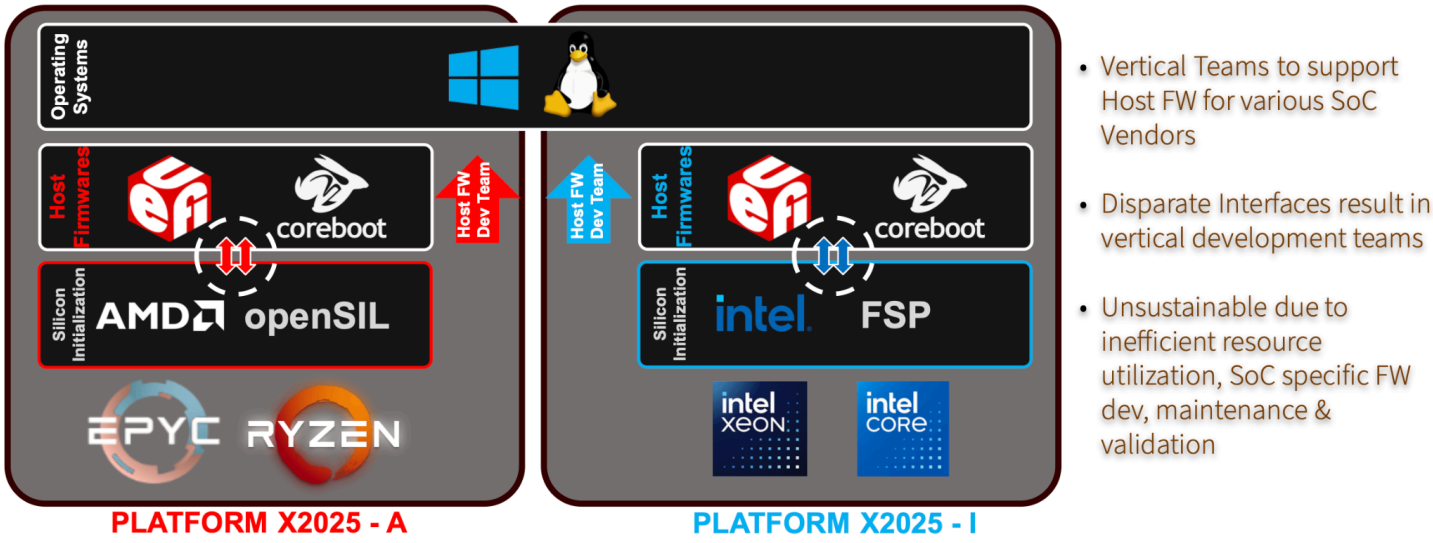
- Merging of vendor-specific silicon firmware implementations
- Non-x86 architectures (for this initial release)

This work is licensed under a Creative Commons Attribution 4.0 International License.

CC BY 4.0

5. Problem Statement

Modern day host firmware lacks a unified interface to interact with silicon initialization code from different silicon vendors. As a result, integrators are forced to implement and maintain vendor-specific boot flows, leading to fragmented designs, duplicated engineering efforts, and non-reusable codebases. Even when memory and I/O components are identical across platforms, firmware teams must rework their integration for each host SoC architecture. This results in extended development timelines, increased support overhead, higher maintenance costs, and inefficient resource utilization.



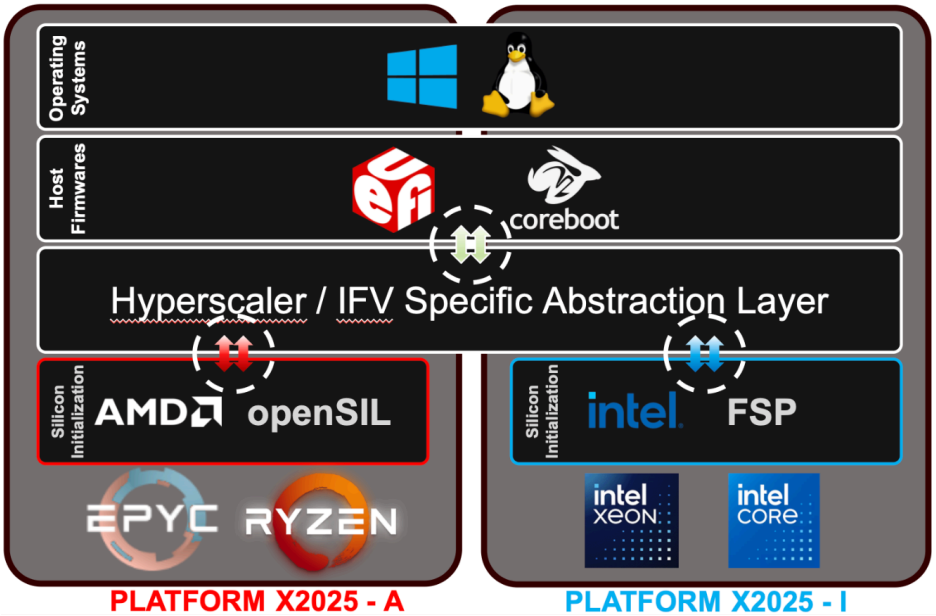
The design and maintenance of host firmware is increasingly unsustainable as silicon diversity grows. Platform integrators must support multiple silicon vendor SoCs, often within the same generation of products, even when memory and I/O subsystems remain unchanged. This diversity results in duplicated firmware stacks, fragmented debug processes, and increased costs.

Every silicon vendor currently defines its own binary or API interface for silicon initialization, leading to tight coupling between silicon and host firmware. The lack of a unified abstraction layer forces firmware teams to maintain separate branches for each SoC, reducing code reuse and increasing development overhead.

This work is licensed under a Creative Commons Attribution 4.0 International License.

CC BY 4.0

Attempts to create internal abstraction layers by platform owners (or IFVs) have shown to be fragile, requiring constant adaptation and refactoring to changes in vendor specific silicon initialization architecture. Furthermore, opaque silicon init flows hinder security auditing, transparency, and validation.



- Maintain abstraction layer specific to OEM/ODM/IFV
- Periodic architecture changes to Silicon Init results in unsustainability

Finally, redundant engineering and infrastructure overhead for duplicated firmware validation contributes negatively to sustainability goals. The absence of a common silicon initialization interface is a critical gap in the ecosystem that openSFI aims to address.

This work is licensed under a Creative Commons Attribution 4.0 International License.

CC BY 4.0

6. Introduction

The Open Silicon Firmware Interface (openSFI) is a specification that defines a common, architecture-neutral interface for host firmware to interact with silicon initialization firmware. Its goal is to enable modular, scalable, and vendor-agnostic integration of silicon into host firmware environments.

openSFI Specification is the Foundational Contract. Host firmware doesn't care how the silicon initialization is implemented. It just needs a clean, stable API contract to call the right function, with the right inputs, and get deterministic outputs.

6.1. Description

openSFI aims to address the fragmentation and fragility in firmware systems by defining a common silicon firmware interface. This design specification focuses on creating a scalable and flexible framework that enhances interoperability across various silicon platforms. Key features include:

- **Unified Interface Definition:** openSFI will standardize interactions between host firmware and silicon vendors, simplifying integration and interaction.
- **Platform Integration:** Streamlined methods to integrate diverse silicon, aligning with Open Compute Project (OCP) principles.
- **Resource Efficiency:** Promoting sustainability by optimizing firmware processes and reducing redundant engineering work across multi-generational platforms.
- **Industry Collaboration:** Encourages collaboration among industry participants to bridge ecosystem gaps while maintaining backward compatibility.

6.2. Scope

openSFI defines a standard interface between host firmware and silicon lifecycle management components, supporting both initialization and post-boot operations.
The openSFI specification defines a C-based interface between host firmware and silicon initialization components (e.g., AMD openSIL, Intel FSP). It is targeted at x86 SoCs for the initial release but designed to be ISA-neutral. openSFI standardizes early boot silicon initialization during the SEC, PEI, and early DXE phases (in UEFI terminology), or equivalent early stages in other firmware frameworks like coreboot.

6.3. Goals

The primary goals of the openSFI are to:

- **Unify** silicon initialization interfaces across vendors.
- **Simplify** host firmware integration.
- **Enable** reuse of host firmware across platforms and vendors.
- **Reduce** engineering cost, duplication, and validation overhead.
- **Promote** sustainability through efficient resource usage and common tooling.

This work is licensed under a Creative Commons Attribution 4.0 International License.
CC BY 4.0

6.4. Interface Compatibility Testing

Interface compatibility testing for openSFI ensures predictable interoperability across diverse silicon platforms while maintaining adherence to established interface definitions.

- **Compatibility Procedures:**
  - Validate API implementation against defined specification, data-types and encoding format of input and output parameters, and return values.
  - Define and execute test cases that confirm cross-platform interoperability, leveraging existing platforms or a vendor-neutral simulated reference implementation to validate integration behavior and performance. This enables consistent benchmarking, helps detect regressions, and ensures that openSFI-compliant integrations meet the intended abstraction guarantees.
  - Execute test cases to ensure no output buffer overflows occur across API calls, alignment requirements are met and APIs can handle malformed or incorrect input parameters and return appropriate error codes.
- **Success Criteria:**
  - Implementation conforms to API contract in terms of data-types of input and output parameters, required encoding formats are strictly met(such as UPCI specified encoding), and return values are compliant with the specification.
  - Host firmware can successfully interface with silicon from multiple vendors without requiring vendor-specific changes in the host firmware.
  - Any non-functional requirements are met like execution performance(E.g.: time-bounded return of API calls) and robustness of implementation ensuring no unwanted side effects such as data corruption occurs.

6.5. Standards Interoperability (Non-Normative) Compliance

openSFI does not establish or replace external industry standards (e.g., ACPI, SMBIOS, UEFI). However, implementations must not preclude standard host firmware behavior or break assumptions made by standards compliant firmware frameworks and operating systems.

In practice, this means:

- **Interoperable Outputs:** Data produced via openSFI (e.g. hardware description, discovery, or telemetry) MUST be emitted in a form that host firmware can reliably translate into standards compliant artifacts without ambiguity or loss of fidelity.
- **No Contract Violations:** openSFI MUST NOT introduce ordering, timing, or data dependencies that conflict with normative behavior required by ACPI/UEFI/SMBIOS (or other applicable standards) in the platform's boot flow.
- **Forward Compatibility:** Where a vendor returns "fragments" or partial descriptors, those fragments MUST be "standard-ready", i.e. structurally sufficient and unambiguous so the host can deterministically assemble fully compliant tables/records.

Here's a succinct example (ACPI): A VAL may return ACPI ready fragments (e.g. CPU topology, memory map deltas, device resources) that are not complete ACPI tables by themselves. These fragments MUST be

This work is licensed under a Creative Commons Attribution 4.0 International License.

CC BY 4.0

formatted and annotated such that the host firmware can compose fully compliant ACPI structures (e.g. MADT, SRAT, SLIT, HEST) without guessing or vendor specific heuristics. Failure to meet this requirement risks breaking OS expectations and is non-conformant with openSFI's interoperability rules.

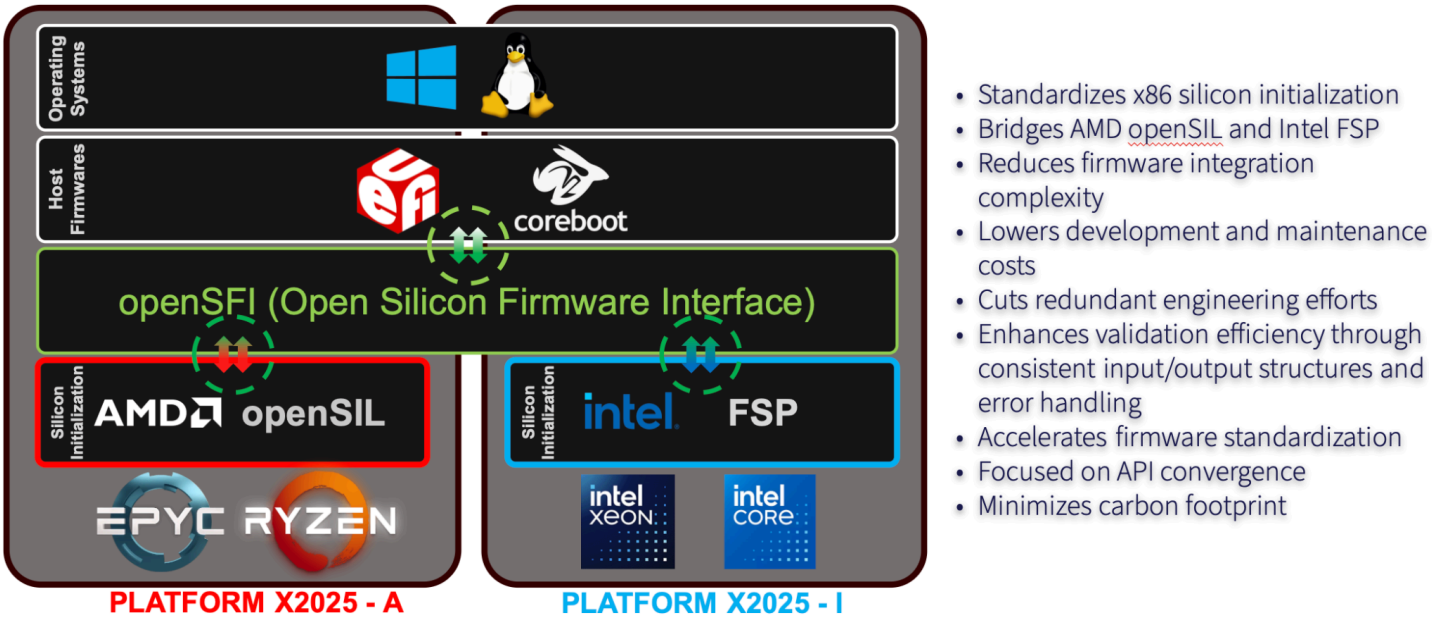
This work is licensed under a Creative Commons Attribution 4.0 International License.

CC BY 4.0

7. Architecture Overview

openSFI adopts a Producer-Consumer API model for its external interface, where openSFI acts as the producer and host firmware operates as the consumer. Internally, the architecture leverages a factory design pattern to provide a clean, consistent interface to silicon initialization code. This factory pattern enables the creation of appropriate initialization objects without exposing the concrete implementation to the client code. By abstracting the creation process, the system achieves a highly scalable interface between host firmware and silicon components, allowing for flexible expansion when new silicon variants are introduced without requiring modifications to the consumer side. This architectural approach effectively decouples the host firmware from the specifics of silicon initialization implementations, promoting maintainability and adaptability as hardware evolves.

Figure 7-1: openSFI Architectural Overview



The openSFI interface provides:

- Standardized entry points for silicon initialization (`InitializeSilicon()`) and runtime operations (`ExecuteSiliconRuntimeOp()`), implemented by a vendor-supplied Library.
- StageID based lifecycle, with a universal, enumerated superset of discrete initialization stages.
- Explicit SoCID targeting to identify the silicon instance and vendor.
- Stage specific Input/Output structures, with optional metadata for cross-vendor compatibility.
- Host allocated output buffers, the openSFI Library only writes within the memory provided by the host.
- Stateless execution model, avoiding persistent Library state between calls.

This work is licensed under a Creative Commons Attribution 4.0 International License.

CC BY 4.0

8. API Definition

The openSFI interface defines two core APIs to separate early boot-time silicon initialization from post-boot lifecycle operations.

The openSFI interface implementation is a responsibility of the silicon vendors via a **Vendor Adaptation Layer (VAL) Library** (or simply **VAL** or **VAL Library**), a Library conforming to this API contract. For a detailed explanation of the VAL concept, and implementation formats, see **Appendix E - Abstraction Layer (Defined Interface Spec)**.

8.1. InitializeSilicon()

This API is the common / unified (across the SoC vendors) entry point for silicon initialization during platform boot stages.

Prototype:

```
int InitializeSilicon(
    void *Input,
    void *Output,
    uint32_t Stage,
    uint32_t SoCID
);
```

Arguments:

Input	Pointer to InitStageID specific silicon configuration input structure
Output	Pointer to InitStageID specific structure with return status, diagnostics data or handoff buffer
InitStageID	Enum indicating the silicon init phase (e.g., PRE_MEM, MEM_INIT, IO_INIT)
SoCID	Vendor-specific identifier for target SoC configuration

Return Value(s):

zero	Success
non-zero	Error (error codes **TBD**)

Notes:

- **SoCID Usage – Forward-Looking Clarification**

This work is licensed under a Creative Commons Attribution 4.0 International License.

CC BY 4.0

The definition and role of SoCID, particularly whether it should be explicitly included as part of input/output structures, is currently under review. A clearer position will be established and reflected in the specification starting with v0.5+, in alignment with design goals for scalability and abstraction.

8.2. ExecuteSiliconRuntimeOp()

This API is used for executing runtime lifecycle operations on silicon post-initialization.

Prototype:

```
int ExecuteSiliconRuntimeOp(
    void *Input,
    void *Output,
    uint32_t RuntimeStageID,
    uint32_t SoCID
);
```

Arguments:

Input	Pointer to RuntimeStageID specific input structure
Output	Pointer to RuntimeStageID specific output struc with return status, diags data or handoff bu
RuntimeStageID	Enum defining the type of lifecycle action (e.g. telemetry, reconfiguration)
SoCID	Vendor-specific identifier for target SoC or board configuration

Returns Value(s):

0	Success
non-zero	Error (error codes **TBD**)

The separation of initialization and runtime APIs improves clarity, composability, and vendor adoption. Implementations may expose both or a subset based on silicon capabilities.

Notes:

- **Scope Clarification for ExecuteSiliconRuntimeOp()**
  The definition and intended scope of ExecuteSiliconRuntimeOp() are not yet finalized in v0.3. It remains under discussion whether this operation is meant to service silicon bound runtime services during OS runtime, or to target a transitional phase after silicon initialization completes

This work is licensed under a Creative Commons Attribution 4.0 International License.

CC BY 4.0

but before OS handoff, or both. This ambiguity is expected to be resolved in v0.5+, along with a more definitive API contract.

This work is licensed under a Creative Commons Attribution 4.0 International License.

CC BY 4.0

9. Initialization Stages

Each of the following stages represent a discrete steps in silicon bring-up:

```
typedef enum {
  SFI_INIT_STAGE_LIB_INIT = 0,  ///> Library initialization (Optional)
  SFI_INIT_STAGE_PRE_MEM,       ///> Pre-memory initialization
  SFI_INIT_STAGE_MEM_INIT,      ///> DRAM training and memory
                                ///> controller init
  SFI_INIT_STAGE_IO_INIT,       ///> PCIe, USB, SATA, and other IO
  SFI_INIT_STAGE_CPU_START,          ///> Core/cluster wake-up
  SFI_INIT_STAGE_SEC_INIT,      ///> Security controller init (e.g.,
                                ///> ASP, ME)
  SFI_INIT_STAGE_FINALIZE,      ///> Final handoff to OS loader
} INIT_STAGES;
```

The stage identifiers listed in above are illustrative placeholders only, and are not intended to represent a finalized or ratified list of silicon lifecycle stages for this revision of the spec. Please refer to an aligned list of InitStages in Appendix C - Unified Enumerated Stages List, which is expected to be defined in v0.5.

These examples are included to facilitate architectural discussions and clarify the role of `InitializeSilicon()` & `ExecuteSiliconRuntimeOps()` across different lifecycle transitions.

A formally ratified, vendor-inclusive superset of initialization stages will be defined in openSFI v0.5, starting with alignment across x86 vendors (AMD, Intel) and extending to additional architectures (e.g., ARM) in future revisions (v1.0+).

9.1. SFI_INIT_STAGE_LIB_INIT - Library Initialization Metadata Query Stage

This ***optional*** stage enables host firmware to query metadata about the VAL implementation, including memory sizing recommendations for stage-specific *Output structures, before invoking any silicon-specific initialization stages.

Stage ID: `SFI_INIT_STAGE_LIB_INIT`

Use Case:

- Enables host firmware to adapt its memory allocation strategies based on the size requirements reported by the silicon vendor's VAL implementation.
- Especially useful in pre-DRAM boot stages where memory is highly constrained.
- Avoids the need for a separate API by reusing the existing InitializeSilicon() interface.

Expected Behavior:

- The VAL will populate the provided Output structure with one or more fields describing:

This work is licensed under a Creative Commons Attribution 4.0 International License.

CC BY 4.0

- Recommended Output buffer sizes for known stages.
- Maximum buffer size required across all supported stages.
- Optional implementation metadata (VAL version, supported SoCIDs, etc.).

Input/Output:

- The host firmware shall pass:
  - A valid SoCID
  - An empty Input structure (or a vendor-defined structure for future extensibility)
  - A valid Output buffer
- If successful, the VAL returns status `SUCCESS` and fills Output.
- If unsupported, the VAL returns `UNSUPPORTED_STAGE`.

Notes:

- Host firmware must not invoke other initialization stages until after this call (if used).
- This stage is optional and may be deprecated in v0.5+ if proven unnecessary.

9.2. Stage Execution Order Manifest (Normative)

The numeric values of INIT_STAGES are identifiers, not an execution sequence. The host firmware determines execution order per SoC by consulting a const, compile-time manifest. This keeps enums append-only (for forward compatibility) while making the call order explicit and vendor-specific.

Requirements

1. **Order defined by manifest**: For each supported SoC, the host firmware SHALL reference a constant, read-only manifest, implemented as a const array of INIT_STAGES, that explicitly encodes the required initialization sequence for that SoC. This manifest is SoC-specific, immutable at runtime, and serves as the sole authoritative definition of execution order. The manifest MUST be included in the platform build and selected based on the SoCID of the target

   e.g.,
   ```
   const INIT_STAGES SoC_Init_Sequence[] = {
       SFI_INIT_STAGE_LIB_INIT,
       SFI_INIT_STAGE_IO_INIT,
       SFI_INIT_STAGE_CPU_START,
       SFI_INIT_STAGE_SEC_INIT,
       SFI_INIT_STAGE_FINALIZE,
   };
   ```

2. **No inference from enum values**: The host and VAL MUST NOT infer order from the numeric values of INIT_STAGES.
3. **Per-SoC selection**: The host SHALL select the appropriate manifest using the SoCID it is targeting.

This work is licensed under a Creative Commons Attribution 4.0 International License.
CC BY 4.0

4. **Idempotence / skip model**: If a listed stage is not supported for a given SoC revision, the VAL SHALL return SFI_STATUS_UNSUPPORTED_STAGE; the host SHALL skip that stage and continue.
5. **Mutability**: Manifests are static (const) and shipped with the platform build; runtime modification is out of scope.

This work is licensed under a Creative Commons Attribution 4.0 International License.

CC BY 4.0

10. Platform Input Configuration Semantics

The openSFI interface defines a standardized contract between platform firmware and silicon-specific initialization logic. As part of this contract, the Input and Output parameters passed to InitializeSilicon() and ExecuteSiliconRuntimeOp() are critical structures through which initialization commands, parameters, and resulting metadata are exchanged.

This chapter outlines the principles for defining, documenting, and handling these structures.

11.1 Purpose and Scope

Each stage of silicon initialization or runtime operation may require a distinct set of input arguments and may produce unique output metadata. Rather than mandate a universal schema across all silicon vendors or stage types, openSFI leaves the structure and interpretation of these Input and Output arguments to the silicon vendor's implementation.

This allows for maximum flexibility while maintaining standardization at the API boundary.

11.2 Vendor Responsibility for Schema Definition

- Each silicon vendor is responsible for defining the per-stage structures used for input and output.
- These structures are documented and published in vendor-provided headers such as:
  - AMDSoCStageSpecificInputStructs.h
  - AMDSoCStageSpecificOutputStructs.h
- These definitions must remain stable for any given SoCID and StageID to preserve interoperability.

11.3 Format of Input and Output Structures

- The Input and Output pointers passed to openSFI APIs are expected to reference native binary structures, laid out per the vendor-defined headers.
- There is no requirement for serialization, self-description, or encoding into interpretable binary formats like CBOR or JSON.
- This minimizes complexity and avoids unnecessary processing overhead in constrained pre-DRAM environments.

Note: Vendors may choose to internally encode or decode binary data for secondary consumers (e.g., µCs, test harnesses etc.), but this is outside the scope of the openSFI interface.

This work is licensed under a Creative Commons Attribution 4.0 International License.

CC BY 4.0

11.4 Relationship to UPCI

The Unified Platform Configuration Interface (UPCI) may eventually serve as a governing mechanism for schema compatibility across layers and tools. However, for the purposes of openSFI v0.3:

- There is no runtime introspection of schemas.
- The layout and interpretation of input/output arguments is assumed to be statically known by both producer (host firmware) and consumer (VAL implementation).

Future versions (v0.5+) may define optional annotations or UPCI linkages, but these are not part of the current release.

11.5 Error Handling for Memory Allocation

Given the lack of dynamic schema introspection, it is the host firmware's responsibility to allocate appropriate memory for the Output structure. If insufficient memory is allocated, the openSFI implementation will return an appropriate status code (e.g., `INSUFFICIENT_MEMORY_ALLOCATED`).

To optionally accommodate cases where pre-sizing the structure is beneficial, a special initialization stage (e.g., `SFI_INIT_STAGE_LIB_INIT`) has been introduced, allowing the host to query memory requirements in a vendor-defined way. This provision remains provisional and may be deprecated in a future revision of the specification should it prove unnecessary.

11.6 Post DRAM Initialization Memory Availability

It is expected that once the host firmware reaches the stage *after* DRAM initialization (e.g., SFI_STAGE_MEM_INIT and beyond), it has access to significantly more system memory. In such scenarios, the host firmware is encouraged to allocate a generously sized output buffer (e.g., 10MB or more) to cover even the most complex vendor-specific output data requirements, such as detailed memory maps, firmware resource descriptors, or ACPI table fragments. This proactive allocation strategy avoids introducing unnecessary API complexity or memory query calls. The burden of managing memory efficiency still lies with the vendor provided silicon library, which should only populate the portion of the output structure relevant to the current hardware configuration.

This work is licensed under a Creative Commons Attribution 4.0 International License.

CC BY 4.0

11. Standards Alignment

openSFI aligns with existing firmware standards (e.g., ACPI, SMBIOS, UEFI) by producing compliant tables and exposing standard descriptors. While not itself a standard, openSFI may influence future industry specifications by identifying ambiguity or proposing extensions.

This work is licensed under a Creative Commons Attribution 4.0 International License.

CC BY 4.0

12. Governance

The openSFI specification is governed and maintained under the OCP Platform Firmware (OPF) Project, with active contributions from AMD, Intel, AMI, Insyde, 9elements and hyperscalers. Oversight is provided by a community-led group with benevolent oversight and consensus-based evolution. Changes to this specification require public review and version control by stakeholders.

This work is licensed under a Creative Commons Attribution 4.0 International License.

CC BY 4.0

Appendix A PascalCase Naming Convention (with Capitalized Acronyms)

PascalCase is a naming convention where the first letter of every word is capitalized and no underscores or separators are used. In this document, we adopt an extended PascalCase style that **fully capitalizes acronyms** to maintain clarity and consistency across hardware-related identifiers.

Purpose:

To ensure clear, concise, and readable naming for types, structures, classes, and modules across the openSFI and UPCI specifications and codebase. These conventions collectively support readable, maintainable, and scalable code design across the openSFI and UPCI ecosystems.

Guidelines:

- Each word begins with a capital letter (e.g., ConfigSchema, BootManager).
- Acronyms (e.g., CPU, USB, PCIe, IFST) are capitalized in their entirety (e.g., CPUTargets, USBDevice).
- Avoid underscores (_) or hyphens (-).

Examples:

- IFSTCPUTargets
- Stage1CPUTargets
- InitializeUSBDevice
- ConfigurePCIeTopology
- ReadECCStatus

1. Naming Convention for Constants

Constants (macros, defines, enum values) should be written in **ALL_CAPS with underscores** separating words. This helps them stand out clearly in source code and align with long-standing firmware and C-language traditions.

Purpose:

To distinguish constants and macro definitions from other symbols, ensuring visibility in low-level firmware environments.

Guidelines:

- Use uppercase letters with underscores: MAX_CPU_COUNT, DEFAULT_TIMEOUT_MS
- Prefix with component or module where necessary: IFST_STAGE1_MASK, UPCI_FIELD_VERSION

This work is licensed under a Creative Commons Attribution 4.0 International License.

CC BY 4.0

Examples:

- MAX_NUM_OF_SOCKETS
- IFST_STAGE1_BITMASK
- DEFAULT_BIOS_BOOT_MODE
- UPCI_CONFIG_VERSION

Enumerated Literals (e.g., true, false) – Style Guidance

For **boolean literals** such as true, false, null, and other language-level keywords or enumerated constants:

Do Not PascalCase or CamelCase. These values **should always remain in lowercase**, as per **language conventions**, even if the rest of your codebase uses PascalCase or CamelCase.

Example:

```
None
bool EnableIFST = true;  // correct
bool EnableIFST = True;  // incorrect (not defined)
```

Why?

- true and false are **language-defined keywords** in C/C++, and their casing is standardized by the language and the compiler.
- Changing their casing will **break compilation**, as the compiler will not recognize True or False unless you explicitly redefine them (which is discouraged and confusing).
- This applies even when adopting strict PascalCase naming for variables, functions, structs, etc.

Additional Tip:

If our project uses custom enumerated types (e.g., typedef enum { Off = 0, On = 1 } State;), those enumerated constants **should follow our naming convention**, i.e., **PascalCase**

2. Naming Convention for Functions, Variables, and Parameters

In alignment with openSFI's coding style and readability goals, all **functions, variables, and parameters** shall adopt **PascalCase**, including **fully capitalized acronyms** (e.g., CPU, IO, IFST). This provides a uniform and visually distinct naming style, making it easier to recognize semantic boundaries and maintain clarity in a multi-vendor codebase.

Rules for Naming

- Use **PascalCase** for all identifiers (no underscores).

This work is licensed under a Creative Commons Attribution 4.0 International License.

CC BY 4.0

- Capitalize **every word**, including **acronyms** (e.g., CPU, IO).
- Avoid abbreviations unless they are widely accepted acronyms.
- Names should be descriptive and unambiguous.

Examples

Category	Bad Example	Good Example
Function	`init_cpu_config()`	`InitCPUConfig()`
Variable	`cpu_status_flag`	`CPUStatusFlag`
Parameter	`target_io_channel`	`TargetIOChannel`
Struct Field	`enable_ifst_iod0`	`EnableIFSTForIOD0`
Constant Ref	`MAX_CORES`	`MaxCores (or kMaxCores if using prefix convention)`

Special Notes

- Anonymous union or struct members should also follow PascalCase for clarity.
- Arrays should be pluralized meaningfully, e.g., CPUList, IOModules.

This work is licensed under a Creative Commons Attribution 4.0 International License.

CC BY 4.0

Appendix B - Clarifying Notes

1. **Stack and Heap Usage** – The openSFI VAL library does not require any explicit memory allocations beyond the buffers allocated for the *Input and *Output arguments. It operates within the same memory and runtime context as the host firmware, using the existing stack and heap organically.
2. **Variable Output Buffer Size** – While the VAL library is not required to know the allocated size of the *Output structure at runtime, and must assume the buffer provided by the host firmware is sufficient for its operation, there is a finite possibility that silicon initialization firmware could inadvertently write beyond the bounds of the allocated buffer. This concern was raised during the v0.30 specification review.
    a. To mitigate this potential risk, we propose adding an explicit indication of the output buffer size to the API interface. To maintain efficiency and preserve the four-argument API limit, this size will be included as an element within the *Input structure rather than as a separate API argument.
    b. The intended usage is as follows: if the VAL detects that the allocated output buffer is insufficient, it must terminate further processing and return control to the caller with the SFI_STATUS_OUT_OF_MEMORY status code.
    c. This requirement will be implemented provisionally. If, by the v1.0 release of the specification, no implementation has triggered this condition during unit testing or validation, the requirement will be removed from the specification.
    d. If the *Output structure is known and fixed in size (e.g., a compile-time constant determined by using the sizeof operator on the associated StageID-specific output structure from the openSFIOutputStructures.h file), this concern does not apply, as the host firmware can deterministically allocate the exact required memory. Such a scenario is expected to cover all fixed-format output structures.
3. **StageIDs** – When defined as an enumerated list, StageIDs must be universal and represent a superset across all SoCIDs and all SoC vendors.
4. **Input Structure Definitions** – These must be StageID specific and universal across all SoC vendors. Metadata can indicate if a given structure element is not applicable to a particular SoC vendor.
5. **Output Structure Definitions** –
    a. If known at build time, these must be universal to all SoC vendors, with non-applicable elements indicated via metadata.
    b. If not known at build time, the host firmware may not be able to interpret the returned data without additional mechanisms. In such cases, we need to identify the scope of these scenarios (e.g., standard hardware descriptor fragments such as e820, ACPI, SMBIOS) and determine whether serialization is warranted, unless we can unify the structure definitions across vendors. This requires further discussion - **Post v0.3 Spec release, expected in v0.5.**
6. **Precedent for Stateless Static Libraries** – When a custom library is statically linked to a C program, its code and data become part of the final executable, sharing the same address space and heap as the host. In common practice, stateless, statically linked libraries, such as c runtime libraries (even the most complex ones), do not require special memory allocation for their internal operation. Temporary buffers are typically handled in processor registers or existing stack space. Only in rare cases requiring large working memory should special allocation be considered, and such exceptions should be

This work is licensed under a Creative Commons Attribution 4.0 International License.
CC BY 4.0

addressed case by case. We should avoid introducing such complexity prematurely, to maintain a simple, predictable, and vendor-agnostic model.

This work is licensed under a Creative Commons Attribution 4.0 International License.

CC BY 4.0

Appendix C - Unified Enumerated Stages List

Place holder for v0.5 content (`InitStages` and `RuntimeOps`)

This work is licensed under a Creative Commons Attribution 4.0 International License.

CC BY 4.0

Appendix D - Supporting Documentation & Guidance

In addition to the openSFI specification, a complete integration ecosystem includes the following complementary documents, each owned and governed appropriately:

1. openSFI FAS (openSFI Firmware Architecture Specification)

This Firmware Architecture Specification (FAS) is the authoritative, version-controlled definition of the openSFI abstraction layer. Maintained by the OCP SPF openSFI Working Group, this document describes the stable API contract, its function prototypes, parameter semantics, expected return codes, and lifecycle rules for invocation. It is intended to remain consistent across silicon vendors and platform firmware environments, ensuring that host firmware developers can rely on a common interface surface. Changes to the FAS follow a governed change-control process to maintain backward compatibility wherever possible. This specification is normative, meaning that any implementation claiming openSFI conformance must adhere fully to its API definitions and behavioral requirements. The FAS defines this contract and serves as the foundation for all other integration and implementation guidance.

- ***Purpose***: Defines the authoritative, versioned contract between host firmware and silicon initialization backends. Specifies all data structures, enumerations, calling conventions, and error codes for InitializeSilicon() and ExecuteSiliconRuntimeOp().
- ***Scope***: Normative and ratified by the OCP OPF openSFI Project Working Group. Changes follow the specification governance process.
- ***Audience***: Host firmware developers, silicon vendors, and platform integrators requiring a stable, silicon-agnostic API.

2. openSFI SSImG (openSFI Silicon Specific Implementer's Guide)

The Silicon-Specific Implementer's Guide (SSImG) is an engineering-focused reference maintained by the Silicon Provider (SiP). This document explains how to implement the VAL for a given SoC family in a way that is compliant with the openSFI FAS. It details how the abstraction layer maps to the SiP's internal silicon initialization sequences and platform control flows, including stage-specific behavior, error handling expectations, and any optimizations possible within the vendor's firmware architecture (e.g., openSIL, FSP).

While the SSImG may be released into the open-source domain at the SiP's discretion, it is primarily an internal development document, written for firmware engineers implementing or modifying the VAL. Because silicon architecture and firmware internals evolve more rapidly than the openSFI spec itself, the SSImG is governed independently by the SiP, allowing it to track product-specific changes without requiring updates to the normative openSFI specification.

- ***Purpose***: Internal developer guide owned by the Silicon Provider (SiP). Describes how the vendor's firmware architecture (e.g., openSIL for AMD, FSP for Intel) implements the openSFI interface. May include build instructions, code flow diagrams, and silicon-specific constraints.

This work is licensed under a Creative Commons Attribution 4.0 International License.

CC BY 4.0

- ***Scope***: Non-normative and changeable at the SiP's discretion to reflect ongoing silicon or firmware evolution.
- ***Audience***: Primarily internal engineering teams at the SiP; optional release to the open-source community at the vendor's discretion.

3. openSFI SSInG UEFI (openSFI SoC-Specific Integrator's Guide for UEFI Host Firmware)

The SoC-Specific UEFI Integrator's Guide describes how to integrate a given silicon vendor's VAL into a UEFI-based platform firmware environment (e.g., EDK2). Maintained by the Silicon Provider, this guide is intended for platform integrators, independent firmware vendors (IFVs), and OEM/ODM engineering teams who need to consume the VAL in UEFI-compliant firmware stacks.

The document covers build integration steps, platform configuration requirements, firmware volume placement, dependency resolution, and debug hooks specific to UEFI. It may also detail interactions with the UEFI DXE, PEI, and SMM phases, along with platform policy settings relevant to silicon bring-up via openSFI. This guide is expected to be public and could be contributed to the Open Compute Project (OCP) to encourage multi-vendor alignment and reuse across the UEFI ecosystem.

- ***Purpose***: Public-facing document maintained by the SiP, describing how UEFI-based host firmware (e.g., EDK2) integrates with the vendor's openSFI-compliant VAL.
- ***Scope***: May be contributed to OCP for transparency and reuse. Updated as needed to reflect changes in UEFI integration flows or silicon lifecycle stages.
- ***Audience***: IFVs, OEMs, and platform integrators building UEFI firmware for systems using the vendor's silicon.

4. openSFI SSInG coreboot (openSFI SoC-Specific Integrator's Guide for coreboot Host Firmware)

The SoC-Specific coreboot Integrator's Guide serves the same role as the UEFI variant but is tailored for integrating a given SiP's VAL into a coreboot-based platform firmware stack. Maintained by the Silicon Provider, it addresses the unique build system, initialization sequencing, and payload hand-off mechanics of coreboot.

The guide describes how to integrate the VAL into coreboot stages (e.g., romstage, ramstage), manage stage-specific buffer allocations, and align with coreboot's device tree and Kconfig structures. It may also cover integration with common payloads (e.g., SeaBIOS, Tianocore) and provide debugging and logging recommendations consistent with both coreboot and the silicon vendor's diagnostics framework. Like the UEFI guide, it is expected to be public and may be contributed to OCP to foster broader ecosystem adoption and cross-platform interoperability.

- ***Purpose***: Public-facing document maintained by the SiP, describing how coreboot-based host firmware integrates with the vendor's openSFI-compliant VAL.
- ***Scope***: May be contributed to OCP for community benefit. Updated in alignment with changes in coreboot or silicon integration flows.

This work is licensed under a Creative Commons Attribution 4.0 International License.

CC BY 4.0

- ***Audience***: Open-source firmware developers, community maintainers, and integrators building coreboot firmware for systems using the vendor's silicon.

This work is licensed under a Creative Commons Attribution 4.0 International License.

CC BY 4.0

Appendix E - Abstraction Layer (Defined Interface Spec)

To achieve the goal of silicon-agnostic host firmware integration, openSFI introduces two closely related architectural constructs:

- Abstraction Layer (defined by the spec)
- Vendor Adaptation Layer (VAL) (implementation-specific shim layer)

These components work together to isolate host firmware from vendor-specific silicon initialization implementations while enabling extensibility and maintainability.

12.1. Abstraction Layer (Defined Interface Spec)

The abstraction layer is formally defined by this specification. It describes a common C-based API for invoking silicon lifecycle operations (initialization and runtime). These interfaces are stable and documented, and include:

- `int InitializeSilicon(...)`
- `int ExecuteSiliconRuntimeOp(...)`

The abstraction layer standardizes the interface surface, enabling consistent API calls from the host firmware (the consumer), regardless of the silicon vendor behind the implementation (the producer).

Example:

```
// Defined by openSFI Spec - implemented by the silicon vendor
int InitializeSilicon(void *Input, void *Output, uint32_t Stage, uint32_t
     SoCID);
```

This function is the contract between host firmware and the silicon initialization backend.

12.2. Vendor Adaptation Layer (VAL / Shim Layer)

The VAL Library is an implementation construct, a thin translation layer developed by silicon vendors (or debatably by platform integrators and / or IFVs). **It adapts the openSFI abstraction to vendor-specific initialization or runtime mechanisms** (e.g., AMD's InitializeAMDSilicon() or Intel's FspSiliconInit()).

The VAL can be delivered as either a binary or source code library that encapsulates silicon specific behavior for initialization and runtime services, while conforming to the unified interface defined by the openSFI specification. In doing so, it ensures consistency and compatibility both across vendors and across that vendor's own generations of silicon, allowing host firmware to interact in a canonical, vendor neutral manner regardless of underlying hardware differences. This canonical model enables predictable integration behavior, minimizes porting complexity, and fosters long term interoperability within the openSFI ecosystem.

This work is licensed under a Creative Commons Attribution 4.0 International License.

CC BY 4.0

The VAL Library is not part of the openSFI specification, but its presence is instrumental during early adoption phases or proof-of-concept (POC) efforts, and may eventually be optimized away as vendors integrate the abstraction directly into their native flows.

Example Platform Integrator Implementation:

```
// openSFIShim.c
int InitializeSilicon(void *Input, void *Output, uint32_t InitStageID,
     uint32_t SoCID) {
    if (IsAMDSoC(SoCID)) {
        return InitializeAmdSilicon(Input, Output, InitStageID, SoCID);
    } else if (IsIntelSoC(SoCID)) {
        return FSPSiliconInit(Input, Output, Stage, SoCID);
    }
    return UNSUPPORTED_SOC; // Unsupported SoC
}
```

In this model:

- The abstraction layer is spec-defined, universal, and fixed.
- The VAL Library is implementation-defined, flexible, and SoC vendor-specific that can be integrated by a Platform Integrator or an IFV to serve platform designs based on more than one silicon vendor SoC.

12.3. Lifecycle of the VAL Library

The VAL Library is expected to evolve:

- **Initial POC Phase:** Serves as a translation layer to map openSFI APIs to vendor-specific APIs.
- **Mid-Adoption:** Some common logic may migrate into host firmware or silicon packages.
- **Mature Stage:** The VAL may disappear entirely as silicon vendors embed native support for openSFI directly into their firmware packages (e.g., openSIL or Intel FSP implementing InitializeSilicon() natively)

While the VAL Library is expected to evolve, a vendor may well choose to jump directly to native integration or remain in the shim model indefinitely if it suits their business and / or architecture needs.

12.4. Design Goals and Benefits

- **Encapsulation:** Host firmware is completely decoupled from vendor-specific APIs (unless the silicon vendor has migrated to the aforementioned ‘Mature Stage’ natively supporting the openSFI interfaces).
- **Portability:** The same host firmware can support different SoCs with minimal changes.
- **Debug Efficiency:** The VAL enables traceability and diagnostics mapping across vendors.
- **Security:** A standardized interface simplifies security auditing and validation of firmware interactions.

This work is licensed under a Creative Commons Attribution 4.0 International License.

CC BY 4.0

12.5. Illustration – Conceptual Mapping

Layer	Purpose	Ownership
Abstraction (API)	Defined contract (e.g., InitializeSilicon)	Spec (openSFI WG)
Vendor Adaptation Layer (VAL)	Translation to SoC-specific API	Silicon Vendor / IFV / Integrator
Vendor Silicon Code	Actual init logic (e.g., FSP/openSIL)	Silicon Vendor

Note: Vendor-specific integration details, including mapping openSFI to native silicon initialization flows (e.g., openSIL, Intel FSP), will be documented in SiP-maintained guides separate from this specification. These include the Silicon-Specific Implementer's Guide (SSImG) and Silicon-Specific Integrator's Guide (SSInG) integration guides for specific host firmware environments such as UEFI or coreboot. This separation allows the openSFI specification to remain stable and vendor-neutral, while enabling each SiP to update their integration guidance as their silicon and firmware stacks evolve. Please see **Appendix D - Supporting Documentation & Guidance** for details.

This work is licensed under a Creative Commons Attribution 4.0 International License.

CC BY 4.0

Appendix F

Checklist for Steering Committee (SC) approval of this Specification (to be completed by contributor(s) of this Spec)

Complete all the checklist items in the table with links to the section where it is described in this spec or an external document .

Item	Status or Details	Link to detailed explanation
Is this contribution entered into the OCP Contribution Portal?	Yes	If no, please state the reason.
Was it approved in the OCP Contribution Portal?	Yes	If no, please state the reason.
Is there a Supplier(s) that is building a product based on this Spec? (Supplier must be an OCP Solution Provider)	Yes or No	List Supplier Name(s)
Will Supplier(s) have the product available for GENERAL AVAILABILITY within 120 days?	Yes or No	If more time is required, please state the timeline and reason for extension request. Please have each Supplier fill out Appendix B.

This work is licensed under a Creative Commons Attribution 4.0 International License.

CC BY 4.0